Entropy-regularized subgame solving in sequential Bayesian games with public actions

Sobhan Mohammadpour
MIT

Samuel Sokota
Carnegie Mellon University

Brandon Kaplowitz
New York University

J. Zico Kolter
Carnegie Mellon University

Noam Brown
OpenAI

Gabriele Farina
MIT

Abstract

Subgame solving is central to scaling equilibrium approximation in large imperfect-information games. We introduce a small set of reusable GPU primitives for high-performance, entropy-regularized subgame solving in public-action Bayesian games (PBGs). Instantiated in Liar's Dice and Heads-up Hold'em, our approach achieves one to two orders of magnitude speedups and substantially lower memory usage compared to baselines.

1 Introduction

Imperfect-information games (IIGs) model multiagent sequential decision making with hidden states and stochasticity, covering settings such as auctions, security, negotiations, and recreational games such as poker and Stratego. These games are ubiquitous in real-world settings, but are challenging to solve. One major challenge stems from the fact that the frequency with which an action is taken changes the value of that action. For example, in poker, the value of bluffing decreases as bluffing becomes more frequent. As a consequence, the backward-induction-based algorithms underlying artificial intelligence for chess (e.g., Stockfish [Stockfish Developers]) and Go (e.g., AlphaGo [Silver et al., 2016]) cannot be safely applied to IIGs, as the dependence of value on action frequency breaks fundamental assumptions of the algorithms. Instead, successful algorithms for IIGs like poker have been based on techniques that decompose the problem into sub-problems — referred to as subgames — which are solved using iterative methods such as counterfactual regret minimization [CFR, Zinkevich et al., 2007] and fictitious play [FP, Brown, 1949].

In the past, these algorithms such as DeepStack [Burch et al., 2014] and ReBeL [Brown et al., 2020] have generally used GPUs only for value-function evaluation, solving subgames on the CPU. The only significant exception to this pattern seems to be Zarick et al. [2020], who provide little to no implementation detail. With the proliferation of GPUs, it is natural to revisit subgame solving on GPUs, to develop more general and reusable primitives. The aim of this work is to distill the operations needed to apply subgame solving into a compact set of composable primitives that enable batched execution on GPUs.

2 Game-theoretic background

We study *public-action Bayesian games* (PBGs): tree-structured[1] sequential games in which a private type (hand) is dealt to each player at the start of the game and all actions are publicly observed. The subclass of PBGs in which there exists only a single possible type is equivalent to tree-structured Markov games. All PBGs can be represented as extensive-form games [EFGs, Kuhn, 1953].

[1] No state is visited twice.

39th Conference on Neural Information Processing Systems (NeurIPS 2025) Workshop: ScaleOPT.

Many games of interest—e.g., Liar's Dice and poker—fit naturally into this template. Others possess phases that can be useful to view as PBGs. For instance, given the distribution of ship placements in Battleship, the main phase can be viewed as a PBG. A similar view holds for the main phases of Stratego and Pokémon Video Game Championships (VGC).

It can be computationally advantageous to view games (or phases of games) as PBGs. Each joint hand and terminal state corresponds to a unique history, and a single hand together with a public state corresponds to an information state. This structure provides a natural way to batch computation along the hand dimension. This amenability to batching contrasts with EFGs, the standard formalization of IIGs.[2]

It is also advantageous to define subgame decompositions in terms of PBG states. As shown by Burch et al. [2014], Brown and Sandholm [2017], extensive-form subgames must be defined with care to avoid leaking information. In contrast, every PBG public state is a valid starting point for a subgame. These subgames are defined using distributions over private hands conditioned on the public states — called *public belief states* (PBSs) — and can be used in concert with entropy regularization [Sokota et al., 2023], among other approaches [Moravčík et al., 2017, Brown et al., 2020] to solve IIGs. In this work we focus on entropy regularized subgame solving, bridging the gap between ReBeL and the framework proposed by Sokota et al. [2023].

3 Notation

A PBG is specified by the following:

1. $\mathcal{P}$, the set of players (e.g., $\{1, 2\}$); we use $c \in \mathcal{P}$ to denote chance player which models exogenous outcomes (e.g., public-card reveals in poker),
2. $\mathcal{H}_i$, the (finite) set of private hands for player i; $\mathcal{H} = \times_{i \in \mathcal{P} \setminus \{c\}} \mathcal{H}_i$ is the set of all possible joint hands,
3. $\mathcal{S}$, the (finite) set of (public) states; $\mathcal{T} \subseteq \mathcal{S}$ is the set of terminal states,
4. $s_0 \in \mathcal{S}$, the initial state,
5. $\mathbb{P}_0 \in \Delta(\mathcal{H})$, the joint distribution over hands,
6. $A : \mathcal{S} \to P(\mathcal{S})$, the legal successor function mapping each state to its (finite) set of children, where P denotes the power-set operator,
7. $\Pi : \mathcal{S} \setminus \mathcal{T} \to \mathcal{P}$, the acting player at each nonterminal state,
8. $T : \mathcal{S} \to \mathcal{S}$, the parent of each state, and
9. $R_i : \mathcal{T} \times \mathcal{H} \to \mathbb{R}$, the terminal payoff for player i, given the state and hands.

In this notation, actions and the transition function are implicitly defined.

A (behavioral) policy $\pi(s, h) \in \Delta(A(s))$ maps a state–hand pair to a distribution over legal successor states. For non-chance players, the legal successor state's distribution depends only on the public state and that player's hand, whereas chance may depend on all players' hands. We use a per-player regularizer $\Omega_i : \Delta \to \mathbb{R}$ that is concave in player i's policy (a bonus) and convex in the opponents' policies (a penalty). We assume $\sum_{i \in \mathcal{P}} \Omega_i = 0$ and $\Omega_c = 0$. A typical choice takes Ω to be scaled entropy. The sequence-form reach $\sigma(s, h; \pi) \in \mathbb{R}_{\geqslant 0}$ is the product of legal successor state's probabilities along the path from the root to state s given hands h under π. Likewise, $\sigma_i(s, h; \pi)$ is the product of the probabilities of legal successor state taken by player i on that path, and $\sigma_{-i}(s, h; \pi)$ is the product for all players other than i.

For two-player zero-sum games (i.e., $\mathcal{P} = \{1, 2, c\}$ and $R_1 = -R_2$), we write $R = R_1$ and $\Omega = \Omega_1$, with player 1 maximizing and player 2 minimizing the reward R. The (hard) value, or expected

[2]The crux of the issue is the discrepancy between the size of the control, pointers or indices of the children and parents, and the work that needs to be done for every node. In EFGs every history is represented by a different node, making it not amenable to batch computation. In contrast, PBGs every node represents the set of all possible state hand pairs, increasing the amount of work that needs to be done for every node. We elaborate on the difficulties of traversing trees in the next section.

return, of a policy $\pi = (\pi_1, \pi_2)$ is

$$\bar{V}^{\pi} = \sum_{h \in \times_i \mathcal{H}_i} \sum_{t \in \mathcal{T}} \mathbb{P}^{\pi}(t, h)\, R(t, h),$$

where $\mathbb{P}^{\pi}(t, h) := \mathbb{P}_0(h)\, \sigma(t, h; \pi)$. The exploitability (loss against an optimal opponent) is

$$\max_{\hat{\pi}_1} \bar{V}^{(\hat{\pi}_1, \pi_2)} \; - \; \min_{\hat{\pi}_2} \bar{V}^{(\pi_1, \hat{\pi}_2)}.$$

Similarly, the soft (regularized) value is

$$V^{\pi} = \bar{V}^{\pi} + \sum_{h \in \mathcal{H}} \sum_{s \in \mathcal{S} \setminus \mathcal{T}} \mathbb{P}^{\pi}(s, h)\, \Omega(\pi(s, h))\,.$$

These values can be computed recursively with a regularized Bellman equation. For player i with private hand h_i, the regularized value under policy π satisfies the recursion

$$V_i^{\pi}(s, h_i) = \begin{cases} \mathbb{E}_h^{\pi}[R_i(s, h) | h_i] & s \in \mathcal{T}, \\ \mathbb{E}_{h, s'}^{\pi}\Big[V_i^{\pi}\big(s', h_i\big) + \Omega_i\big(\pi(s, h)\big) \mid h_i \Big] & \text{o.w.}, \end{cases}$$

where h denotes the joint hands. Note that V^{π} can be calculated as $\mathbb{E}_{h_i}^{\pi}[V^{\pi}(s_0, h_i)]$. The best response (optimal policy against a fixed adversary) can be calculated by maximizing the value of a player while keeping other player's strategy fixed.

Instead of expected values, we use counterfactual values (CFVs), as calculating them does not necessitate expectations over the opponent's hand. We define CFVs as $C_i^{\pi}(s, h_i) = V_i^{\pi}(s, h_i) \sum_{h | h_i} \mathbb{P}_0(h) \sigma_{-i}(s, h)$ where $\sum_{h | h_i}$ refers to the sum over all joint hands h that assign hand h_i to player i. The resulting equation takes the sum over opponent nodes instead of averaging. We leverage counterfactual values to compute soft best responses required for FP.

4 Graphics processing units

A full treatment of GPU architectures is beyond our scope, but a few key facts shape our interest in PBGs. Code is executed in many "warps"[3] in parallel. Although the programming model often presents the illusion of independent threads, execution follows a single-instruction multiple-threads (SIMT) model: all threads in a warp share a single program counter, and control-flow divergence forces the warp to serialize branches. Furthermore, warps achieve highest throughput when accessing contiguous memory, so scattered small reads are inefficient.

Two practical consequences follow for our setting: (i) memory coalescing and regular access patterns are crucial, because global-memory bandwidth and latency dominate our memory-bound kernels; (ii) minimizing warp divergence (e.g., by grouping states with similar branching factor) increases effective throughput.

These considerations suggest that traversing irregular trees is difficult. Indeed, prior work like Goldfarb et al. [2013] introduces alternative traversal modes. In contrast, PBGs largely forgo these issues, since the control (the tree structure) is regular across hands.

5 Main operations

We store batches of subgames as a forest (a collection of trees). All data is indexed first by node (state), then by hand, which allows coalesced access to per-state data. For policies, we store the successor state's probability at its outcome (i.e., we store the probability of going from s to s' at s'). Nodes are laid out so that siblings (children of the same parent) are adjacent and are sorted by depth. This enables efficient batched traversals without ordering concerns. For each node, we store its player, parent, and the indices of its first and last children.

We define two higher-order operations, $\uparrow$ and $\downarrow$. For a reduction function $f : \mathcal{S} \times \big(\times_{N=1}^{\infty} \mathbb{R}^N \big) \to \mathbb{R}$, $f^{\uparrow}$ reduces the children's values into the value at their parent node, proceeding from the bottom

[3] A warp consists of 32 threads on NVIDIA GPUs.

level to the top. Because nodes are depth-sorted, we can apply f in batch to all nodes at each depth and move upward. Similarly, for $f : \mathcal{S} \times \mathbb{R} \to \mathbb{R}$, $f^{\downarrow}$ applies f from parent to child, starting at the roots and proceeding toward the leaves. Note that since the parent is unique, it is not an input of the function. In both cases, f can be executed in batch.

To simplify the notation, we adopt the convention that dropping the last index input of a function means we collect its value over that index, increasing the order (dimension) of its output by one. For example, for a fixed hand h_1,

$$\mathbb{P}_0(h_1)$$

denotes a vector whose entries are $\mathbb{P}_0(h_1, h_2)$ for all h_2. Similarly, $\mathbb{P}_0$ denotes the matrix whose entry at coordinate (h_1, h_2) is $\mathbb{P}_0(h_1, h_2)$. This convention implicitly implies that we assume some ordering over hands.

Sequence-form policies $\sigma_i(s, h)$ can be computed via $f^{\downarrow}$ with

$$f(s, h) = \sigma_i(T(s), h) \begin{cases} 1 & \text{if } \Pi(T(s)) \neq i, \\ \pi\left(s \mid T(s), h\right) & \text{o.w.} \end{cases}$$

For convenience, we define $\sigma_i(s_0, h_i)$ as the marginal of the initial hand distribution, i.e., $\sum_{h|h_i} \mathbb{P}_0(h)$. To reconstruct $\mathbb{P}_0$, we define a fundamental matrix F such that $\mathbb{P}_0(h) = F(h) \prod_i \sigma_i(s_0, h_i)$. We can now express $\mathbb{P}(s)$ in terms of the fundamental matrix F and two marginals, $\sigma_1(s)$ and $\sigma_2(s)$ such that $\mathbb{P}(s, h) = F(h) \prod_i \sigma_i(s, h_i)$. As we show in the next section, the fundamental matrix of many games is highly structured.

Posteriorization To convert counterfactual values to expected values and vice versa, we compute a matrix-vector product between $F \odot \sigma_c(s)$ and $\sigma_i(s)$ where $\odot$ is the element-wise product. In many cases, this can be accelerated by exploiting the structure of F.

Bellman backups Oftentimes the soft Bellman backup has an analytical solution, for instance when $\Omega(\pi)$ is the entropy function $\sum_i \pi_i \log \pi_i$, the solution to $\max_\pi \sum_i c_i \pi_i + \Omega(\pi)$ is the log-sum-exp of c or $\log \sum_i \exp c_i$. In general, we define the convex conjugate of Ω as

$$\Omega^\star(c) = \max_\pi c'\pi + \Omega(\pi).$$

The backup can be implemented with the convex conjugate Ω^* and the $\uparrow$ transformation. However, since we are using counterfactual values not the expected value, we need to first posteriorize the counterfactual values to obtain expected values, calculate the backup and then convert the results back to counterfactual values.

Terminal reward Each game also needs a reward function that evaluates the bilinear payoff weighted by chance. Concretely, we multiply by the matrix

$$L(s, h) = \sigma_c(s, h) \odot R(s, h).$$

As shown by Johanson et al. [2011], this operation can often be done in $\mathcal{O}(\sum_i |\mathcal{H}_i|)$ time per public state, instead of the naive $\mathcal{O}\big(|\mathcal{H}|\big)$, using matrix vector multiplication by $L(s)$. This factorization can often be much smaller than the matrix.

6 Games

6.1 Liar's Dice

Liar's Dice is a bluffing game in which each player rolls a private set of dice. On each turn, a player either makes a claim about the number of dice showing a given face or challenges the previous player. Each new claim must strictly exceed the previous one: either increase the face value (any count) or, at the same face value, increase the count. The round ends when a claim is challenged. If the challenged player was bluffing, they lose a die; otherwise, the challenger loses a die. While the whole game cannot be viewed as a PBG, each round up to the loss of a die can be viewed as a PBG. Since the rounds are independent, policies for each of the rounds can be trained independently and combined. Much of the reinforcement learning (RL) literature focuses on a single "round" instead of the whole game.

Posteriorization The fundamental matrix of Liar's Dice and more generally games with uniform hand distribution, i.e., $\mathbb{P}_0(h_1, h_2) = 1/|\mathcal{H}|$, is the one matrix $F = 1$. For a function $y : \mathcal{H}_2 \to \mathbb{R}$, $(Fy)_j$ is equal to $\sum_i y_i$. Note that since Liar's Dice has no chance node, so the chance reach is also the all matrix i.e., $\sigma_c = 1$.

Terminal reward ReBeL [Brown et al., 2020] introduced linear time terminal reward calculation for Liar's Dice but did not give a description of the procedure in their paper so we provide it here: For any given terminal state with claim $n \times f$ (n dice with face f), we can create a vector whose ith entry is the sum of the reach of the opponent at hands that have i die with face f. We then take the cumulative sum over this vector to obtain a vector whose i index is the sum of all the reaches of the opponent at hands that have at least i die with face f. For every hand of this player and this vector, a simple addition yields the terminal reward.

6.2 Heads-up Hold'em

Heads-up Hold'em is a two-player card game. At the start of the game, the players are dealt two cards, and the game proceeds in rounds. In each round, a certain number of cards[4] are revealed, and the players can either call[5] (match the other player's contribution to the pot and move to the next round unless it is the first action of the round), fold (exit the game and put no more money in the pot), or raise (match the other player's contribution and add more). The game ends when one player folds or the game reaches the last round. In the former case, the nonfolding player wins the pot; in the latter, the player with the higher-scoring hand wins.

Variants such as *limit* and *no-limit* primarily differ in bet sizing and raise caps; these differences do not affect the core mechanics. The techniques described here are GPU adaptations of Johanson et al. [2011]'s ideas.

Posteriorization While more complicated than Liar's Dice, posteriorization is still possible in poker. Let C be the set of cards, $\mathcal{H}_1 = \mathcal{H}_2$, K be a mapping from hands to pairs of cards, and K^- be the inverse set-valued function, mapping from cards to set of hands.

The fundamental matrix F in poker is a binary matrix where index i, j is zero if $K(i)$ intersects with $K(j)$ and one otherwise. Thus, to calculate $[Fx]_j$, we can first take the sum of x_i, subtract the sum of x_i for both cards contained in $K(j)$, and add back x_j. Practically, we calculate the $\sum_{h \in K^-(c)} x_h$ by having each thread calculate the sum of a given card c.

The chance reach $\sigma_c(s, h)$ is zero if the hands intersect with each other or the board, and can be calculated by filtering the vector being multiplied by $F\sigma_c(s, h)$ as hands that intersect are already set to zero by F.

Terminal reward At folds — terminal states where one player has folded — the reward is the full pot, so it can be calculated with posteriorization. In showdowns, the public cards induce a mapping from hands to numbers (potentials) such that the hand with the higher index wins. This implies that we can reorder hands such that the payoffs are grouped in the winning, tying, and losing portions as seen in Figure 1. This structure allows for efficient algorithms for CPU. Either the problem can be formulated as a sparse matrix multiplication [Farina and Sandholm, 2022], or we can start at the weakest hand, and keep the total of opponent sequences that win, tie, and lose against our hand. At each step, we can use these quantities to calculate the reward (by doing a sublinear number of operations for incompatibility removal) and move the hand index by one. Every time we move the hand index by one, we just need to update the 3 quantities with constant amortized operations. Unfortunately, this lends itself poorly to GPU calculation. Instead, we observe that the potential of each hand can be compressed to 150 unique potentials per board and 23 if we fix the board and a card. We first accumulate the potential of each card/card-potential; we then scatter these values on the 150 unique potentials and calculate the cumulative sum in both cases. Using the cumulative sums, we can easily calculate the win, tie, and loss value for each hand and its incompatibilities in constant time in parallel. For added precision, we do the accumulations in double precision. The potentials that we use for poker take only 1326 bytes of space which is much smaller than the terminal matrix.

[4] 0, 3, 1, 1

[5] We assume checking is a form of calling

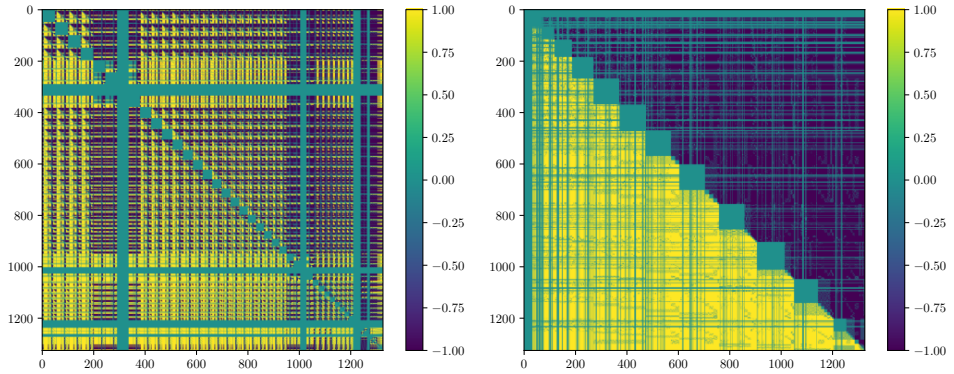

Figure 1: Terminal reward matrix, before multiplication by pot size, for a showdown before and after sorting. Incompatible hands also have a reward of zero, as we calculate the counterfactual value not the expected value.

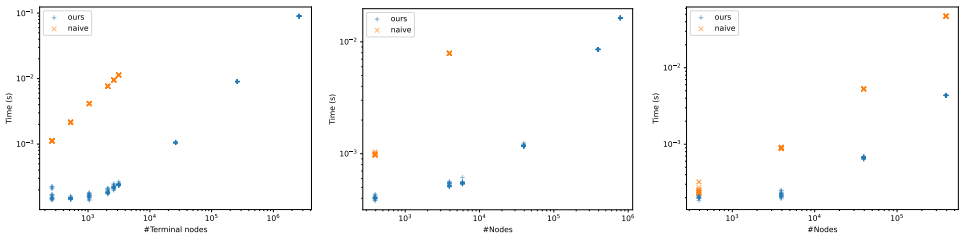

Figure 2: From left to right, time for calculating terminal rewards, soft best responses, and best responses.

7 Experiments

We implement the necessary kernels in CUDA. Each kernel accepts a template parameter for posteriorization. We integrate our code into JAX, as we found it performed better than PyTorch in ergonomics, memory usage, runtime, and compilation time.

We test the proposed ideas for poker on an H200. We use batches of rivers and report the results in Figure 2. We execute each function 20 times, discarding the first 4 calls. The measurement variance is low; the points in the plots are closely clustered. We do not flush the cache between executions, as this reflects a more realistic game-solving pipeline. Overall, we observe one to two orders of magnitude speedup. The improved time for best response computation arises because it is not expressible as efficiently using JAX primitives.

8 Conclusion

In this paper, we explored computational aspects of subgame solving on GPUs for PBGs. We presented practical recipes for efficient GPU implementations. Our experiments showed substantial gains over the naive GPU-based baselines.

Acknowledgments and Disclosure of Funding

SM is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC), 599271-2025. GF is supported by NSF Award CCF-244306, ONR grant N00014-25-1-2296, and an AI2050 Early Career Fellowship.

References

Stockfish Developers. Stockfish chess engine. `https://stockfishchess.org/`.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

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

George W. Brown. Some notes on computation of games solutions. Technical report, Santa Monica, CA, 1949.

Neil Burch, Michael Johanson, and Michael Bowling. Solving imperfect information games using decomposition. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 28, 2014.

Noam Brown, Anton Bakhtin, Adam Lerer, and Qucheng Gong. Combining deep reinforcement learning and search for imperfect-information games. *Advances in neural information processing systems*, 33:17057–17069, 2020.

Ryan Zarick, Bryan Pellegrino, Noam Brown, and Caleb Banister. Unlocking the potential of deep counterfactual value networks. *arXiv preprint arXiv:2007.10442*, 2020.

Harold W Kuhn. Extensive games and the problem of information. *Contributions to the Theory of Games*, 2(28):193–216, 1953.

Noam Brown and Tuomas Sandholm. Safe and nested subgame solving for imperfect-information games. *Advances in neural information processing systems*, 30, 2017.

Samuel Sokota, Ryan D'Orazio, Chun Kai Ling, David J Wu, J Zico Kolter, and Noam Brown. Abstracting imperfect information away from two-player zero-sum games. In *International Conference on Machine Learning*, pages 32169–32193. PMLR, 2023.

Matej Moravčík, Martin Schmid, Neil Burch, Viliam Lisỳ, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael Bowling. Deepstack: Expert-level artificial intelligence in heads-up no-limit poker. *Science*, 356(6337):508–513, 2017.

Michael Goldfarb, Youngjoon Jo, and Milind Kulkarni. General transformations for gpu execution of tree traversals. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2013.

Michael Johanson, Kevin Waugh, Michael Bowling, and Martin Zinkevich. Accelerating best response calculation in large extensive games. In *IJCAI*, volume 11, pages 258–265, 2011.

Gabriele Farina and Tuomas Sandholm. Fast payoff matrix sparsification techniques for structured extensive-form games. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 36, pages 4999–5007, 2022.