Solving imperfect information games on heterogeneous hardware by operation aggregation

Anonymous Author(s)
Affiliation
Address
email

Abstract

Solving large-scale imperfect information game is one of the most challenging tasks in machine learning, the difficulties come both from algorithm design and system support. From the perspective of system, the computation pattern of existing algorithm for solving imperfect information is complex and irregular. As a result, despite GPU provides powerful computing power and becomes the standard on training neural networks, most algorithms for solving imperfect information games still only work on CPU clusters. In this paper, we present a two-phase aggregation procedure to refactor the execution plan for the algorithms designed to solve imperfect information games. This procedure first aggregates isolated scalar operations into vector operations, then it further combines some of those vector operations into matrix operations that can be expressed by Basic Linear Algebra Subprograms (BLAS). We evaluate our methods by running the Counterfactual Regret Minimization (CFR) algorithm to solve two games, Bluff and Heads-Up Flop Hold'em Poker. Results shown that, comparing with the single thread CPU implementation, our aggregation procedure achieved an acceleration ratio of more than 31 times. Furthermore, as the game size increase, the acceleration ratio become higher.

1 Introduction

Extensive-form game with imperfect information is a general framework that can model real-world sequential decision-making problems with imperfect information, like trading, auctions, negotiations, etc. Solving the imperfect information games can have great impact in the real world. Unlike in the research lab, real world problems have high requirements for computation resources due to the problem size. Thus, the implementation must exploit all the computation power of heterogeneous hardware, including CPU, GPU, FPGA and all the other special designed devices.

Currently, the state-of-the-art solution for solving imperfect information games are purely running on CPU clusters [Brown and Sandholm, 2017a]. The reason is that, heterogeneous hardware except CPU are specially designed for a certain type of task, e.g. fast matrix multiplication. Thus, they can not perform operations as efficient and flexible as the CPU. In contrast, in making full use of hardware computing power, systems for solving perfect information game doing very well. The most famous example is AlphaGo, which employs Neural Network (NN) for state generalization and Monte-Carlo Tree Search (MCTS) for searching [Silver *et al.*, 2016, 2017]. The MCTS part runs on CPUs and NN part runs on GPU or Tensor Processing Unit (TPU). In imperfect information games, due to the existence of hidden information, the outcome varies largely given the same history of public actions. NNs are not good at handling this situation, thus it is only used on small-scale games as research attempts [Brown *et al.*, 2018; Moravčík *et al.*, 2017]. Be specific, Libratus [Brown and Sandholm, 2017a,b], the system which beats top human players in the Texas hold'em poker game, is trained

Submitted to 32nd Conference on Neural Information Processing Systems (NIPS 2018). Do not distribute.

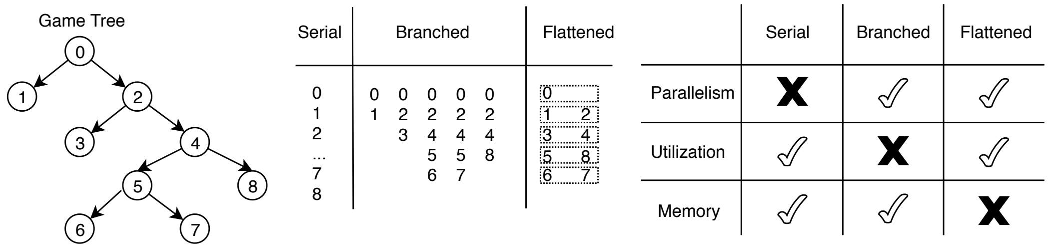


Figure 1: Execution Analysis

over 25 million of CPU hours which is rather time-consuming, while leaving the powerful GPU cards idled.

An algorithm can be seen as a series of basic operations, and an execution plan refers to how to perform those operations, such as on what hardware and in which order. [1] Our key observation is that, on imperfect information games some states are indistinguishable, this character makes it is possible to handle those states in a uniform manner. Based on this observation, we introduce a two-phase aggregation procedure to refactor the default execution plan of the algorithm to exploit the computation power of heterogeneous hardware. Be specific, each possible game situation in the imperfect information game is called a state. Due to the existence of private actions, the player or audience can only specify a set of indistinguishable states, named the information set (infoset), that might be the current state. On the first phase, states belonging to the same infoset are aggregated together, and scalar operations on those states are replaced by a vector operation on that infoset. On the second phase, operations that can be expressed by a dot product or its variation, will be aggregate together and replaced by a single BLAS operation. Thus, the first-phase aggregation exploits the parallel computing power and the second-phase takes the advantage of the hardware's acceleration on BLAS operations. [2] Noteworthy, this procedure did not rely on any implementation level features, such as memory access pattern. Thus, it can be widely used in various scenarios and keeps transparent to the algorithm side researcher and practitioner.

We evaluate the performance of the operation aggregation by running the CFR algorithm on different size of bluff and heads-up flop hold'em poker. We compare three different execution plan: the serial, the flattened and the aggregated. The serial plan runs on a single CPU core, and flattened and aggregated plans run on a single GPU card, details will be described in section2. Results shown that the running time of each round of the aggregated plan is less than one-thirtieth of the serial plan, while only requires the same amount of the GPU memory as the serial plan required for the main memory. On the other hand, despite the flattened plan is as fast as the aggregated plan on the running speed, its memory consumption is much larger, which also keeps it away from solving large-scale problems. Thus, our aggregation procedure is the most suitable plan for solving large-scale imperfect information games.

2 Execution Analysis

In this section, we adopt a subgame of the heads-up flop hold'em poker to analysis the detail of different execution plans. The performance differences between different execution plans are rooted in the hardware characteristics. Taking CPU and GPU as an example, CPU has the advantage of high frequency and out-of-order execution supporting, but it only has several cores. In contrast, GPU has thousands of cores but each of them is much slower and weaker than the CPU core. As a result, we need to design targeted execution plan to make efficient use of the heterogeneous hardware.

We analysis three typical execution plans, namely serial, branched and flattened. On Fig.1, the left part is the game tree of the subgame, the middle part illustrates the visiting sequence of each execution plan on that subgame, and the right part compares the characteristics of these execution plans.

[1]The term "Execution plan" is used by the database community. On other research community e.g. program optimization or multi-threading, they use instance or schedule to name it.

[2]GPU provide special designed processing unit, Tenseo Core, to accelerate the BLAS operations. https://www.nvidia.com/en-us/data-center/tensorcore/

Algorithm 1 Counterfactual Regret Minimization

```
function TreeTraverse(infoset, reaching_prob)
    if infoset is terminal then
        return utility(infoset)
    end if
    reward = [], reward_sum = 0
    for action ∈ valid actions(infoset) do
        next_reaching_prob = reaching_prob * strategy(infoset, action),
        next_infoset = simulate(infoset, action)
        reward[action] = TreeTraverse (next_infoset, next_reaching_prob)
        reward_sum = reward_sum + reward[a]
    end for
    oppo_reward = <reward · reaching_prob[opponent]>
    strategy(I) = RegretMatching(oppo_reward, reaching_prob)
    return reward_sum
end function
```

A serial execution plan simply visits the tree node one by one. It is the by default execution plan, which does not have any parallelism, but are good at hardware utilization and the memory cost is small. In order to improve the parallelism, the most straightforward way is to assign each leaf node an independent execution flow, so different leaf node can be processed in parallel. The parallelism of branched execution plan is good, and the memory cost is acceptable. However, if the game tree is highly unbalanced, cores finished early have to wait for cores finished later to merge their results for back-propagation, which reduced the hardware utilization. The flattened execution plan adopts a Breadth First Search (BFS) to flatten the tree structure into several intervals. As shwon in Fig.1, operations inside an interval are naturally paralleled, and it does not introduce any synchronize barriers. Thus both the parallelism degree and hardware utilization of the flattened execution plan are pretty well. The drawback of the flattened execution plan is its memory consumption. The total size of all those intervals is as large as the size of the whole game tree, while the serial and branched execution plan only consumes the same amount of memory space as the tree depth.

3 Operation Aggregation

In order to solve those drawbacks of existing execution plans, we introduce the operation aggregation method. We first introduce two different descriptions of the imperfect information extensive game for the following discussion, namely game tree and public tree, and the full definition of extensive game are putted in Appendix A. Game tree and public tree are differed by the available information. Game tree describe the full game, where each node of the game tree represents a different state of the game. Public tree [Johanson *et al.*, 2011] describe the game from the perspective of the audience, which he can not distinguish state that only differed by the player's private actions. Each node in public tree corresponds to an infoset that is a collection of all possible states. Fig. 2 illustrate the game tree and public tree.

The operation aggregation consists of two-phase : vectorization and BLAS replacement. The first phase, vectorization, is based on the natural intuition that we can traverse the public tree instead of traverse the monolithic game tree, thus the scalar operations performed on the game tree is equal to a vector operation performed on the public tree. The second phase, BLAS replacement, is to further aggregate the vector operations which can be expressed by the dot product or variation into matrix operations. Then use BLAS operations to perform those matrix operations.

In this section we first prove the invariance before and after applying the operation aggregation. Then, we discuss the detail of applying our two-phase operation aggregation on Counterfactual Regret Minimization (CFR) algorithm.

3.1 Equivalence Prove

Here we first make a mild assumption on the statistics we need to solve the extensive games with imperfect information.

Assumption 1 (Additive Property)**.** *For any infoset I that each history $h_i \in I$ are not distinguishable for player p, the statistics we need satisfy the following equation:*

$$T(I) = \sum_{h_i \in I} w_i T(h_i)$$

where $T(I)$ is some statistics for I and $T_i(h_i)$ are the corresponding statistics for h_i.

Moreover, for any history h, the statistics we need satisfy that

$$T(h) = \sum_{a \in A(h)} w_a T((h,a))$$

where a is a valid action on infoset I, and T_a is the corresponding statistics for (h, a).

It's a natural assumption for hierarchical structures like extensive games and several commonly used statistics to solve the imperfect information game follow this assumption, e.g. the reaching probability $\pi_\sigma(I)$ and utility $u(\sigma, I)$ [3]. Under this assumption, we will further show that either traversing the monolithic game tree or traversing the public tree will give us the same statistics we want.

Corollary 1 (Invariance)**.** *Under Assumption 1, the statistics we get from traversing and back propagate the game tree and public tree will be the same.*

Proof. We proof this corollary with mathematical induction.

First it's obvious that for the terminal nodes of public tree, the statistics from both methods are the same. Then for the non-terminal node I, if its children holds the invariance property, then with back propagation,

$$T(I) = \sum_{a \in A(I)} w_a T(I,a) = \sum_{a \in A(I)} w_a \sum_{h_i \in I} w_i T((h_i,a)) = \sum_{h_i \in I} w_i T(h_i).$$

The second equation holds due to $(h_i, a) \in (I, a)$, and the last equation holds due to the sum operator is exchangeable and our assumption. As the terminal nodes of public tree hold the invariance property, all of the nodes in public tree hold the invariance property. □

As the statistics we collect and back propagate in the public tree are always the same, we need only traverse the public tree, and enumerate all distinguishable states at the terminal node, then back propagate the statistics we need on the public tree, which validates our methods.

3.2 Two-phase operation aggregation

In our method, the first-phase, vectorization, aggregates the scalar operations into vector operations to exploit the parallel computing power. The second-phase, BLAS replacement, extracts the sharing parts from different vector operations, then adopts the BLAS subroutine to replace them.

Given the invariance property, we aggregate operations to perform efficiently on GPU. Specifically, Counterfactual Regret Minimization (CFR) algorithm is used to solve the imperfect information extensive game, the pseudo code is presented in Alg.1. We use a small game as the example to demonstrate the detail of vectorization, the game tree and public tree is shown in Fig. 2. At the beginning of the game, each player gets a private card. The value of the private card is only known to the player himself. Then on each round, each player has two valid actions, quit or bid. Quit will end the game immediately and lose 1 point, bid will continue the game. Note that, all those actions are visible to all players. At the end of the game, whoever has the higher face value of the private card wins the game. The winner gets 2 points, and the opponent loses 2 points.

3.2.1 Vectorization

The right part of Fig. 2 demonstrate the public tree. The public tree represents the game process in the perspective of the audience, who only knows the public actions. Since the private cards of player0

[3] Notice that $\pi_\sigma(I) = \sum_{h_i \in I} \pi_\sigma(h_i)$ and $u(\sigma, I) = \sum_{h_i \in I} \pi_\sigma(h_i) u(\sigma, h_i)$

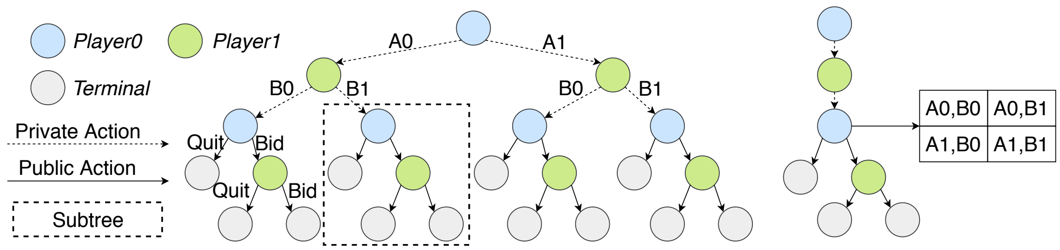


Figure 2: The game tree and public Tree

and player1 are invisible for the audience, all possible states are indistinguishable for the audience. Thus, after each player gets their private cards, each node in the public tree indicates for four states, namely the state matrix. And different sub-trees in the game tree are merged into a single sub-tree in the public tree.

After the private card is deal, each node on the public tree maintains the reaching probability and counterfactual value matrix instead of a scalar. Be specific, the update of the reaching probability for each player is performed by multiply the old value with the strategy for current node. From the perspective of player0, he is not able to distinguish states inside same row of the state matrix, so his strategy for states in the same row remains the same. It makes it possible to use a single vector operation to replace several scalar operations. In the same manner, scalar operations performed in line 6 to line 11 in Alg. 1 are all replaced by equivalence vector operations.

3.2.2 BLAS Replacement

Based on the vectorization, we further aggregate the vector dot product operations into BLAS matrix multiplication. The BLAS are routines that provide standard building blocks for performing the fundamental linear algebra operations. Due to the universality of BLAS operations, various hardware provides extra computing units to performs the BLAS operations. For example, modern GPU adopts the Tensor Core to accelerate those large matrix operations. It is able to perform mixed-precision matrix multiply and accumulate calculations in a single operation.

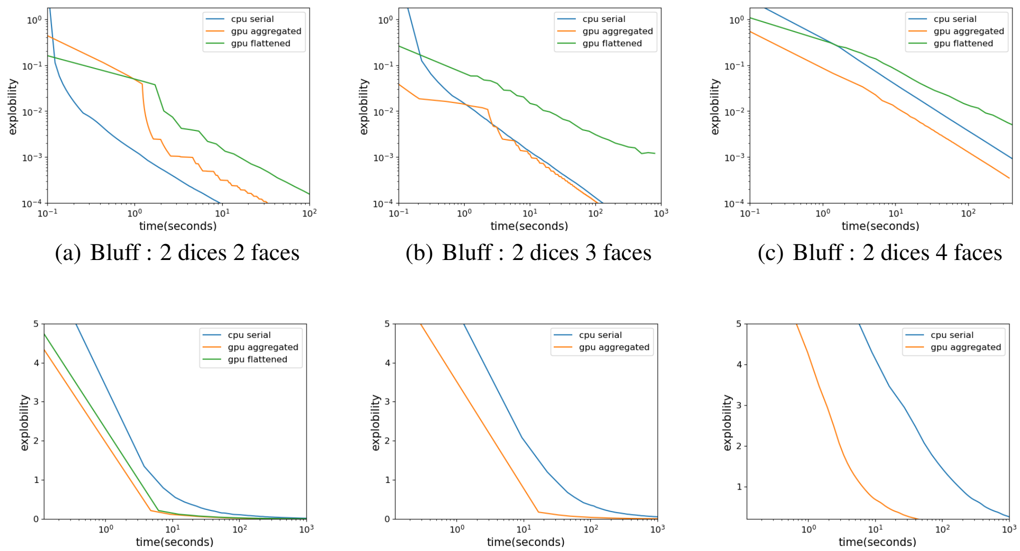
As shown in Fig. 2, each node in the public tree indicates for a matrix, where each element of the matrix is a state. Since each terminal state is mapped to a reward by the utility function, each terminal node of the public tree has a utility matrix. In most two player zero sum extensive games, for a subgame, all the utility matrix can be decomposed into a scalar multiplied by a symbol matrix (assume that positive means player0 wins, and negative means player1 wins). For example, in Fig. 2, terminal nodes of the subtree (framed by the dotted line) share a same symbol matrix. Because all cards are dealt, nodes in that subtree only differs by the total ante on the table. Thus we have the following equation.

$$
\begin{aligned}
< reaching\_prob \cdot utility\_matrix > &= < reaching\_prob \cdot (ante * symbol\_matrix) > && (1)\\
&= < (reaching\_prob * ante) \cdot symbol\_matrix > && (2)
\end{aligned}
$$

Line 12 and 13 in the Alg.1 perform a dot product between the utility matrix and the reaching probability vector, then feed the result to the regret matching subroutine. They can be delayed until the entire subtree for the subgame is traversed. Based on Eq. 3.2.2, those delayed vector-matrix multiplication can be replaced by a matrix-matrix multiplication, which can ben efficiently processed by the BLAS operations.

4 Evaluation

In this section, we evaluate the performance of our method by running CFR to solve bluff and heads-up flop hold'em poker. We use exploitability to measure the strength of the obtained strategy. The exploitability of a strategy is how much utility it will lost when playing with an optimal opponent, which is used to measure the strength of a strategy[Johanson *et al.*, 2011]. Our aggregated execution plan (running on GPU, namely GPU aggregated) is compared with the serial (CPU serial) and flattened (GPU flattened) execution plans.



(a) Bluff : 2 dices 2 faces (b) Bluff : 2 dices 3 faces (c) Bluff : 2 dices 4 faces

(d) Poker : 4 suits 3 cards (e) Poker : 4 suits 4 cards (f) Poker : 4 suits 5 cards

Figure 3: Convergence of different execution plan on different game and size.

The experiment result is shown Fig. 3. In general, the acceleration of GPU version over CPU version increases as game size becomes larger. On small scale games, like 2 dices and 2 faces of the bluff game, CPU serial version performs best. The reason is that the kernel launch will lead to extra time consuming and the clock speed of the GPU is much lower than the clock speed of the CPU.

As the game grows in size, the number of states in each node of the public tree is getting bigger. Operations on those states can be performed in parallel on the GPU. The execution plan who has higher parallelism, e.g. the GPU flattened and GPU aggregated, runs much faster than the CPU serial. However, the GPU flattened execution plan needs to expand the entire game tree before traverse it, which consumes too much memory space. Thus, it failed on the heads-up flop hold'em poker when the game size is bigger than 4 suits 3 card. GPU aggregated obtains additional performance acceleration by the BLAS replacement, so it consumes less time then the GPU flatten.

	Serial	Flattened	Aggregation	Speedup
2 dices 2 numbers	0.00038	0.00090	0.00138	0.27
2 dices 3 numbers	0.00487	0.00374	0.00414	1.17
2 dices 4 numbers	0.13489	0.03229	0.04663	2.89
4 suits 3 cards	0.427	0.061	0.044	9.7
4 suits 4 cards	2.369	-	0.165	14.35
4 suits 5 cards	15.151	-	0.481	31.49

Table 1: The running time (in seconds) and the speedup of aggregated to serial. All measured in a single round.

5 Conclusion and Future Work

In this paper, we show that those naturally un-paralleled tasks, e.g. solving extensive game, can be executed in parallel on GPU by operation aggregation. We propose a novel two-phase aggregation procedure which is based on the algorithm's computation pattern and the operations character. Thus this procedure is independent with the implementation and transparent to algorithm researchers. The experiment result shown that after the operation aggregation, the algorithm runs tens of times faster than before. And as the size of the game continues to increase, there is room for further improvement in the speedup. Future work will focus on combining this operation aggregation procedure with modern CFR variants to further improve the efficiency, while extend this work to real-world large-scale applications.

References

Noam Brown and Tuomas Sandholm. Libratus: the superhuman ai for no-limit poker. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*, 2017.

Noam Brown and Tuomas Sandholm. Superhuman ai for heads-up no-limit poker: Libratus beats top professionals. *Science*, page eaao1733, 2017.

Noam Brown, Adam Lerer, Sam Gross, and Tuomas Sandholm. Deep counterfactual regret minimization. *arXiv preprint arXiv:1811.00164*, 2018.

Michael Johanson, Kevin Waugh, Michael Bowling, and Martin Zinkevich. Accelerating best response calculation in large extensive games. In *Twenty-Second International Joint Conference on Artificial Intelligence*, 2011.

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.

Martin J Osborne and Ariel Rubinstein. *A course in game theory*. MIT press, 1994.

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.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

Martin Zinkevich, Michael Johanson, Michael Bowling, and Carmelo Piccione. Regret minimization in games with incomplete information. In *Advances in neural information processing systems*, pages 1729–1736, 2008.

A Definition and Notation

We present the extensive game and Counterfactual Regret Minimization algorithm to introduce concepts and notations used in our paper.

A.1 Extensive Games with Imperfect Information

Extensive game is a standard framework to describe the sequential decision-making problem with multiple agents. We here firstly define an extensive game formally, introducing the notation we use throughout the paper.

Formally, an extensive game has the following components [Osborne and Rubinstein, 1994]: a finite player set N; a chance player c. Chance player is introduced to handle the uncertainty in the environment; a finite set H of sequence, each member of H is a history, which is an action sequence taken by the players (including chance player), $A(h) = \{a : (h, a) \in H\}$ are the available actions after a non-terminal history. The empty sequence is in H and every prefix of a non-empty sequence in H is also in H. $Z \subseteq H$ denote the terminal history set, each of its elements is not a prefix of any other sequences; a function P that assigns to each non-terminal history (each member of $H \setminus Z$) a member of $N \cup \{c\}$. P is the player function, $P(h)$ being the player who takes an action after the history h. If $P(h) = c$ then chance player determines the action taken after the history h; the utility functions u_i for each player $i \in N$ that mapping the terminal states Z to $\mathbb{R}$.

Notice that extensive games can be represented with the tree structure due to the hierarchical nature of history set H. Each node on the this tree, namely a state s, corresponds to a history h [4], and each edge starts from a certain node represents for a valid action under the corresponding state. In this paper, we use the term *game tree* to denote such tree structure.

In an extensive game, players select the action with their strategy i.e. a probability simplex over available actions given their private information. Formally, a strategy of player i is a function σ^i which assigns h a distribution over $A(h)$ if $P(h) = i$. A strategy profile σ consists of the strategy for each player, i.e. $\sigma^1, \cdots, \sigma^N$. We'll use σ^{-i} to denote all the strategies except σ^i.

In games with imperfect information, actions of other players are partially observable to a player $i \in [N]$. So for player i, the game tree can be partitioned into disjoint infosets, $\mathcal{I}^i$. That is, two histories $h, h' \in I \in \mathcal{I}^i$ are not distinguishable to player i. Thus, σ_i should assign the same distribution over actions to all histories in an infoset $I \in \mathcal{I}^i$. So that, with little abuse of notations, we let $\sigma^i(I)$ denote the strategy of player i on infoset $I \in \mathcal{I}^i$.

For better understanding of infoset, here we introduce an intuitive example that, in Poker games, each player is only able to see his own private cards and all played public cards, while the private card of the opponent player is invisible to him. Thus the player can only make decision with his private card and all played public cards.

Moreover, let $\pi_\sigma(h)$ denote the probability of arriving at a history h if the players take actions according to strategy σ. Obviously, we can decompose $\pi_\sigma(h)$ into the product of each player's contribution, i.e., $\pi_\sigma(h) = \prod_{[N]\cup\{c\}} \pi^i_\sigma(h)$. Similarly, we can define $\pi_\sigma(I) = \sum_{h\in I} \pi_\sigma(h)$ as the probability of arriving at an infoset I and $\pi^i_\sigma(I)$ denote the corresponding contribution of player i. Let $\pi^{-i}_\sigma(h), \pi^{-i}_\sigma(I)$ denote the product of the contributions of all players except player i.

A.2 Counterfactual Regret Minimization

Counterfactual regret minimization (CFR) is now the state-of-the-art algorithm for solving extensive games with imperfect information. We first introduce the concept of regret for player i, which is defined as:

$$R^i_T := \max_{\sigma^i} R^i_T(\sigma^i) := \sum_{t=1}^{T} u^i(\sigma^i, \sigma^{-i}_t) - \sum_{t=1}^{T} u^i(\sigma^i_t, \sigma^{-i}_t)$$

Counterfactual regret minimization works based on the observation that, the time-averaged strategy $\bar{\sigma}^i_T(I) = \frac{\sum_t \pi^i_{\sigma_t}(I)\sigma^i_t(I)}{\sum_t \pi^i_{\sigma_t}(I)}$ achieves ϵ-NE if $\frac{1}{T}R^i_T < \frac{\epsilon}{2}$, which is the objective of solving extensive

[4]Unless otherwise specified, the term *state* denotes the node on the game tree in this paper.

games. Directly apply regret minimization algorithms need to deal with trajectories, which is exponential in state number. [Zinkevich *et al.*, 2008] bounded the original regret by the summation of immediate regret:

$$R_T^i < \frac{1}{T} \sum_t \sum_{I \in \mathcal{I}^i} \pi_{\sigma_t}^{-i}(I)(u^i(\sigma_t|_{I \to \sigma(I)}, I) - u^i(\sigma_t, I)),$$

where $\sigma_t|_{I \to \sigma(I)}$ is the strategy which selects action according to σ at infoset I and according to σ_t at other infoset. Then CFR use a standard regret minimization algorithm called *regret matching* to minimize this upper bound. Notice that to calculate the counterfactual regret and apply regret matching, we need to traverse the whole game tree, which is still time-consuming and hard to execute in parallel.