Hierarchical Abstraction, Distributed Equilibrium Computation, and Post-Processing, with Application to a Champion No-Limit Texas Hold'em Agent

Noam Brown, Sam Ganzfried, and Tuomas Sandholm
Computer Science Department
Carnegie Mellon University
{nbrown, sganzfri, sandholm}@cs.cmu.edu

ABSTRACT

The leading approach for solving large imperfect-information games is automated abstraction followed by running an equilibrium-finding algorithm. We introduce a distributed version of the most commonly used equilibrium-finding algorithm, counterfactual regret minimization (CFR), which enables CFR to scale to dramatically larger abstractions and numbers of cores. The new algorithm begets constraints on the abstraction so as to make the pieces running on different computers disjoint. We introduce an algorithm for generating such abstractions while capitalizing on state-of-the-art abstraction ideas such as imperfect recall and earth-mover's distance. Our techniques enabled an equilibrium computation of unprecedented size on a supercomputer with a high inter-blade memory latency. Prior approaches run slowly on this architecture. Our approach also leads to a significant improvement over using the prior best approach on a large shared-memory server with low memory latency. Finally, we introduce a family of post-processing techniques that outperform prior ones. We applied these techniques to generate an agent for two-player no-limit Texas Hold'em that won the 2014 Annual Computer Poker Competition, beating each opponent with statistical significance.

Categories and Subject Descriptors

I.2.11 [**Distributed Artificial Intelligence**]: Multiagent Systems; J.4 [**Social and Behavioral Sciences**]: Economics

General Terms

Algorithms, Economics, Theory

Keywords

Game Theory, Distributed AI, Imperfect Information

1. INTRODUCTION

The leading approach for creating strong agents for large imperfect-information games—which is used by all of the strongest Texas Hold'em (TH) poker agents—is to first create a sufficiently small strategic approximation of the full game, called an *abstraction*, then to apply an *equilibrium-finding algorithm* [27, 14] to the abstraction, and finally to apply *post-processing* techniques [11, 23, 6, 4] to obtain a strategy in the original game from the approximate equilibrium of the abstraction. Initially abstractions were created manually [24, 2], while nowadays they are created by algorithms [8, 9, 10, 26, 18, 5]. The equilibrium-finding algorithm used by today's strongest TH agents is a Monte Carlo version of the counterfactual regret minimization algorithm (MCCFR) [21]. That algorithm involves repeatedly sampling chance outcomes and actions down the tree, and updating regret and average strategy values that are stored at each information set.

On a shared-memory architecture, MCCFR can be parallelized straightforwardly. True shared-memory architectures typically come with relatively little memory and relatively few cores, however, and it would be desirable for scalability to be able to run on architectures that have more memory (in order to be able to run on larger, more detailed abstractions) and more cores (for speed). However, on distributed architectures and supercomputers with high inter-blade[1] memory access latency, straightforward MCCFR parallelization approaches lead to impractically slow runtimes because when a core does an update at an information set (extensive-form games and information sets therein are formally defined in Appendix B) it needs to read and write memory with high latency. A second issue in MCCFR (even on a shared-memory architecture) is that different cores working on the same information set may need to lock memory, wait for each other, possibly over-write each others' parallel work, and work on out-of-sync inputs. Our approach solves the former problem and also helps mitigate the latter issue.

To obtain these benefits, our algorithm creates an information abstraction that allows us to assign different components of the game tree to different blades so the trajectory of each sample only accesses information sets located on the same blade. At a high level, the first stage of our hierarchical approach is to cluster public information at some early point in the game (public flop cards in the case of poker[2]), giving a global basis for distributing the rest of the game into non-overlapping pieces; then our algorithm conducts clustering of private information. A key contribution is the specific way to cluster the public information. As we will detail in Section 2, two prior abstraction algorithms motivated

[1]Such supercomputers consists of *blades*, which are themselves computers that are plugged into racks. A core can access memory on its blade faster than memory on other blades—seven times faster on the computer we used. On regular distributed systems, the difference between local and remote memory access is even greater.

[2]For rules of Texas Hold'em poker, we refer the reader to `http://en.wikipedia.org/wiki/Texas_hold_em`.

by similar considerations have been developed for poker by others [26, 15], but ours differs in that it does not use hand-crafted poker features, is applicable to the large, and does not have the conceptual weaknesses from which they suffer.

We developed an equilibrium-finding algorithm that can be applied to this abstraction. It is a modified version of external-sampling MCCFR [21]. Applied to TH, it samples one pair of preflop (i.e., first betting round) hands per iteration. For the later betting rounds, each blade samples public cards from its public cluster and performs MCCFR within each cluster. Our algorithm weights the samples to remove bias. Ours is similar to the algorithm of Jackson [15]. However, we implement MCCFR instead of chance-sampled CFR, and split only based on public information (chance actions) rather than players' actions. Another related prior approach used vanilla CFR (which converges significantly slower in practice) and split based only on players' actions (which does support nearly as much parallelization) [16].

The new abstraction and equilibrium-finding algorithms enabled an equilibrium computation of unprecedented size on a supercomputer with high inter-blade memory access latency. Experiments also show that this run outperforms the strongest prior approach executed on a large shared-memory server with low memory latency but fewer cores.

Finally, post-processing techniques have been shown to be useful to mitigate the issues from overfitting to one's abstraction and approximate equilibrium finding. We introduce a family of post-processing techniques that outperform prior ones. Our techniques combine 1) the observation that rounding action probabilities mitigates the above-mentioned issues [6], 2) the new observation that similar abstract actions should be bucketed before such rounding so that fine-grained action discretization (aka action abstraction) does not disadvantage those actions, and 3) the new observation that biasing toward actions that reduce variance is helpful in a strong agent and our experiments show that this increases expected value as well.

We applied all of the above-mentioned techniques to generate an agent for two-player no-limit TH (NLTH). It won the 2014 Annual Computer Poker Competition (ACPC), beating each opponent with statistical significance.

2. ABSTRACTION ALGORITHM

The first contribution of this paper is a new hierarchical abstraction algorithm. It is domain independent, although in many places of the description we present it in the context of poker for concreteness. In order to enable distributed equilibrium finding, it creates an information abstraction that assigns disjoint components of the game tree to different blades so that sampling in each blade will only access information sets that are located on that blade.

At a high level, the first stage of our hierarchical abstraction algorithm is to cluster public information at some early point in the game (public flop boards, i.e., combinations of public flop cards, in the case of TH), giving a global basis for distributing the rest of the game into non-overlapping pieces. Then, as a second stage our algorithm conducts clustering of information states (that can include both public and private information) in a way that honors the partition generated in the first stage.

As an example, suppose that in the first stage we cluster public flop boards into 60 buckets. Suppose bucket 4 contains only the boards AsKhQd and AsKhJd. Then we cluster all private hands for each betting round, starting with the flop, i.e., the second round (we assume the abstraction for the preflop round has already been computed—the strongest agents, including ours, use no abstraction preflop). We perform abstraction over full (five-card) flop hands separately for each of the 60 blades. For blade 4, only the hands for which the public board cards are AsKhQd or AsKhJd are considered (for example, 5s4s-AsKhQd and QcJc-AsKhJd). There are 2,352 such hands. If we allowed an abstraction at the current round with 50 private buckets per blade, we would then group these 2,352 hands into 50 buckets (using some abstraction algorithm; we discuss ours in detail later). We then perform a similar procedure for the third (aka turn) and fourth (aka river) rounds, ensuring that the hands for each blade are limited only to the hands that contain a public flop board that was assigned to that blade in the first stage of the algorithm.

A game has *perfect recall* if, informally, no player ever forgets information that he knew at an earlier point in the game. This is a useful concept for several reasons. First, certain equilibrium-finding algorithms can only be applied to games with perfect recall [19, 14]. Second, other equilibrium-finding algorithms, such as CFR [27] and its sampling variants, have no theoretical guarantees in games that have imperfect recall, though they can still be applied. (One notable exception is recent work giving a theoretical guarantee of the performance of CFR in one class of imperfect-recall games called well-formed games [20].) And third, Nash equilibria are not even guaranteed to exist in general in behavioral strategies in games with imperfect recall.

Despite these limitations, poker agents using abstractions with imperfect recall have consistently been shown to outperform agents that use perfect recall abstractions [26]. Intuitively, perfect-recall abstractions force agents to distinguish all information at a later round in the tree that they were able to distinguish at an earlier round, even if such a distinction is not very significant at the later round. For example, if an agent can distinguish between Kh3c and Kh4c in the preflop round (as is the case in the abstractions of the best agents), then a perfect-recall abstraction would force them to be able to distinguish between Kh3c on a KsJd9h flop, and Kh4c on the same flop, despite the fact that the 3c vs. 4c distinction is extremely unlikely to play a strategic role in the hand. On the other hand, with imperfect recall, agents are not forced to remember all of these distinctions simply because they knew them at a previous round, and are free to group any hands together in a given round without regard to what information was known about them in prior rounds of the abstraction. The most successful prior abstraction algorithms use imperfect recall [18, 5].

Unfortunately, running CFR on imperfect-recall abstractions on a machine with high inter-blade memory access latency can be problematic, since regrets and strategy values at different buckets along a sample may be located on different blades. We now describe in detail our new approach that enables us to produce strong abstractions for this setting. Our approach requires players to remember certain information throughout the hand (public flop bucket), but does not force players to distinguish between other pieces of information that they may have been able to distinguish between previously (if such distinctions are no longer relevant). Thus, our approach achieves the benefits of imperfect recall to a large extent (though not the flexibility of full imperfect recall) while achieving partitioning of the game into disjoint pieces for different blades to work on independently.

2.1 Main Abstraction Algorithm

Our main abstraction algorithm, Algorithm 1, which is domain independent, works as follows. Let $\hat{r}$ be the special round of the game where we perform the public clustering. For the initial $\hat{r} - 1$ rounds, we compute a (potentially imperfect-recall) abstraction using an arbitrary algorithm A_r for round r. For example, in poker the strongest agents use no abstraction in the preflop round (and even if they did use abstraction for it, it would not require public clustering and could be performed separately). Next, the public states at round $\hat{r}$ are clustered into C buckets. The algorithm for this public clustering is described in Section 2.2. Once this public abstraction has been computed, we compute abstractions for each round from $\hat{r}$ to R over all states of private information separately for each of the public buckets that have been previously computed. These abstractions can be computed using any arbitrary approach, A_r. For our poker agent, we used an abstraction algorithm that had previously been demonstrated to perform well as the A_r's [18].

Algorithm 1 Main abstraction algorithm

Inputs: number of rounds R; round where public information abstraction is desired $\hat{r}$; number of public buckets C; number of desired private buckets per public bucket at round r, B_r; abstraction algorithm used for round r, A_r

```
for $r = 1$ to $\hat{r} - 1$ do
    cluster information states at round $r$ using $A_r$
cluster public information states at round $\hat{r}$ into $C$ buckets
(e.g., using Algorithm 2)
for $r = \hat{r}$ to $R$ do
    for $c = 1$ to $C$ do
        cluster private information states at round $r$ that
        have public information in public bucket $c$ into $B_r$
        buckets using abstraction algorithm $A_r$
```

2.2 Algorithm for Computing Abstraction of Public Information

The algorithm used to compute the abstraction of public information at round $\hat{r}$ is shown as Algorithm 2. For TH, this corresponds to computing a bucketing of the public flop boards. To do this, we need a distance function $d_{i,j}$ between pairs of public states (or, equivalently, a similarity function $s_{i,j}$ that can be transformed into a distance function). We use this distance function to compute the public abstraction using the clustering algorithm described in Section 2.3.

Two prior approaches have been applied to abstract public flop boards. One uses poker-specific features that have been constructed manually [15]. The second, due to Waugh et al., uses k-means clustering with L_2 distance over transition tables that were constructed from a small perfect-recall *base abstraction* with 10 preflop buckets and 100 flop buckets [26]. The entry $T[f][i][j]$ in the table gives the probability of transitioning from preflop bucket i to flop bucket j in the abstraction when the public flop board is f. In addition to potentially prohibitive computational challenges of scaling that approach to large base abstractions (such as the one we will use, which has 169 preflop and 5,000 flop buckets), there are also conceptual issues, as the following example illustrates. Consider the similar public flop boards AhKs3d and AhKs2d. Suppose the base abstraction does not perform abstraction preflop and places 4c3s-AhKs3d and 4c2s-AhKs2d into the same flop bucket, (which we would expect, as they are very similar—both have bottom pair with a 4 "kicker"), say bucket 12, while it places 4c3s-AhKs2d and 4c2s-AhKs3d into bucket 13 (these hands are also very similar—the worst possible non-pair hand with a "gutshot" straight draw). Suppose 4c3s is in bucket 7 preflop and 4c2s is in bucket 8. Then the transition table for AhKs2d would have value 0 for the probability of transitioning from preflop bucket 7 into flop bucket 12, while it would have value 1 for transitioning from preflop bucket 8 into flop bucket 12 (and the reverse for AhKs3d). So the L_2 distance metric would maximally penalize the boards for this component, despite the fact that they should actually be considered very similar based on this component, since they map hands that are extremely similar to the same bucket. Our new approach accounts for this problem by building a distance function based on how often public boards result in a given flop bucket in the base abstraction for *any* private cards (not necessarily the same exact private cards, as the prior approach has done).

We have developed an efficient approach that was able to use the strong 169-5,000-5,000-5,000 imperfect-recall abstraction as its base. We refer to this abstraction as A. The algorithm is game independent, and pseudocode (that is not specific to poker) is presented in Algorithm 2. As in Waugh's approach described above, we first compute a transition table T that will be utilized later in the algorithm, though our table will contain different information than theirs. For concreteness, and to demonstrate the implementation used by our agent so that it can be replicated, we will describe how the table is contructed in the context of TH poker.

We first construct a helper table called PublicFlopHands. The entry PublicFlopHands[i][j] for $1 \leq i \leq 1,755$, $1 \leq j \leq 3$ gives the j'th public flop card corresponding to index i, using a recently developed indexing algorithm that accounts for all suit isomorphisms [25] (there are $\frac{52 \cdot 51 \cdot 50}{6} = 22,100$ total public flop hands, but only 1,755 after accounting for all isomorphisms). We specify one such canonical hand for each index. Next, using this table, we create the transition table T, where the entry $T[i][j]$ for $1 \leq i \leq 1,755$, $1 \leq j \leq 5,000$ gives the number of private card combinations for which a hand with public flop i transitions into bucket j of the abstraction A, which has $B = 5,000$ buckets. This is computed by iterating over all public flop indices, then looking up the canonical hand in PublicFlopHands, and iterating over the $\frac{49 \cdot 48}{2} = 1,176$ possible private card combinations given that public flop hand. We then construct the 5-card flop hand by combining the two private cards with the given public flop hand, look up the index of this hand (again using Waugh's indexing algorithm), and then look up what bucket A places that flop hand index into. Thus, the creation of the transition table involves iterating over $1,755 \cdot 1,176 = 2,063,880$ combinations, which can be done quickly.

In poker-independent terms, $T[i][j]$ stores how often public state i will lead to bucket j of the base abstraction, aggregated over all possible states of private information. In contrast, Waugh's table stores separate transition probabilities for each state of private information.

We would like our distance function to assign a small value between public states that are frequently grouped into the same bucket by A, since we already know A to be a very strong abstraction. We compute distances by iterating over the B (private) buckets in round $\hat{r}$ of A. We initialize a variable $s_{i,j}$ which corresponds to the similarity between i and j to be zero. For each bucket b, let c_i denote the number of private states with public state i that are mapped to b under A (and similarly for c_j). For example, suppose i corresponds to the public flop board of AsQd6h and $b = 7$. Then c_i would denote the number of private preflop card combinations (x,y), such that the flop hand xy-AsQd6h is placed in bucket 7 under A. We then increment $s_{i,j}$ by the minimum of c_i and c_j. For example, if $c_i = 4$ and $c_j = 12$, this would mean that i and j are *both* placed into the current bucket b four times. Then the distance $d_{i,j}$ is defined as $\frac{V-s_{i,j}}{V}$, which corresponds to the fraction of private states that are not mapped to the same bucket of A when paired with public information i and j.[3]

Algorithm 2 Algorithm for computing abstraction of public information

Inputs: number of public buckets C; number of public states M; number of private information sets per public state V; prior abstraction A with B buckets; transition table T for public states into buckets of A; clustering algorithm L

```
for $i = 1$ to $M - 1$ do
    for $j = i + 1$ to $M$ do
        $s_{i,j} \leftarrow 0$
        for $b = 1$ to $B$ do
            $c_i \leftarrow T[i][b]$, $c_j \leftarrow T[j][b]$, $s_{i,j}$ += min($c_i$, $c_j$)
        $d_{i,j} \leftarrow \frac{V-s_{i,j}}{V}$
Cluster the $M$ public states into $C$ clusters using $L$ with distance function $d$
```

For our application of Algorithm 2 to poker, the number of public buckets we used is $C = 60$, the total number of private states for each public state is $V = 1,176$, and $B = 5,000$ as described above. The full number of public flop boards after accounting for all suit isomorphisms is $M = 1,755$. Thus, to compute all of the distances we must iterate over $\frac{BN(N-1)}{2} =$ 7.7 billion triples. This can be performed quickly in practice, since for each item we only need to perform lookups in the precomputed transition table.

2.3 Public Abstraction Clustering Algorithm

Given the distance function we have computed, we next perform the clustering of the public states into C public clusters, using the procedure shown in Algorithm 3. The initial clusters c^0 are computed by applying k-means++ [1], using the pairwise point distance function $d_{i,j}$, which is taken as an input. The k-means++ initialization procedure only requires knowing distances between data points, not distances from a point to a non-data-point. Next, for each iteration t, we iterate over all points i. We initialize clusterDistances to be an array of size K of all zeroes, which will denote the distance between point i and each of the current clusters. We then iterate over all other points $j \neq i$, and increment clusterDistances$[c^{t-1}[j]]$ by $d_{i,j}$. Once we have iterated over all values of j, we let $c^t[i]$ denote the cluster with smallest distance from i. If no clusters changed from the clustering at the previous iteration, we are done. Otherwise, we continue this procedure until T iterations have been performed, at which point we output $c^T[i]$ as the final abstraction.

Algorithm 3 Clustering algorithm for public abstraction

Inputs: Number of public states to cluster M; desired number of clusters K; distances $d_{i,j}$ between each pair of points; number of iterations to run T

```
Compute initial clusters $c^0$ (e.g., using k-means++)
for $t = 1$ to $T$ do
    for $i = 1$ to $M$ do
        clusterDistances ← array of size $K$ of zeroes
        for $j = 1$ to $M$, $j \neq i$ do
            clusterDistances$[c^{t-1}[j]]$ += $d_{i,j}$
        $c^t[i]$ ← cluster with smallest distance
    if no clusters were changed from previous iteration then
        break
```

This algorithm only takes into account distances between pairs of data points, and not distances between points in the space that are not data points (such as means). Clustering algorithms that are designed for metric spaces, such as k-means, are not applicable to this setting.[4]

3. EQUILIBRIUM-FINDING ALGORITHM

To solve the abstract game, one needs an algorithm that converges to a Nash equilibrium. The most commonly used equilibrium-finding algorithm for large imperfect-information extensive-form games is counterfactual regret minimization (CFR) and its extensions. We review CFR and the formal notation of extensive-form games in the appendix.

There is a large benefit to not needing to sample all actions at every iteration of CFR, and the variants that selectively sample more promising actions more often are Monte Carlo CFR and Pure CFR. The external sampling variant of Monte Carlo CFR (MCCFR) converges faster than Pure CFR in practice but requires twice as much memory [7]. We build our equilibrium-finding algorithm starting from MCCFR because it converges faster and given that we are able to run on distributed architectures, we are no longer memory constrained.

MCCFR works by sampling opponent actions and chance nodes down the game tree (while exploring all our actions),

[3]Note that d is not a distance metric. It is possible to have $d_{i,j} = 0$ for boards that are different, if the boards send the same number of preflop hands into each flop bucket in A. It also fails the triangle inequality. For example, suppose public state i is always mapped to bucket 1 under A, state j is always mapped to bucket 2, and state k is mapped with bucket 1 with probability 0.2, mapped to bucket 2 with probability 0.2, and mapped to bucket 3 with probability 0.6. Then $d_{i,j} = 1$, $d_{i,k} = 0.2$, and $d_{j,k} = 0.2$. Thus, we view d as an arbitrary matrix of distances rather than viewing the space as a metric space. This will affect selection of the clustering algorithm, which is described in Section 2.3.

[4]We could have used the k-medoid algorithm (though it has a significant computational overhead over our approach, both in terms of running time and memory), or used the objective of minimizing the average distance of each point from the points in a cluster (rather than the sum). It would be interesting to explore the effect of using different choices for the clustering objective on abstraction quality. We chose the sum objective because it is computationally feasible and gives a clustering with clusters of more balanced sizes than the average objective.

and updating regret and average strategy for each information set, using regret minimization [21]. This is problematic on a machine with high inter-blade memory access latency because the information sets traversed on a single sample (aka iteration) of play can be located on different blades. On the supercomputer we used, for example, accessing memory on the same blade takes 130 nanoseconds, while accessing memory on different blades takes about one microsecond.

As discussed in the previous section, our new abstraction addresses this issue by ensuring that (for the flop through river rounds in the case of TH) all information sets encountered in the current MCCFR iteration are stored on the same blade (i.e., the blade that the public flop was assigned to in the first stage of the abstraction algorithm).

We developed a modification of MCCFR specifically for architectures with high inter-blade memory access latency. It designates one blade as the "head" blade, which is used to store the regrets and average strategies for the top part of the game tree (preflop round in TH). The algorithm begins by sampling private information and conducting MCCFR on the head blade. When an action sequence is reached that transitions outside the top of the game tree (to the flop in TH), the algorithm will send the current state to each of the C child blades. Each child blade then samples public information from its public bucket, and continues the iteration of MCCFR. Once all the child blades complete their part of the iteration, their calculated values are returned to the head blade. The head blade calculates a weighted average of these values, weighing them by the number of choices of public information (possible flops in TH) that they sampled from. This ensures that the expected value is unbiased. The head node then continues its iteration of MCCFR, repeating the process whenever the sample exits the top part (a flop sequence is encountered), until the iteration is complete. Pseudocode of the detailed algorithm appears in Algorithm 4.

In practice, rather than communicating with the child nodes every time sampling passes beyond the top part of the tree (i.e., a flop sequence is encountered in TH), we instead use a two-pass approach. On the first pass, we only record which continuation (flop) sequences were encountered. These sequences are then sent to the child blades, so they can calculate values for those sequences; the child blades work in parallel, but within each child blade the continuation sequences assigned to that blade are handled one after another. The head blade then does a second pass that is identical to the first, except that values returned from the child blades are used whenever a the sample gets beyond the top part of the tree (i.e., the flop is reached in TH).

Our algorithm encounters the inter-blade latency whenever the head node sends data to the cluster blades, and again when receiving the responses. This only amounts to less than a millisecond per MCCFR iteration. Each iteration takes about 15 milliseconds, so this latency overhead is negligible. In settings where this overhead were significant, one can easily make it negligible by having the child blades take more samples on each iteration, thereby increasing the ratio of time spent sampling to time spent on latency.

Since the head node can only proceed after receiving a response from all the cluster blades, some clusters may be idle for a significant amount of time if their MCCFR iterations complete faster than other blades'. This happens despite the fact that our abstraction algorithm evenly divides the game tree among the child blades: on some blades the current strategies computed by MCCFR are such that the path of play ends sooner (e.g., by folding in poker).

Algorithm 4 Our equilibrium-finding algorithm

```
for all histories $h$ at the end of the first part of the tree do
// combinations of the players' preflop hands
    for all clusters $C_n$, $n \neq 0$ do // public (flop) clusters
        $|F_{n,h}| \leftarrow$ number of public samples in $C_n$ given $h$
for all information sets $I$ and actions $a$ do
    regret $r_I[a] \leftarrow 0$
    cumulative strategy $s_I[a] \leftarrow 0$
loop // Keep iterating
    for all $p \in N, p \neq c$ do // Players other than chance
        Iter($\emptyset$, $p$, $C_0$)

function ITER(History $h$, Player $p$, Cluster $C$)
    if $h \in Z$ then // Terminal state
        return $u(h)$
    else if $P(h) = c$ then // Chance node
        Draw action $a \in A(h)$ according to $f_c(\cdot|h)$
        if $C = C_0$ and $(h, a) \notin TopOfTree$ then
            $\vec{u} \leftarrow 0$
            for all $C_n \in Clusters$ do
                $\vec{u} \leftarrow \vec{u} + |F_{n,h}| \cdot Iter((h,a), p, C_n)$
            $\vec{u} \leftarrow \vec{u} / \sum_n |F_{n,h}|$ // Remove bias
        else
            $\vec{u} \leftarrow Iter((h,a), p, C)$
    else if $P(h) = p$ then
        for all $a \in A(h)$ do // Traverse all actions
            $Pr(a) \leftarrow \frac{\max\{r_I[a],0\}}{\sum_{a'} \max\{r_I[a'],0\}}$ // Regret matching
            if $C = C_0$ and $(h, a) \notin TopOfTree$ then
                $\vec{u'}[a] \leftarrow 0$
                for all $C_n \in ChildClusters$ do
                    $\vec{u'}[a] \leftarrow \vec{u'}[a] + |F_{n,h}| \cdot Iter((h,a), p, C_n)$
                $\vec{u'}[a] \leftarrow \vec{u'}[a] / \sum_n |F_{n,h}|$ // Remove bias
            else
                $\vec{u'}[a] \leftarrow Iter((h,a), p, C)$
            $\vec{u} \leftarrow \vec{u} + Pr(a) \cdot \vec{u'}[a]$
        for all $a \in A(h)$ do
            $r_I[a] \leftarrow r_I[a] + u'_p[a] - u_p$ // Update regret
    else// Sample an action
        $\vec{\sigma}_I \leftarrow \frac{\max\{\vec{r}_I,0\}}{\sum_{a'} \max\{\vec{r}_I,0\}}$
        $\vec{s}_I \leftarrow \vec{s}_I + \vec{\sigma}_I$ // Update cumulative strategy
        Draw action $a \in A(h)$ from $\vec{\sigma}_I$
        if $C = C_0$ and $(h, a) \notin TopOfTree$ then
            $\vec{u} \leftarrow 0$
            for all $C_n \in Clusters$ do
                $\vec{u} \leftarrow \vec{u} + |F_{n,h}| \cdot Iter((h,a), p, C_n)$
            $\vec{u} \leftarrow \vec{u} / \sum_n |F_{n,h}|$ // Remove bias
        else
            $\vec{u} \leftarrow Iter((h,a), p, C)$
    return $\vec{u}$
```

In more detail, the algorithm begins by sampling private information and conducting MCCFR on the head blade. When an action sequence is reached that transitions beyond the top part of the tree (i.e., transitions to the flop in Texas Hold'em), the algorithm sends the current state to each of the K child blades $C_1, C_2, ..., C_K$. Each child blade C_k then samples public information from its public bucket (i.e., a flop from the valid flops F_k assigned to it), and continues the iteration of MCCFR. Once all the children blades complete their part of the iteration, their calculated values $\vec{u}_k$ are returned to the head blade. The head blade calculates a weighted average of these values, weighing them by the number of choices of public information (possible flops in Texas Hold'em) that they sampled from:

$$\vec{u} = \frac{\sum_{k=1}^{K} F_k \vec{u}_k}{\sum_{k=1}^{K} F_k}$$

This ensures that the expected value is unbiased, that is, in expectation each flop is weighed equally. The head node then continues its iteration of MCCFR, repeating the process whenever the sample exits the top part (a flop sequence is encountered), until the iteration is complete.

In practice (unlike shown in the pseudocode), rather than communicating with the child nodes every time sampling passes beyond the top part of the tree (i.e., a flop sequence is encountered in Texas Hold'em), we instead use a two-pass approach. On the first pass, we only record which continuation (flop) sequences were encountered. These sequences are then sent to the child blades, so they can calculate values for those sequences; the child blades work in parallel, but within each child blade the continuation sequences assigned to that blade are handled one after another. The head blade then does a second pass that is identical to the first, except that values returned from the child blades are used whenever a the sample gets beyond the top part of the tree (i.e., the flop is reached in Texas Hold'em).

Within each child blade—i.e., each child cluster—we actually have, and use, multiple cores (not shown in the pseudocode for simplicity). Whenever a child cluster is reached, each core is given the same inputs but uses a different random number seed to select which public sample (public flop in Texas Hold'em) from within the cluster to work on, and how to randomly sample actions thereunder according to MCCFR. Given the nature of the game, the cores will do redundant work with very low probability, and iterates in different parts of the cluster will be stale by at most one iteration. (Another choice would be to lock parts of the tree within the cluster to prevent cores from working on the same information sets, but that would introduce overhead, and does not seem warranted at least in Texas Hold'em.)

4. NEW FAMILY OF POST-PROCESSING TECHNIQUES

Post-processing is important in solving imperfect-information games. In games where the action spaces are very large, action abstraction is typically used to select only some actions (e.g., bet sizes in poker) to include in the abstraction. However, the opponent may use actions that are not part of the abstraction. This begets the need to map the opponent's actions back into the abstract game. Throughout our experiments we used the leading reverse mapping approach, the pseudo-Harmonic mapping [4], which has been adopted broadly among the top NLTH agents over the last two years.

Post-processing techniques have also been shown to be useful for mitigating the issue of overfitting the equilibrium to one's abstraction and the issue that approximate equilibrium finding may end up placing positive probability on poor actions.[5] Two approaches have been studied, *thresholding* and *purification* [6]. In thresholding, action probabilities below some threshold are set to zero and then the remaining probabilities are renormalized. Purification is the special case of thresholding where the action with the highest probability is played with probability 1 (ties are broken uniformly at random).

We observe that combining reverse mapping and thresholding leads to the issue that discretizing actions finely in some area of the action space disfavors those actions because the probability mass from the equilibrium finding gets diluted among them. To mitigate this problem, we propose to *bucket* abstract actions into similarity classes for the purposes of thresholding (but not after thresholding). For example, in no-limit poker any bet size is allowed up to the number of chips a player has left. In a given situation our betting abstraction may allow the agent to fold, call, bet 0.5 pot, 0.75 pot, pot, 1.5 pot, 2 pot, 5 pot, and all-in. If the action probabilities are (0.1, 0.25, 0.15, 0.15, 0.2, 0.15,0,0,0), then purification would select the call action, while the vast majority of the mass (0.65) is on betting actions. In this example, our approach—detailed below—would make a pot-sized bet (the highest-probability bet action).

Finally, we observe that biasing toward conservative actions that reduce variance (e.g., the fold action in poker) is helpful in a strong agent (variance increases the probability that the weaker opponent will win). Our experiments will show that preferring the conservative "fold" action in TH increases expected value as well. One reason may be that if an agent is uncertain about what should be done in a given situation (the equilibrium action probabilities are mixed), the agent will likely be uncertain also later down that path and it may be better to end the game here instead of continuing to play into a part of the game where the agent is weak.

Our new post-processing technique combines all the ideas listed above. It first separates the available actions into three categories: fold, call, and bet. If the probability of folding exceeds a threshold parameter, we fold with probability 1. Otherwise, we follow purification between the three options of fold, call, and the "meta-action" of bet. If bet is selected, then we follow purification within the specific bet actions.

Clearly, there are many variations of this technique—so it begets a family—depending on what threshold for definitely using the conservative action (fold) is used, how the actions are bucketed for thresholding, what thresholding value is used among the buckets, and what thresholding value is used within (each of possibly multiple) meta-actions.

5. EXPERIMENTS

We experimented on the version of two-player no-limit Texas Hold'em (NLTH) used in the ACPC, which has 10^{165} nodes [17] in its game tree.

We used our new abstraction algorithm to create an information abstraction with 169 preflop buckets, 60 public flop buckets, and 500 private buckets for the flop, turn, and river for each of the public flop buckets, that is, 30,000 total private buckets for each of the three postflop rounds. Our

[5]It is easy to see that each of the post-processing techniques discussed in this section can increase the exploitability of the agent. However, opponents may have a hard time determining how to exploit the agent, especially in complex imperfect-information games. In the ACPC, post-processing has been shown to be beneficial in practice [6].

O1	O2	O3	O4	O5	O6	O7	O8	O9	O10	O11	O12	O13
261 ± 47	121 ± 38	21 ± 16	33 ± 16	20 ± 16	125 ± 44	499 ± 68	141 ± 45	214 ± 57	516 ± 61	980 ± 34	1474 ± 180	1819 ± 111

Table 1: Win rate (in mbb/h) of our agent in the 2014 Computer Poker Competition against opposing agents.

action abstraction had 6,104,546 nodes (including leaves). In total, our abstract game then had $5.49 \cdot 10^{15}$ nodes (including leaves), $6.6 \cdot 10^{10}$ information sets (not including leaves), and $1.8 \cdot 10^{11}$ *infoset actions* (a new measure of game size that is directly proportional to the amount of memory that CFR uses [17]). This is six times larger than the largest abstractions used by prior NLTH agents—and, to our knowledge, the largest imperfect-information game ever tackled by an equilibrium-finding algorithm. This scale was enabled by our new, distributed approach.

We ran our equilibrium-finding algorithm for 1,200 hours on a supercomputer (Blacklight) with a high inter-blade memory access latency using 961 cores (60 blades of 16 cores each, plus one core for the head blade), for a total of 1,153,200 core hours. Each blade had 128 GB RAM.

The results from the 2014 ACPC against all (anonymized) opponents are shown in Table 1. The units are milli big blinds per hand (mbb/h), and the ± indicates 95% confidence intervals. Our agent beat each opponent with statistical significance, with an average win rate of 479 mbb/h.

We also compared our algorithm's performance to using the prior best approach on a low-latency shared-memory server with 64 cores and 512 GB RAM. This is at the upper end of shared-memory hardware commonly available today. The algorithm run on the server used external sampling MC-CFR on a 169-5,000-5,000-5,000-bucket imperfect-recall card abstraction (this size was selected because it is slightly under the capacity of 512 GB RAM). We computed that information abstraction using the state-of-the-art non-distributed abstraction algorithm [5]. We used the same action abstraction as for the distributed case. The abstract game then had $1.5 \cdot 10^{14}$ nodes (including leaves), $1.1 \cdot 10^{10}$ information sets (not including leaves), and $3.1 \cdot 10^{10}$ infoset actions.

We benchmarked both against the two strongest agents from the 2013 competition, Figure 1.[6] The new approach outperformed the old against both agents for all timestamps tested. So, it is able to effectively take advantage of the additional distributed cores and RAM.

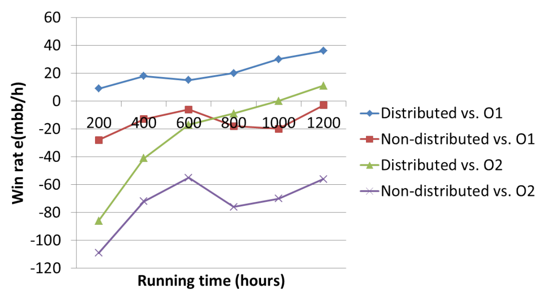

Figure 1: Win rates over time against the two strongest agents from the 2013 poker competition.

[6]Both our distributed and parallel algorithms were evaluated in play with purification (except no post-processing of the first action), which had been shown to perform best among prior techniques. This is also one of the benchmarks we evaluate in the experiments presented in Table 2.

We also studied the effect of using our new post-processing techniques on the final strategies computed by our distributed equilibrium computation. We compared using no threshold, purification, a threshold of 0.15,[7] and using the new technique with a threshold of 0.2.[8] We tested against the same two strongest agents from the 2013 competition. Results are shown in Table 2. The new post-processor outperformed the prior ones both on average performance and on worst observed performance.

	O1	O2	Avg	Min
No Threshold	+30 ± 32	+10 ± 27	+20	+10
Purification	+55 ± 27	+19 ± 22	+37	+19
Thresholding-0.15	+35 ± 30	+19 ± 25	+27	+19
New-0.2	+39 ± 26	+103 ± 21	+71	+39

Table 2: Win rate (in mbb/h) of several post-processing techniques against strongest 2013 agents.

6. CONCLUSION

We introduced a distributed version of the most commonly used algorithm for large-scale equilibrium computation, counterfactual regret minimization (CFR), which enables CFR to scale to dramatically larger abstractions and numbers of cores. Specifically, we based our algorithm on external-sampling Monte Carlo CFR. The new algorithm begets constraints on the abstraction so as to make the pieces running on different computers disjoint. We introduced an algorithm for generating such abstractions while capitalizing on state-of-the-art abstraction ideas such as imperfect recall and the earth-mover's-distance similarity metric. Our techniques enabled an equilibrium computation of unprecedented size on a supercomputer with a high inter-blade memory latency. Prior approaches run slowly on this architecture. Our approach also leads to a significant improvement over using the prior best approach on a large shared-memory server with low memory latency. Finally, we introduced a family of post-processing techniques that outperform prior ones. We applied these techniques to generate an agent for two-player no-limit Texas Hold'em. It won the 2014 Annual Computer Poker Competition, beating each opponent with statistical significance.

The techniques are game independent. While we presented them for a setting that does not require abstraction before the public information arrives, and there is only one round of public information, they can be extended to settings with any sequence of interleaved public and private information delivery—while keeping the information sets on different blades disjoint. Also, while we presented techniques for two levels in the distribution tree (one blade to handle the top part and the rest split disjointly among the other blades), it is easy to see how the same idea can be directly extended to trees with more than two levels of blades.

[7]This value was a prior benchmark [6]. Our exploratory data analysis concurred that it is a good choice.

[8]This was a good choice based on exploratory analysis, and it performed clearly better than 0.1 against both opponents.

REFERENCES

[1] D. Arthur and S. Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2007.

[2] D. Billings, N. Burch, A. Davidson, R. Holte, J. Schaeffer, T. Schauenberg, and D. Szafron. Approximating game-theoretic optimal strategies for full-scale poker. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI)*, 2003.

[3] N. Cesa-Bianchi and G. Lugosi. *Prediction, learning, and games*. Cambridge University Press, 2006.

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

[5] S. Ganzfried and T. Sandholm. Potential-aware imperfect-recall abstraction with earth mover's distance in imperfect-information games. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, 2014.

[6] S. Ganzfried, T. Sandholm, and K. Waugh. Strategy purification and thresholding: Effective non-equilibrium approaches for playing large games. In *Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, 2012.

[7] R. Gibson. *Regret Minimization in Games and the Development of Champion Multiplayer Computer Poker-Playing Agents*. PhD thesis, University of Alberta, 2014.

[8] A. Gilpin and T. Sandholm. A competitive Texas Hold'em poker player via automated abstraction and real-time equilibrium computation. In *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 2006.

[9] A. Gilpin and T. Sandholm. Better automated abstraction techniques for imperfect information games, with application to Texas Hold'em poker. In *Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, 2007.

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

[11] A. Gilpin, T. Sandholm, and T. B. Sørensen. A heads-up no-limit Texas Hold'em poker player: Discretized betting models and automatically generated equilibrium-finding programs. In *Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, 2008.

[12] G. J. Gordon. No-regret algorithms for online convex programs. *Advances in Neural Information Processing Systems*, 19:489, 2007.

[13] A. Greenwald, Z. Li, and C. Marks. Bounds for regret-matching algorithms. In *ISAIM*, 2006.

[14] S. Hoda, A. Gilpin, J. Peña, and T. Sandholm. Smoothing techniques for computing Nash equilibria of sequential games. *Mathematics of Operations Research*, 35(2):494–512, 2010. Conference version appeared in WINE-07.

[15] E. Jackson. Slumbot NL: Solving large games with counterfactual regret minimization using sampling and distributed processing. In *AAAI Workshop on Computer Poker and Incomplete Information*, 2013.

[16] M. Johanson. Robust strategies and counter-strategies: Building a champion level computer poker player. Master's thesis, University of Alberta, 2007.

[17] M. Johanson. Measuring the size of large no-limit poker games. Technical report, University of Alberta, 2013.

[18] M. Johanson, N. Burch, R. Valenzano, and M. Bowling. Evaluating state-space abstractions in extensive-form games. In *Proceedings of the International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS)*, 2013.

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

[20] M. Lanctot, R. Gibson, N. Burch, M. Zinkevich, and M. Bowling. No-regret learning in extensive-form games with imperfect recall. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2012.

[21] M. Lanctot, K. Waugh, M. Zinkevich, and M. Bowling. Monte Carlo sampling for regret minimization in extensive games. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS)*, 2009.

[22] M. J. Osborne and A. Rubinstein. *A Course in Game Theory*. MIT Press, 1994.

[23] D. Schnizlein, M. Bowling, and D. Szafron. Probabilistic state translation in extensive games with large action sets. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI)*, 2009.

[24] J. Shi and M. Littman. Abstraction methods for game theoretic poker. In *CG '00: Revised Papers from the Second International Conference on Computers and Games*, London, UK, 2002. Springer-Verlag.

[25] K. Waugh. A fast and optimal hand isomorphism algorithm. In *AAAI Workshop on Computer Poker and Incomplete Information*, 2013.

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

[27] M. Zinkevich, M. Bowling, M. Johanson, and C. Piccione. Regret minimization in games with incomplete information. In *Proceedings of the Annual Conference on Neural Information Processing Systems (NIPS)*, pages 905Ű–912, 2007.

APPENDIX

A. REGRET MATCHING

In regret-minimization algorithms, a strategy is determined through an iterative process. While there are a number of such algorithms (e.g., [13, 12]), this paper will focus on a typical one called *regret matching* (specifically, the polynomially weighted average forecaster with polynomial degree 2). We will now review how regret matching works, as well as the necessary tools to analyze it.

A normal-form (aka bimatrix) game is defined as follows. The game has a finite set N of players, and for each player $i \in N$ a set A_i of available actions. The game also has:

- For each player $i \in N$ a payoff function $u_i : A_i \times A_{-i} \to \Re$, where A_{-i} is the space of action vectors of the other agents except i. Define $\Delta_i = \max_{\langle a_i, a_{-i}\rangle} u_i(a_i, a_{-i}) - \min_{\langle a_i, a_{-i}\rangle} u_i(a_i, a_{-i})$ and define $\Delta = \max_i \Delta_i$.
- For each player i, a *strategy* σ_i is a probability distribution over his actions. The vector of strategies of players $N \setminus \{i\}$ is denoted by σ_{-i}. We define $u_i(\sigma_i, \sigma_{-i}) = \sum_{a,a_{-i}} p_{\sigma_i}(a) p_{\sigma_{-i}}(a_{-i}) u_i(a, a_{-i})$. We call the vector of strategies of all players a *strategy profile* and denote it by $\sigma = \langle \sigma_i, \sigma_{-i}\rangle$. Moreover, the *value* of σ to player i is defined as $v_i(\sigma) = u_i(\sigma_i, \sigma_{-i})$.

Let σ_i^t be the strategy used by player i on iteration t. The *instantaneous regret* of player i on iteration t for action a is

$$r_{t,i}(a) = u_i(a, \sigma_{-i}^t) - u_i(\sigma^t, \sigma_{-i}^t) \tag{1}$$

The *regret*, for player i on iteration T for action a is

$$R_{T,i}(a) = \frac{1}{T}\sum_{t=1}^{T} r_{t,i}(a) \tag{2}$$

Also, $R_{T,i} = \max_a\{R_{T,i}(a)\}$.

In the regret-matching algorithm, a player simply picks an action in proportion to his positive regret on that action, where positive regret is $R_{t,i}(a)_+ = \max\{R_{t,i}(a), 0\}$. Formally, at each iteration $t+1$, player i selects actions $a \in A_i$ according to probabilities

$$p_{t+1}(a) = \begin{cases} \frac{R_{t,i}(a)_+}{\sum_{a' \in A_i} R_{t,i}(a')_+}, & \text{if } \sum_{a' \in A_i} R_{t,i}(a')_+ > 0 \\ \frac{1}{|A|}, & \text{otherwise} \end{cases} \tag{3}$$

As shown in [3, p. 10], one can bound regret as

$$R_{T,i} \le R_{T,i+} \le \frac{\Delta_i \sqrt{|A_i|}}{\sqrt{T}} \tag{4}$$

Thus, as $T \to \infty$, $R_{T,i+} \to 0$.

B. EXTENSIVE-FORM GAMES

An extensive form game is defined as follows [22].

- A finite set N of players.
- A finite set H of sequences, the possible histories of actions, such that the empty sequence is in H and every prefix of a sequence in H is also in H. $Z \subseteq H$ are the terminal histories (those which are not a prefix of any other sequences). $A(h) = a : (h, a) \in H$ are the actions available after a nonterminal history $h \in H$.
- A function P that assigns to each nonterminal history (each member of $H \setminus Z$) a member of $N \cup c$. P is the player function. $P(h)$ is the player who takes an action after the history h. If $P(h) = c$ then chance determines the action taken after history h.
- A function f_c that associates with every history h for which $P(h) = c$ a probability measure $f_c(\cdot|h)$ on $A(h)$ ($f_c(a|h)$ is the probability that a occurs given h), where all pairs of measures are independent.
- For each player $i \in N$ a partition $\mathcal{I}_i$ of $\{h \in H : P(h) = i\}$ with the property that $A(h) = A(h')$ whenever h and h' are in the same member of the partition. For $I_i \in \mathcal{I}_i$ we denote by $A(I_i)$ the set $A(h)$ and by $P(I_i)$ the player $P(h)$ for any $h \in I_i$. We define $|A_i| = \max_{I_i} |A(I_i)|$ and $|A| = \max_i |A_i|$. $\mathcal{I}_i$ is the information partition of player i; a set $I_i \in \mathcal{I}_i$ is an information set of player i. We denote by $|\mathcal{I}_i|$ the number of information sets belonging to player i in the game and $|\mathcal{I}| = \max_i |\mathcal{I}_i|$.
- For each player $i \in N$ a payoff function u_i from Z to the reals. If $N = 1, 2$ and $u_1 = -u_2$, it is a zero-sum extensive game. Define $\Delta_i = \max_z u_i(z) - \min_z u_i(z)$ to be the range of payoffs to player i.

C. COUNTERFACTUAL REGRET MINIMIZATION (CFR)

Regret matching, described in Appendix A, is for normal-form games. However, it can be efficiently generalized to extensive-form games by using the counterfactual regret minimization (CFR) algorithm. CFR and its extensions are widely used in solving large imperfect-information games.

In CFR [27], $u_i(\sigma, h)$ is defined as the expected utility to player i given that history h has occurred, assuming all players then play according to σ. Next, *counterfactual utility* is defined as the expected utility given that information set I is reached and all players play according to σ except that player i plays to reach I. Formally, if $\pi^\sigma(h)$ is the probability of reaching history h according to σ, and $\pi^\sigma(h, h')$ is the probability of going from history h to history h', then

$$u_i(\sigma, I) = \frac{\sum_{h \in I, h' \in Z} \pi_{-i}^\sigma(h) \pi^\sigma(h, h') u_i(h')}{\pi_{-i}^\sigma(I)}$$

Further, for all $a \in A(I)$, $\sigma|_{I \to a}$ is defined to be a strategy profile identical to σ except that player i always chooses action a when in information set I. *Immediate counterfactual regret* for an action is defined as

$$R_{i,imm}^T(I, a) = \frac{1}{T}\sum_{t=1}^{T} \pi_{-i}^{\sigma^t}(I)(u_i(\sigma^t|_{I \to a}, I) - u_i(\sigma^t, I))$$

and for an information set as

$$R_{i,imm}^T(I) = \max_{a \in A(I)} R_{i,imm}^T(I, a)$$

In CFR, on iteration $T+1$ a player at an information set selects among actions $a \in A(I)$ by

$$p_{T+1}(a) = \begin{cases} \frac{R_i^{T,+}(I,a)}{\sum_{a \in A(I)} R_i^{T,+}(I,a)}, & \text{if } \sum_{a \in A(I)} R_i^{T,+}(I, a) > 0 \\ \frac{1}{|A(I)|}, & \text{otherwise} \end{cases}$$

Using CFR, one can bound regret in extensive-form games.