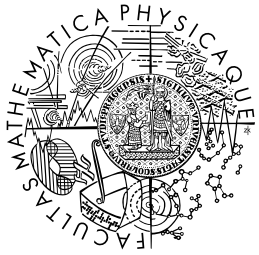

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Bc. Hynek Schlindenbuch

General Game Playing and Deepstack

Department of Software and Computer Science Education

Supervisor of the master thesis: Mgr. Jakub Gemrot, Ph.D.
Study programme: Computer Science
Study branch: Artificial Intelligence

Prague 2019

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ..... signature of the author

I would like to thank my supervisor Mgr. Jakub Gemrot, Ph.D. for his patience and guidance throughout the work on this thesis.

Title: General Game Playing and Deepstack

Author: Bc. Hynek Schlindenbuch

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Jakub Gemrot, Ph.D., Department of Software and Computer Science Education

Abstract: General game playing is an area of artificial intelligence which focuses on creating agents capable of playing many games from some class. The agents receive the rules just before the match and therefore cannot be specialized for each game. Deepstack is the first artificial intelligence to beat professional human players in heads-up no-limit Texas hold'em poker. While it is specialized for poker, at its core is a general algorithm for playing two-player zero-sum games with imperfect information – continual resolving. In this thesis we introduce a general version of continual resolving and compare its performance against Online Outcome Sampling Monte Carlo Counterfactual Regret Minimization in several games.

Keywords: general game playing, imperfect information games, counterfactual regret minimization, continual resolving

Contents

Introduction

Various types of games have long been a test bed for artificial intelligence. Superhuman artificial intelligence was already created for number of games such as Chess [Campbell et al., 2002], Checkers [Schaeffer et al., 1996], and more recently Go [Silver et al., 2016]. These three specifically are all perfect information games (all players know the precise state of the game). Imperfect information adds another level of complexity to a game and therefore makes it more difficult to play. Heads-up no-limit Texas hold'em poker is an imperfect information game comparable in size to Go. The first bot to beat professional human players in it was Deepstack [Moravčík et al., 2017].

Apart from creating artificial agents for specific games, some research has also been focused on creating agents capable of playing general games. This is called general game playing (GGP) [Genesereth and Thielscher, 2014]. In GGP agents are provided with game rules just before each match. Since the rules are not known at the time of creating the agent, it must be able to play well in variety of games in order to be successful.

While Deepstack was built specifically for poker, at its core is a general algorithm for playing two-player zero-sum games with imperfect information – continual resolving. **The goal of this thesis is to evaluate how well does continual resolving perform in the context of GGP.**

In Chapter 1 we introduce the definitions and prior work used throughout the rest of the thesis. In Chapter 2 we present our generalized version of the continual resolving algorithm from Deepstack. In Chapter 3 we describe the details of our implementation – most notably the various hyperparameters of supported algorithms. In Chapter 4 we briefly explain the console interface of the application we created for the evaluation. The design of our experiments and the results are presented in Chapter 5. In Chapter 6 we discuss some of the possible shortcomings of continual resolving's performance. In Chapter 7 we list related work and other approaches to general game-playing with imperfect information.

1. Notation and background

In this chapter we introduce prior work in the area, notation and theoretical background, which we will use throughout the rest of the thesis. The notation is mostly the same as in the original papers. When different papers use conflicting notation, we choose one of them.

1.1 Game Theory

First we need to formalize games and their properties. We rely on the notion of extensive games, which is widely used in this context.

Definition 1 (J. Osborne and Rubinstein [1994], p. 200; Zinkevich et al. [2007])**.** *A finite* ***extensive game*** *consist of the following:*

- *A finite set of* ***players*** N.
- *A finite set of sequences H, such that empty sequence is in H, and all subsequences of a sequence in H are also in H. The sequences are possible* ***histories*** *of actions taken by the players, with $\emptyset$ being the initial state of the game. The set of actions available after history h is denoted $A(h) = \{a|(h, a) \in H\}$, if $A(h) = \emptyset$ then h is* ***terminal*** *history. The set of terminal histories is denoted Z.*
- *A function $P\colon H \setminus Z \to N \cup \{c\}$ that assigns a player to each non-terminal history. If $P(h) = c$ then the action after h is determined by chance.*
- *A function f_c that assigns a probability distribution $f_c(\cdot|h)$ over $A(h)$ for each h where $P(h) = c$. $f_c(a|h)$ is the probability that action a is taken after history h.*
- *For each player $i \in N$ a partition $\mathcal{I}_i$ of $\{h \in H|P(h) = i\}$ so that $\forall I \in \mathcal{I}_i \forall s, t \in I\colon A(s) = A(t)$. For $I \in \mathcal{I}_i$ we denote $A(I) = A(h)$ and $P(I) = i$ for any $h \in I$, and $I_i(h) = I$ if $h \in I$. I is an* ***information set*** *of player i.*
- *For each player $i \in N$ a utility function u_i from Z to real numbers $\mathbb{R}$.*

We also need to define several other properties of extensive games:

- An extensive game has **perfect information** if all information sets for all players contain exactly one history (i.e. players are able to distinguish all histories), otherwise the game has **imperfect information**.
- An extensive game has **perfect recall** if all players retain all of their past knowledge (i.e. each of their information sets contains only histories with the same sequence of the owner's actions and information sets), otherwise the game has **imperfect recall**.
- A two-player extensive game is called **zero-sum** if $\forall z \in Z\colon u_1(z) = -u_2(z)$.

In this thesis we will consider only finite two-player zero-sum perfect recall extensive games with imperfect information.

Player's behavior is described by a strategy. **Strategy** σ_i of player i in an extensive game with imperfect information is a probability distribution over $A(I_i)$ for all $I_i \in \mathcal{I}_i$, and Σ_i is a set of all such strategies. Combined strategy profile of all players is denoted $\sigma = (\sigma_1, \cdots, \sigma_n)$, σ_{-i} denotes strategies of all players but i.

Given a strategy profile σ, **reach probability** of history h is defined as $\pi^\sigma(h) = \prod_{sa \sqsubseteq h} \sigma_{P(s)}(I_{P(s)}(s), a)$ (assuming $\sigma_c = f_c$ and $I_c(s) = \{s\}$). We will also use partial reach probability $\pi_i^\sigma(h)$ containing only probabilities of player i and $\pi_{-i}^\sigma(h)$ which contains probabilities of all players but i.

The **expected utility** of player i is $u_i(\sigma) = \sum_{z \in Z} \pi^\sigma(z) u_i(z)$. The goal of a player is to select their strategy, such that it maximizes their expected utility. However, the strategies of the other players are unknown in the general case. One possibility to overcome this is to instead maximize their worst case expected utility. This idea is formalized by Nash equilibrium.

Definition 2 (Zinkevich et al. [2007])**.** *Nash equilibrium in two-player extensive game is strategy profile $\sigma = (\sigma_1, \sigma_2)$ such that:*

$$u_1(\sigma) \geq \max_{\sigma_1' \in \Sigma_1} u_1(\sigma_1', \sigma_2) \qquad u_2(\sigma) \geq \max_{\sigma_2' \in \Sigma_2} u_2(\sigma_1, \sigma_2')$$

Furthermore σ is ϵ-Nash equilibrium if:

$$u_1(\sigma) + \epsilon \geq \max_{\sigma_1' \in \Sigma_1} u_1(\sigma_1', \sigma_2) \qquad u_2(\sigma) + \epsilon \geq \max_{\sigma_2' \in \Sigma_2} u_2(\sigma_1, \sigma_2')$$

Nash equilibrium is considered to be a solution to the game, since no player can gain by unilaterally changing their strategy.

Strategy σ_i is **best response** to σ_{-i} if $u_i(\sigma_i) \geq \max_{\sigma_i' \in \Sigma_i} u_i(\sigma_i', \sigma_{-i})$ and the set of best response strategies is denoted $BR_i(\sigma_{-i})$. In a zero-sum game, we define **exploitability** as $\epsilon_\sigma = u_1(BR(\sigma_2), \sigma_2) + u_2(\sigma_1, BR(\sigma_1))$[1]. It can be shown that any strategy profile has non-negative exploitability, and only Nash equilibrium strategy profile has zero exploitability. It can therefore be used as a measure of distance from a Nash equilibrium.

1.2 General Game Playing

The goal of general game playing [Genesereth and Thielscher, 2014] is to construct artificial agents capable of playing different games well, without having prior knowledge of them. Rules of the game are provided to the agents right before the match. This results in agents more general than traditional agents designed specifically for one game.

Common way to describe game rules in GGP is the Game Description Language [Love et al., 2008]. GDL is a variant of Datalog, and it allows us to write declarative rules of the game and reason about them. However, original GDL is limited to deterministic games with complete information. This was addressed by

[1] Alternative definition, which divides the sum by 2, also appears in literature, but, as far as we can tell, this version seems to be more prevalent in CFR related literature.

GDL-II [Thielscher, 2010], which is able to describe stochastic games with incomplete information. Both versions are limited to finite, discrete games and assume simultaneous actions of all players, although sequential games can be modeled easily by forcing no-op actions on all but the acting player.

GDL represents game states as sets of facts that hold in them. A game is described by predicates which specify the initial state, legal actions for all players in non-terminal states, state transitions given state and actions of all players, terminal states, and pay-off for all players in the terminal states.

GDL also specifies how matches are run. Each match is handled by a game manager. At the start of the match, the game manager sends to all players their role, the game description in GDL, as well as time limits for computation before the start of the game and for each action. At each turn, the game manager informs all players about the actions taken by the other players in the previous turn, which allows the players to reconstruct the current game state, and asks them to submit their actions for the current turn. If a player fails to submit a legal action to the game manager before the time limit runs out, the game manager selects a random action from the player's legal actions. When a terminal state is reached, the game manager informs the players that the game ended and again includes the actions taken in previous turn.

GDL-II adds a random player, handled by the game manager, which selects one of its legal actions with uniform probability at each turn. This makes it possible to describe stochastic games (rational non-uniform probability distributions can be modeled by duplicating actions). To add imperfect information, the game manager no longer informs the players about the precise actions taken by the other players, but instead sends so called percepts when moving to the next turn. The game rules in GDL-II specify which percepts should be generated for which players for given state and actions. The perfect-recall information sets are determined by a sequence of player's actions and percepts.

1.3 Counterfactual Regret Minimization

Before we can explain how Deepstack works, we have to first introduce counterfactual regret minimization. Counterfactual regret minimization (CFR) [Zinkevich et al., 2007] is an iterative algorithm for approximating Nash equilibria in two-player zero-sum perfect recall extensive games with imperfect information. It extends the concept of regret minimization by introducing counterfactual regret and thus making it more suitable for extensive games.

Regret minimization defines **average overall regret** of player i at time T as:

$$R_i^T = \frac{1}{T} \max_{\sigma_i^* \in \Sigma_i} \sum_{t=1}^{T} (u_i(\sigma_i^*, \sigma_{-i}^t) - u_i(\sigma^t))$$

where σ^t is current strategy profile at time t. The resulting approximate solution after T iterations is the **average strategy**:

$$\overline{\sigma}_i^T(I, a) = \frac{\sum_{t=1}^{T} \pi_i^{\sigma^t}(I)\sigma_i^t(I, a)}{\sum_{t=1}^{T} \pi_i^{\sigma^t}(I)}$$

It is known that if both player's average overall regrets in a zero-sum game are less than ϵ at time T then $\overline{\sigma}^T$ is 2ϵ-Nash equilibrium. Minimizing the regret therefore leads to strategies closer to Nash equilibrium.

Given strategy profile σ **counterfactual values** are defined as:

$$v_i(\sigma, h) = \sum_{z \in Z, h \sqsubseteq z} \pi^{\sigma}_{-i}(h) \pi^{\sigma}(h, z) u_i(z)$$
$$v_i(\sigma, I) = \sum_{h \in I} v_i(\sigma, h)$$

Let $\sigma_{(I \to a)}$ be a strategy profile equal to σ, except that action a is always selected in information set I. CFR then defines **immediate counterfactual regret** for player i at time T and information set I as:

$$R^T_{i,imm}(I) = \frac{1}{T} \max_{a \in A(I)} \sum_{t=1}^{T} (v_i(\sigma^t_{(I \to a)}, I) - v_i(\sigma^t, I))$$

Let $R^{T,+}_{i,imm}(I) = \max(R^T_{i,imm}(I), 0)$, Zinkevich et al. [2007] proves that $R^T_i \leq \sum_{I \in \mathcal{I}_i} R^{T,+}_{i,imm}(I)$ and minimizing $R^T_{i,imm}$ on each information set independently minimizes R^T_i.

CFR works by maintaining cumulative counterfactual regret for all information sets and actions:

$$R^T_i(I, a) = \frac{1}{T} \sum_{t=1}^{T} (v_i(\sigma^t_{(I \to a)}, I) - v_i(\sigma^t, I))$$

Strategy profile σ^{T+1} is selected proportionally to positive cumulative regret at time T. This process is called **regret matching**:

$$\sigma^{T+1}_i(I, a) = \begin{cases} \frac{R^{T,+}_i(I,a)}{\sum_{b \in A(I)} R^{T,+}_i(I,b)} & \text{if } \sum_{b \in A(I)} R^{T,+}_i(I, b) > 0 \\ 0 & \text{otherwise} \end{cases} \tag{1.1}$$

If regret matching is used to select strategies for player i, then $R^T_{i,imm}(I) \leq \Delta_{u,i} \sqrt{|A_i|} / \sqrt{T}$, where $|A_i| = \max_{h \in H:\ P(h) = i} |A(h)|$ and $\Delta_{u,i} = \max_{z \in Z} u_i(z) - \min_{z \in Z} u_i(z)$ [Zinkevich et al., 2007]. It follows that after T iterations of CFR, the exploitability of the average strategy $\overline{\sigma}^T$ is $\mathcal{O}(1/\sqrt{T})$.

Vanilla version of CFR with alternating updates is presented in Algorithm 1. It can be modified in many ways – such as using different regret matching schemes, chance sampling, discounting cumulative strategy and regret from earlier iterations, etc. We will now introduce some modified versions.

1.3.1 CFR$^+$

CFR$^+$ [Tammelin et al., 2015] is an improved version of CFR which was used to solve heads-up limit Texas hold'em poker. It doesn't use any sampling and uses alternating updates as is the case in Algorithm 1. It uses weighted average to compute the average strategy $\overline{\sigma}^T = 2/(T^2 + T) \sum_{t=1}^{T} t\sigma^t$. Finally, it uses regret matching$^+$, which tracks only positive regret – $Q^t(a) = (Q^{t-1}(a) + \Delta R^t(a))^+$ and selects the next strategy σ^{t+1} proportionally to Q^t (as in Equation 1.1).

Algorithm 1 Vanilla CFR with alternating updates [Lanctot, 2013]

Initialize cumulative regret tables $\forall I, a\colon r_I[a] \leftarrow 0$
Initialize cumulative strategy tables $\forall I, a\colon s_I[a] \leftarrow 0$
Initialize strategy profile $\forall I, a\colon \sigma^1(I, a) \leftarrow 1/|A(I)|$

function CFR(h, i, t, π_1, π_2)
    if h is terminal **then**
        return $u_i(h)$
    else if h is chance node **then**
        return $\sum_{a \in A(h)} \sigma_c(h, a)$ CFR(ha, i, t, $\sigma_c(h, a) \cdot \pi_1$, $\sigma_c(h, a) \cdot \pi_2$)
    end if
    Let I be the information set containing h.
    $u_\sigma \leftarrow 0$
    $u_{\sigma_{I \to a}}[a] \leftarrow 0$ for all $a \in A(I)$
    for $a \in A(I)$ **do**
        if $P(h) = 1$ **then**
            $u_{\sigma_{I \to a}}[a] \leftarrow$ CFR(ha, i, t, $\sigma^t(I, a) \cdot \pi_1$, π_2)
        else
            $u_{\sigma_{I \to a}}[a] \leftarrow$ CFR(ha, i, t, π_1, $\sigma^t(I, a) \cdot \pi_2$)
        end if
        $u_\sigma \leftarrow u_\sigma + \sigma^t(I, a) \cdot u_{\sigma_{I \to a}}[a]$
    end for
    if $P(h) = i$ **then** ▷ Alternating update
        for $a \in A(I)$ **do**
            $r_I[a] \leftarrow r_I[a] + \pi_{-i} \cdot (u_{\sigma_{I \to a}}[a] - u_\sigma)$
            $s_I[a] \leftarrow s_I[a] + \pi_i \cdot \sigma^t(I, a)$
        end for
        $\sigma^{t+1}(I) \leftarrow$ RegretMatching(r_I) as defined in Equation 1.1
    end if
    return u_σ
end function

function SOLVE
    for $t = 1, \ldots, T$ **do**
        for $i \in \{1, 2\}$ **do**
            CFR($\emptyset$, i, t, 1, 1)
        end for
    end for
end function

1.3.2 Linear and Discounted CFR

While CFR$^+$ uses cumulative strategy discounting (later iterations have larger weight), Brown and Sandholm [2018] proposed **Linear CFR** (LCFR) which discounts regrets as well. This allows the solver to faster recover from mistakes done in early iterations. LCFR is CFR with linear weights for regrets and cumulative strategy. The authors also explore the possibility of LCFR$^+$, but conclude that it performs worse in practice.

To bridge the gap between LCFR and CFR$^+$, they introduce **Discounted CFR** (DCFR). DCFR has three hyperparameters – α, β and γ. At iteration t of DCFR$_{\alpha,\,\beta,\,\gamma}$ accumulated positive regrets are multiplied by $\frac{t^\alpha}{t^\alpha+1}$, negative regrets by $\frac{t^\beta}{t^\beta+1}$, and accumulated strategy by $(\frac{t}{t+1})^\gamma$. CFR, CFR$^+$ and LCFR can all be emulated by DCFR$_{\infty,\infty,0}$, DCFR$_{\infty,-\infty,1}$, and DCFR$_{1,1,1}$ respectively.

The authors also claim that CFR$^+$ performs better with t^2 weights instead of t for the average strategy, and that DCFR$_{1.5,0,2}$ performs consistently better than this modified CFR$^+$.

1.3.3 Monte-Carlo CFR

All the CFR variants we introduced so far need to traverse the whole game tree in each iteration. This can lead to poor early performance, and it also means that the solution is improved in long steps. Lanctot et al. [2009] presented Monte-Carlo CFR (MC-CFR), which instead divides the terminal histories into blocks and samples one block in each iteration. Iterations are therefore shorter and progress towards the solution more gradual.

To define MC-CFR more formally – let $\mathcal{Q} = \{Q_1, \ldots, Q_r\}$ be set of subsets of Z, such that $\bigcup_{i=1}^{r} Q_i = Z$. Each Q_i is called a block, and the probability of sampling Q_i is $q_i > 0$.

Let $q(z) = \sum_{j:\, z \in Q_j} q_j$ be the probability of sampling $z \in Z$, Z_I be the subset of Z with predecessors in I, and $z[I]$ be the predecessor of z in I. **Sampled counterfactual value** of I given block Q_j is:

$$\tilde{v}_i(\sigma, I|j) = \sum_{z \in Q_j \cap Z_I} \frac{1}{q(z)} u_i(z) \pi^\sigma_{-i}(z[I]) \pi^\sigma(z[I], z)$$

The algorithm then works by sampling a single block at each iteration and using sampled counterfactual values for regret updates. Information sets not intersecting the sampled block do not have to be updated, since their sampled counterfactual values are 0 by definition.

There are many ways to choose $\mathcal{Q}$ and sampling probabilities. Lanctot et al. [2009] proposed outcome sampling, which samples one terminal history per iteration, and external sampling, which samples only actions external to the player (ie. opponent's and chance actions) in each iteration. There is also average strategy sampling [Gibson et al., 2012], which samples external actions the same way as external sampling, but also samples a subset of player's actions according to modified player's average strategy.

In this thesis we will use outcome sampling. As we already said, in outcome sampling each block Q_j contains a single terminal history. The sampling

probability q_j is defined as a reach probability given sampling strategy profile $q_j(z) = \pi^{\xi}(z)$. ξ is usually chosen as:

$$\xi(h,a) = \begin{cases} \epsilon \cdot \frac{1}{|A(h)|} + (1-\epsilon) \cdot \sigma^t(I(h),a) & \text{if } P(h) = i \\ f_c(h,a) & \text{if } P(h) = c \\ \sigma^t(I(h),a) & \text{otherwise} \end{cases} \tag{1.2}$$

where t is the current iteration, i is the player updated at this iteration, and $\epsilon \in (0,1]$ is an exploration factor. This can lead to sampling probabilities of some blocks being zero. However, for those blocks the counterfactual reach probability $\pi_{-i}^{\sigma^t}(z)$ is zero and therefore sampling them wouldn't have any impact on the regrets. Lanctot et al. [2009] proved a probabilistic regret bound applicable to this choice of sampling probabilities.

Online version of MC-CFR with outcome sampling is described in Algorithm 2.

1.3.4 Variance Reduction in MC-CFR

One of the problems with MC-CFR, especially with outcome sampling, is that sampled counterfactual values can have high variance. Schmid et al. [2018] proposed using baselines to decrease the variance, and shows this leading to significantly better long-term convergence.

Let $b_i(I,a)$ be a baseline value of taking action a in information set I for player i, $\widehat{b}_i(I,a)$ be a baseline estimate, such that $\mathbb{E}[\widehat{b}_i(I,a)] = b_i(I,a)$, and $\widehat{v}_i(\sigma,I,a)$ be an estimated counterfactual value of doing the same (i.e. $\widetilde{v}_i(\sigma,I,a)$ from MC-CFR). The idea is to use baseline-enhanced estimate of counterfactual value $\widehat{v}_i^b(\sigma,I,a) = \widehat{v}_i(\sigma,I,a) - \widehat{b}_i(I,a) + b_i(I,a)$ instead of $\widehat{v}_i(\sigma,I,a)$. With proper choice of baseline, this will result in reduction of the estimated counterfactual value's variance.

VR-MCCFR also uses baseline values to estimate counterfactual values of actions that were not sampled in given iteration. For outcome sampling[2] this leads to the following definitions:

$$\widehat{b}_i(I,a|z) = \begin{cases} b_i(I,a)/\xi(h,a) & \text{if } ha \sqsubseteq z, h \in I \\ 0 & \text{otherwise} \end{cases}$$

$$\widehat{u}_i^b(\sigma,h,a|z) = \begin{cases} b_i(I(h),a) + \frac{\widehat{u}_i^b(\sigma,ha|z) - b_i(I(h),a)}{\xi(h,a)} & \text{if } ha \sqsubseteq z \\ b_i(I(h),a) & \text{if } h \sqsubset z, ha \not\sqsubseteq z \\ 0 & \text{otherwise} \end{cases}$$

$$\widehat{u}_i^b(\sigma,h|z) = \begin{cases} u_i(h) & \text{if } h = z \\ \sum_{a \in A(h)} \sigma(I(h),a) \widehat{u}_i^b(\sigma,h,a|z) & \text{if } h \sqsubset z \\ 0 & \text{otherwise} \end{cases}$$

$$\widehat{v}_i^b(\sigma,h,a|z) = \frac{\pi_{-i}^{\sigma}(h)}{q(h)} \widehat{u}_i^b(\sigma,h,a|z)$$

It is important to note that, unlike in the definition of $\widetilde{v}_i(\sigma,h,a|z)$, the denominator in the definition of the baseline-enhanced sampled counterfactual value is

[2] VR-MCCFR can also be adapted to other sampling techniques.

$q(h)$ instead of $q(z)$. This is because the tail part of the sampling probability is already included in $\widehat{u}_i^b(\sigma, h, a|z)$.

Baseline value $b_i(I, a)$ should be chosen such that it approximates or correlates with $\mathbb{E}[\widehat{u}_i(\sigma, I, a)]$. Schmid et al. [2018] recommends using exponentially decaying average of $\widehat{u}_i^b(\sigma^t, ha|z)$ with decay rate 0.5.

1.4 Online game playing

Counterfactual regret minimization is intended to approximate a Nash equilibrium strategy for the whole game in advance (offline). However this is not suitable for online game playing (such as in GGP). In online setting an agent is given a time limit to select an action at each of its decision points encountered during a match. Additionally, there is usually some amount of pre-play time available to the agent for initialization.

1.4.1 Online MC-CFR

A very simple approach to online game playing is to solve the game from scratch using one of the offline techniques while playing.[3] MC-CFR is especially useful for this, because of it's shorter iterations and better early convergence. Lisý et al. [2015] introduced online outcome sampling MC-CFR (OOS MC-CFR), which uses incremental game tree building (to avoid spending time up-front to initialize regret-tables, etc.) and allows in-match targeting to concentrate more iterations to the part of the game tree it is currently in.

Incremental game tree building starts with only the root information set in memory. When a new information set is encountered during outcome sampling, it is evaluated using a playout policy (e.g. uniform random playout) and added to memory. Information sets visited during the playout phase are not added to memory.

If targeting is used, a scenario is decided prior to each iteration – with targeting probability δ outcome sampling is limited to the targeted part of the game tree, and with probability $(1 - \delta)$ regular outcome sampling is used. The algorithm maintains both targeted (s_1) and untargeted (s_2) sampling probability for current history – the corresponding overall sampling probability is then $q(h) = \delta s_1 + (1 - \delta)s_2$.

Lisý et al. [2015] presented two targeting schemes – information set targeting (IST) and public subgame targeting (PST). Given the player's current information set I, information set targeting targets $z \in Z$ such that $\exists h \in I : h \sqsubseteq z$. This targeting scheme is possible in any game, however it can have convergence issues.

Public subgame targeting targets terminals consistent with the current public subgame. Public subgame is a set of states with the same sequence of public actions, which are actions observable by all players (e.g. bets in poker). However, some games have no public actions (e.g. incomplete-information Goofspiel) and therefore do not support public subgame targeting.

As targeting changes during a match, the sampling probability of targeted part of the game increases leading to lower weight in cumulative regret and strategy.

[3] We refer to this general approach as online solving throughout the thesis.

This make it difficult to correct mistakes done in early iterations, which had much higher weight. To overcome this issue, OOS MC-CFR increases the weight of iterations after changing the targeting. It also uses explorative regret matching, which combines the regret-matched strategy with uniform strategy with small weight $\gamma = 0.01$, to ensure strategy is improved even in information sets with $\pi_{-i} = 0$.

Pseudo-code for the resulting algorithm is presented in Algorithm 2.

1.4.2 Deepstack

Deepstack [Moravčík et al., 2017] for heads-up no-limit Texas hold'em poker consists of three parts – continual resolving, depth limited lookahead and sparse lookahead trees.

Continual resolving is a general algorithm for playing two-player zero-sum games with imperfect information and perfect recall. Each time it has to act it determines the current subgame, solves it using CFR, plays according to the obtained strategy, and discards it afterwards. More specifically, it uses CFR-D gadget to construct and resolve the subgame.

CFR-D [Burch et al., 2014] is an algorithm for offline solving of large games using decomposition to subgames. However, only the CFR-D gadget – the mock game solved at each step of continual resolving – is important for us. To define a sugbame, CFR-D first extends the definition of information set to augmented information sets. Augmented information set for player i contains histories which have the same sequence of player i's information sets and actions. Unlike information sets, the augmented information sets are also defined for histories where player i is not the acting player, while being equivalent to regular information sets in histories where player i is acting (assuming the game has perfect recall). Given history s in a subgame, history t is in the same subgame if $s \sqsubseteq t$ or $s, t \in I_p$ for augmented information set of any player p.

Now we know which histories to include in a subgame, but to run CFR we need a single initial history. This is where the CFR-D gadget comes in. Let us assume we are trying to resolve a strategy for player 1 in subgame S. Let R be the set of histories at the root of S. Let $CBR_i(\sigma_{-i})$ be a counterfactual best response to σ_{-i} – that is a best response such that $\sigma_i(I, a) > 0$ if and only if $v_i(I, a) \geq \max_{b \in A(I)} v_i(I, b)$. To construct the CFR-D gadget for this subgame we also need $v_2^R(I) = v_2^{\langle \sigma_1, CBR_2(\sigma_1) \rangle}(I)$ for all augmented information sets $I \in \mathcal{I}_2^R$, and $\pi_{-2}^{\sigma}(r)$ for all $r \in R$.

The gadget begins with initial chance node, which leads to opponent's choice node $\widetilde{r}$ with probability $\pi_{-2}^{\sigma}(r)/k$ for all $r \in R$, where $k = \sum_{r \in R} \pi_{-2}^{\sigma}(r)$. The probabilities are normalized by k to ensure that they sum to 1. Payoff function in terminal histories is multiplied by k to undo the normalization. The opponent has two available actions in $\widetilde{r}$ – follow (F) and terminate (T). $\widetilde{r} \cdot T$ is a terminal state with payoff $\widetilde{u}_2(\widetilde{r} \cdot T) = k v_2^R(I(r)) / \sum_{h \in I(r)} \pi_{-2}^{\sigma}(h)$. This means that $\widetilde{u}_2(I \cdot T) = v_2^R(I)$ for all $I \in \mathcal{I}_2^R$. Follow action leads to r from the original game.

Burch et al. [2014] showed that combining the original trunk strategy with resolved subgame strategy results in a strategy with bounded exploitability.

The next component of Deepstack is depth limited lookahead. Since we only need to compute the strategy at the root of a subgame, it is possible to replace

CFR beyond certain depth with a heuristic evaluation function, which returns estimated counterfactual values for both players. In Deepstack this function is approximated using a deep neural network. Moravčík et al. [2017] proves that if the error of the evalution function is less than ϵ and T iterations of depth-limited CFR are used for the re-solve, then the resulting strategy has exploitability less than $k_1\epsilon + k_2/\sqrt{T}$ for game-specific constants k_1, k_2.

Finally, the sparse lookahead tree means, that Deepstack only considers a subset of all available actions at each information set. More specifically for Texas hold'em poker it considers only fold, call, 2 or 3 bet actions, and all-in. This significantly reduces the size of re-solved trees, however it also means that the exploitability bound from previous paragraph does not hold anymore.

Algorithm 2 Online Outcome Sampling MC-CFR [Lisý et al., 2015]

Returns (tail reach probability, root-to-leaf sampling probability, payoff)
function OOS(h, π_i, π_{-i}, s_1, s_2, i)
  if h is terminal **then**
    return $(1,\ \delta s_1 + (1-\delta)s_2,\ u_i(h))$
  else if h is chance node **then**
    $(a,\ \rho_1,\ \rho_2) \leftarrow$ SAMPLE(h, i)
    $(x,\ l,\ u) \leftarrow$ OOS(ha, π_i, $\rho_2\pi_{-i}$, $\rho_1 s_1$, $\rho_2 s_2$, i)
    return $(\rho_2 x,\ l,\ u)$
  end if
  Let I be the information set containing h.
  $(a,\ \rho_1,\ \rho_2) \leftarrow$ SAMPLE(h, i)
  if I is not in memory **then**
    Add I to memory
    $\sigma(I) \leftarrow$ UNIF(A(I))
    $(x, l, u) \leftarrow$ PLAYOUT(ha, $\frac{\delta s_1 + (1-\delta)s_2}{A(I)}$)
  else
    $\sigma(I) \leftarrow$ REGRETMATCHING(r_I)
    $\pi'_{P(h)} \leftarrow \sigma(I,a)\pi_{P(h)}$
    $\pi'_{-P(h)} \leftarrow \pi_{-P(h)}$
    $(x,\ l,\ u) \leftarrow$ OOS(ha, π'_i, π'_{-i}, $\rho_1 s_1$, $\rho_2 s_2$, i)
  end if
  $x' \leftarrow \sigma(I,a)x$
  $\widetilde{u}_\sigma \leftarrow ux'/l$
  for $a' \in A(I)$ **do**
    if $P(h) = i$ **then**
      if $a' = a$ **then**
        $\widetilde{u}_{\sigma_{I\to a'}} \leftarrow ux/l$
      else
        $\widetilde{u}_{\sigma_{I\to a'}} \leftarrow 0$
      end if
      $r_I[a'] \leftarrow r_I[a'] + \pi_{-i} \cdot (\widetilde{u}_{\sigma_{I\to a'}} - \widetilde{u}_\sigma)$
    else
      $s_I[a'] \leftarrow s_I[a'] + \frac{\pi_{-i}\sigma(I,a')}{\delta s_1 + (1-\delta)s_2}$
    end if
  end for
  return (x', l, u)
end function

2. General Continual Resolving

In this chapter we describe our generalized version of continual resolving. To adapt continual resolving to general games we need a CFR solver and a way to construct the proper CFR-D subgame gadget each time it is our turn to act. In the previous chapter we have described several variants of CFR, which we can use. MC-CFR with outcome sampling should be especially suitable for general game playing. It has much shorter iterations compared to original CFR, which makes it easier to fulfill the time constraints for selecting an action. And unlike depth-limited CFR, which can also have short enough iterations, it does not require a game-specific evaluation function.

In order to construct a CFR-D gadget for our turn, we need to know which histories are at the root of the smallest subgame containing our information set, opponent's counterfactual values $v_2^R(I)$ for corresponding augmented information sets, and $\pi^\sigma_{-2}(h)$ for all root histories. We precompute this information for our next turn during the current resolving. The first resolving is done in the original game and thus does not need a gadget to be constructed.

Given history s in the subgame, history t is also in the subgame if $s \sqsubseteq t$, or s and t are both members of the same augmented information set I_p for some player p. Therefore, a history s is at the root of a subgame if it is either the root of the whole game, or if $s = ha$ and $\{P(h), P(ha)\} = \{1, 2\}$. If either $P(h) = c$ or $P(ha) = c$ then h and ha are in the same augmented information set for at least one of the players. This also means, that there are games, which only have one subgame – the whole game, and that the closest subgame may be the same for multiple successive turns.

Now that we know how to recognize subgame roots, we need to map our information set to a set of root histories of the closest subgame. We do this by traversing the current subgame gadget until we find our next turn in a different subgame. This gives us a set of histories $\{(r_i, s_i)|i = 1, \ldots, n\}$ such that s_i is our next turn and r_i is the subgame root closest to s_i. We can then construct a map from $\{I_1(s_i), I_2(s_i)|i = 1, \ldots, n\}$ to sets of corresponding root histories. And finally merge all subgame roots with non-empty intersection. In the next turn we can simply obtain the subgame roots from the map. If our current information set is not in the subgame map, it means we are still in the same subgame as in the previous turn.

The opponent counterfactual values and our reach probabilities are approximately estimated during the resolving of the current subgame. We use self-play values instead of best-response values, and an arithmetic mean of values from each iteration instead of values for the average strategy (same as Deepstack [Moravčík et al., 2017]). The counterfactual values are estimated as:

$$v_2^R(I) = \frac{1}{T}\sum_{t=1}^{T}\sum_{h \in I} v_2^{\sigma^t}(h)$$

Similarly for reach probabilities:

$$\pi^{\overline{\sigma}^T}_{-2}(h) = \frac{1}{T}\sum_{t=1}^{T}\pi^{\sigma^t}_{-2}(h)$$

3. CFRPlayground

CFRPlayground is the application we built to evaluate performance of general continual resolving. The application is written in Java 8, it uses the Gradle build system [gra] and picocli [pic] for its console interface.

3.1 GGP framework

To implement general versions of CFR and continual resolving, we first need a framework for general games. This framework is designed for the specific requirements of general continual resolving. It supports two-player stochastic games with incomplete information. Even though continual resolving further requires zero-sum games with perfect-recall, these properties are not enforced by the framework for simplicity.

A game tree consists of states[1] and it is traversed by taking actions legal in those states. Taking an action in a state can result in some information being revealed to one or both players. This is done by the state generating percepts when an action is taken. Each action can result in several percepts. Each percept is visible to only one player, public percepts are implemented by duplicating the percept for both players. Player i's perfect recall information set contains states with the same sequence of player i's actions and percepts on the paths from the root of the game to the states. Imperfect recall information sets are also allowed by the framework, in which case multiple sequences are part of a single information set (e.g. it does not matter in which order a player marks fields in latent tic-tac-toe).

`ICompleteInformationState` and `IInformationSet` are the two main interfaces which represent this structure. `ICompleteInformationState` provides information about the game state it represents – namely whether the state is terminal, the acting player, the legal actions, the state transitions and percepts for each legal action, the action distribution if the state is random, the payoffs if the state is terminal, and information sets for both players.

`IInformationSet` generally represents a sequence of owning player's actions and percepts (or even multiple such sequences in games without perfect recall). It provides the legal actions when the owner is acting, as well as transition information for legal actions and valid percepts. When the owning player is acting it corresponds to the information set as defined in extensive games. Otherwise, it only serves as an intermediary for tracking past actions and percepts until the owner's next turn.

Actions and percepts are represented by `IAction` and `IPercept` interfaces respectively. Finally the whole game is represented by `IGameDescription` which provides the initial state and initial information sets for the game.

Classes implementing these interfaces should be immutable and serializable, and should correctly implement `hashCode` and `equals` methods.

This design is very similar to the game model used by GDL-II. That is because we originally intended to support GDL-II, however as our focus shifted more to

[1] Histories correspond to paths from the initial state, whereas states are the nodes of the game tree. Both terms are used interchangeably.

continual resolving, it became clear that it would not serve a practical purpose in terms of evaluating continual resolving on general games. We therefore built this simpler framework, which allowed us to avoid the overhead of GDL-II during evaluation. Its similarity to GDL-II's model should make it relatively easy to embed GDL-II into it.

Compared to GDL-II it is limited to two-player games, however unlike GDL-II it allows arbitrary distribution of actions in random states (GDL-II only supports uniform distribution). It is also specifically designed for CFR and continual resolving, and may not be suitable for other types of players. Most notably it would not be suitable for players using neural networks to learn about a game, as it does not expose any game features that could be used as inputs for a neural network. This could of course be added, but we chose to avoid it, as the context of general game playing makes it unfeasible to train a neural network during game-play anyway.

3.2 Games

The games we implemented are all zero-sum, mostly with imperfect recall (because of more straight-forward implementation). Perfect recall support can be added to a game by wrapping it in `PerfectRecallGameDescriptionWrapper`, which forces information sets to correspond to sequences of actions and percepts as defined in the previous section.

3.2.1 Leduc Poker

Leduc poker is a simplified two-player poker game. Players start with M_1 and M_2 chips respectively. At the beginning of each game they both place 1 chip in the pot. Each player is given one private card from a deck of two suites of cards, each containing C cards. Then the first (of two) betting round begins with the first player. Bet size for this round is two chips. In a betting round a player can fold at any time[2], which results in immediate win by the other player. A player can also call, which matches the opponent's last bet (or zero if opponent did not bet any additional chips). Finally, a player can raise, which matches opponent's last bet and also places additional chips into the pot for the opponent to match.

When the first betting round finishes, a public card is drawn from the deck and revealed to both players. Then the second betting round is opened with the first player. Bet sizes for this round are four chips. If a player does not have enough chips to fully match the opponent's bet and chooses to match it (either by call or raise), all of his remaining chips are transferred to the pot and any extra chips returned to the opponent (so that both players have the same amount of chips in the pot). The maximum number of raises in a betting round is B.

When the second betting round ends, the player with the same card as the public one wins. If no player has the same card, then the one with the higher card wins. Otherwise, it is a draw and both players get their chips back. The winner gets all the chips from the pot (payoff is chips won - chips bet). The game is implemented with imperfect recall and it is parameterized by (M_1, M_2, B, C).

[2]Other common version is to allow folds only when the player must place additional chips to the pot to continue playing.

3.2.2 II-Goofspiel

Incomplete information Goofspiel is also a card game. Each player has a deck of cards $1 \dots N$, and there is a sorted public deck of cards $1 \dots N$ (with 1 on top). At each turn players privately bid one of the cards from their private decks. The higher bidder wins the top card from the public deck, which is then removed from the public deck and both private cards are discarded. If both players bid the same card, nobody wins the public card (it is still removed). Both players are informed if the turn was a win, a loss or a draw, but not which card did the opponent bid. The winner is the player with the higher sum of won public cards. If the sum is the same for both players, the game results in a draw. The payoff is +1/-1 for win/loss or 0 for draw. The game is implemented with perfect recall and it is parametrized by N.

3.2.3 Princess and Monster

Princess and monster (PAM) is a pursuit-evasion game. Our implementation uses 3×3 grid with horizontal and vertical connections (no diagonal connections) as the playing field. The princess starts at position $[0, 0]$, the monster starts at position $[2, 2]$. At each turn the princess moves first, followed by the monster. Each move must be from the currently occupied field to a connected field (staying on the same field is not allowed). There is a maximum of T turns in the game. If the monster and the princess occupy the same field at any time, the monster wins. If that does not happen in any of the T turns, the princess wins. The payoff is $2T$ for the princess if she wins, or $(2T - princessMoves - monsterMoves)$ for the monster if it wins. The players do not receive any percepts. The game is parameterized by T and it is implemented with imperfect recall.

3.2.4 Latent Tic-Tac-Toe

Latent tic-tac-toe is similar to a perfect information tic-tac-toe. Our version is played on 3×3 grid. Players take turns in marking fields on the board. The first player to get 3 marks in a row, a column or on a diagonal wins. If neither player manages to do that and there are no more free fields left, or the first player manages to do it first and the second player also does it the very next turn, the game ends in a draw. Unlike in the classical tic-tac-toe, player moves are delayed until after the opponent makes the next move. After that the move is publicly revealed and it must not conflict with previously revealed moves. If the move conflicts it is simply discarded. The payoff is $+1/-1$ for win/loss or 0 for draw. The game is implemented without perfect recall and it has no parameters.

3.2.5 Rock-Paper-Scissors

Rock-paper-scissors is a very simple game, that we include only for debugging purposes. In our version each player chooses a number from the interval $[1, N]$, where N is an odd number. If both players choose the same number, the game ends in a draw. Otherwise, the player with the higher number wins if the difference between the higher number and the lower number is greater than $(N-1)/2$. The payoff is $+1/-1$ for win/loss or 0 for draw.

3.3 Solvers

The two solvers implemented in our application are based on CFR and MC-CFR. They can be further customized by variety of different hyperparameters. In general both solvers maintain a map of information set data used during solving. This includes the current strategy, the cumulative strategy and the regret matching. MC-CFR additionally stores baseline values for both players.

Both solvers allow a caller to register event listeners which are called each time a state is entered and left. Both callbacks include information about probabilities of reaching the state (separately for both players and chance), the leave callback also includes the utility of the state under current strategy (which we will need to implement continual resolving). To allow a caller to gather additional information about the states, the solver does not work directly with the states, but instead uses a `IGameTraversalTracker` provided by the caller, which the solver uses to traverse the game tree.

3.3.1 Regret Matching

General regret matching configuration is represented by a factory which implements `IRegretMatching.IFactory`. It is then used to create `IRegretMatching` objects for each information set encountered in the game. `IRegretMatching` accumulates regret for the information set's actions, and is used to construct a regret-matched strategy. We provide the following regret matching algorithms:

- Regret matching,
- regret matching$^+$,
- discounted regret matching – parameterized by discounting exponents α, β for positive and negative regrets respectively (as described in 1.3.2),
- explorative regret matching – parameterized by the exploration factor γ and the underlying regret matching algorithm (one of the above).

3.3.2 CFR

Our CFR implementation is close to the version described in Algorithm 1. It additionally supports both alternating and simultaneous updates, cumulative strategy discounting (as described in 1.3.2), and depth-limited CFR. When depth-limited CFR is used, the solver must be provided with a depth limit and an utility estimator factory (`IUtilityEstimator.IFactory`). The purpose of the utility estimator is to estimate the utility of a given state under a Nash equilibrium strategy. The estimator can also limit which states it is able to estimate, in case a fixed depth limit is not sufficient for given use-case (such as in continual resolving).

Since the GGP context makes it difficult to come up with a meaningful utility estimator in time, we only include `RandomPlayoutUtilityEstimator` which estimates the utility of a state by sampling N terminal states from the given state and returning the expected utility of those states under uniform random strategy.

3.3.3 MC-CFR

Our MC-CFR implementation is based on Algorithm 2. It uses outcome sampling, incremental game tree building and a uniform random playout to evaluate information sets visited for the first time. However, unlike OOS MC-CFR it does not assign higher weight to future iterations when the in-match targeting is changed. It also supports the same cumulative strategy scheme as our CFR implementation. Additionally, it uses baselines to estimate utility of non-sampled actions (as described in 1.3.4). The two implemented baselines are exponential decaying average and no baseline (which returns zero for all actions and is thus equivalent to MC-CFR without variance reduction).

Another improvement is CFR-D-aware sampling. Which is a modification of the underlying sampling scheme (in this case outcome sampling, but it could also be adapted for other sampling schemes). When CFR-D-aware sampling encounters an opponent choice state from CFR-D, it samples both the terminate and the follow action. We do this because sampling an action from the initial chance node can be relatively expensive compared to regular states because of its size. We therefore do not want to "waste" an iteration on simply sampling a terminal history right below it.

The full list of supported hyperparameters is as follows:

- regret matching,
- cumulative strategy discounting exponent γ (as described in 1.3.2),
- exploration probability $e \in (0, 1]$,
- targeting probability $t \in [0, 1]$,
- in-match targeting scheme (we only provide information set targeting),
- baseline.

3.4 Players

A general player is represented by `IPlayerFactory`, which is responsible for creating `IPlayer` objects for specific game and role. At the beginning of a match both players have some time to initialize. During the game the players are responsible for maintaining their own respective information sets, so that they can infer the legal actions when it is their turn to act. They do this by updating their information set whenever they receive a percept or take an action. Players are given only a limited amount of time to select an action each time they act, however this limit is not currently enforced.

3.4.1 Continual Resolving Player

Continual resolving player is our implementation of general continual resolving described in Chapter 2. The player is parameterized by an object implementing `ISubgameResolver.IFactory`. This factory is used to create a resolver for each visited subgame. Our resolver factory produces resolvers, which use one of the

solvers from the previous section to resolve a given subgame. A resolver is provided with information necessary to construct a CFR-D subgame gadget. Once it constructs the gadget, it first initializes information necessary to construct all the possible next subgame gadgets. This includes the subgame map (which maps information sets to corresponding subgame root states), the reach probabilities for each subgame root state, and opponent's counterfactual values for each opponent's augmented information set at the root of a subgame. Then it runs the solver until the time limit for action selection runs out and returns the cumulative strategy as well the aggregated information necessary to construct a subgame gadget for any of the next possible subgames. If the player acts multiple times in one subgame, it uses the same resolver and simply improves the strategy for the subgame.

If the underlying supports targeting (i.e. it is a MC-CFR solver with $t > 0$), the resolver will provide it with information set targeting.

3.4.2 Solving Player

Solving player is parametrized by one of the solvers described in the previous section. During init and each action selection, the player simply runs the solver in the whole game until the time limit runs out and improves the cumulative strategy. Then it selects an action according to this strategy. If the solver supports targeting, the player will provide it with information set targeting for each action selection. We did not implement public subgame targeting, because not all games have public subgames (in our case goofspiel, and princess and monster), and because, unlike information set targeting, it is of no use to continual resolving.

3.5 Player evaluators

The purpose of a player evaluator is to estimate the strategy played by a given player in a given game. For offline solvers this is trivial, since they give us strategy for the whole game directly. A player, on the other hand, modifies his strategy during each action selection and may not even store a strategy for the whole game. We therefore have to aggregate it. This can be done precisely by traversing the game tree, computing a strategy in each of the player's information sets, and aggregating them together to obtain a strategy for the whole game. This approach is implemented in `TraversingEvaluator`.

However since this approach is unfeasible for larger games. We therefore also include the aggregate method described by Lisý et al. [2015]. It works by playing many matches against a random opponent and aggregating strategies for information sets which are likely to be similar to the ones that would be computed by the exact method. For the solving player, unless information set targeting is used, we include all the player's information set with the same number of his previous actions, since they would have the same amount of solving time to compute a strategy. If information set targeting is used then we include only the targeted information and the player's information sets directly under it . For continual resolving we use a similar approach, but limited to the current subgame. This is implemented in `GamePlayingEvaluator`.

3.6 Configuration

Because our application has a console interface, we need a way to assemble the objects described above from a textual description. Since many of those objects also have complex dependencies we cannot just use simple command-line options for each hyperparameter. The configuration should be short and easily readable. While there are different textual formats for configuration – such as XML or JSON – we were unable to find one that would be quite right for our use-case. We therefore designed our own from scratch.

We used ANTLR 4 parser generator [ant] to write a grammar and generate a parser for it. The grammar supports integers, floats, booleans, strings, objects and arrays. An object has a name and any number of positional and key-value parameters. We use the generated parser to construct an abstract syntax tree.

```
ContinualResolving{VR-MCCFR{rm=RM,e=0.7,bl=ExpAvg{0.3}}}
```

Figure 3.1: Example of a configuration string with nested complex objects, positional and key-value parameters.

The abstract syntax tree is then passed to a configurable factory along with a requested type of the object to be configured. The factory directly supports primitive types. To support additional types, it must be provided with an identifier, a list of possible parameters and a function to assemble the object given the parameters (which are first processed by the factory). The factory then traverses the abstract syntax tree and tries to assemble the object.

4. User manual

In this chapter we briefly describe commands available in the console interface of our application. More details, including the precise command-line options, can be found by using the built-in help command. More details can also be found in Appendix A.4.

The application includes three evaluation commands – `solve`, `evaluate` and `cfrd-eval`. All of these commands include a warm-up phase, save evaluation results to a file, and support quiet and dry-run modes. In the quiet mode a command does not write text to the standard output. In the dry-run mode the evaluation results are not saved to a file.

The `solve` command is used to evaluate performance of an offline CFR solver on a given game by running the solver for specified amount of time and periodically evaluating the exploitability of the accumulated strategy. The final strategy can also be saved to a file.

The `evaluate` command is used to evaluate performance of a player on a given game using a given player evaluator (traversing or game-playing evaluator). Traversing evaluator saves the evaluation results directly to a CSV file. Game-playing evaluator instead serializes the results (including the aggregated strategy). Results from multiple game-playing evaluations with matching settings can be merged together (using the `merge-gp-results` command), and converted to a CSV file (using the `gp-to-csv` command).

The `cfrd-eval` command is used to evaluate performance of a combined trunk strategy and a strategy resolved from a CFR-D subgame gadget. A trunk strategy can be either loaded from a file or computed by a given solver in a given time-limit. Once a trunk strategy is obtained, the game tree is traversed in order to find out the information necessary to construct a CFR-D subgame gadget (opponent's counterfactual values and our reach probabilities). The counterfactual values can be computed either precisely (using counterfactual best response as in Burch et al. [2014]), or approximately (using just the trunk strategy). To specify a subgame to be resolved and a resolving player, a user provides a sequence of action indices. These indices correspond to a single game state. The resolving player is the acting player of this state, and the subgame is the smallest CFR-D subgame containing this state. The combined strategy is obtained by taking the trunk strategy and replacing all the resolving player's information set strategies from the subgame with the corresponding resolved strategy.

The other commands are `run` (which runs a single match of a given game and two given players), `tournament` (which runs multiple matches with given settings and aggregates the results), `game-info` (which returns information about the size of a given game), and of course `help`.

5. Experimental evaluation

In this chapter we evaluate the performance of continual resolving in different games. We start by evaluating offline performance of CFR with different hyperparameters. We then select several of the best solvers for use in the online evaluation. We again start by selecting the best values for online-specific hyperparameters. Finally, we compare continual resolving to online CFR. All evaluation is done in perfect-recall variants of selected games. We used GNU Parallel [Tange, 2018] to run the evaluation jobs concurrently on up to two Linux machines with 24-core Xeon E5-2680 v3 (48 HT threads) processors, and 128 GB and 256 GB RAM respectively. JDK 11.0.2 was used for all evaluation runs. The results of the experiments are attached to the thesis. More details can be found in Appendix A.1.

5.1 Solver hyperparameter selection

As we described in Chapter 3, our CFR and MC-CFR solvers have a variety of hyperparameters. Some of those hyperparameters are present in both CFR and MC-CFR. However, we evaluate them separately because of the differences between the two algorithms. Given the large number of possible combinations of hyperparameter value choices, we optimize each hyperparameter on its own, combine the best values together and compare them to configurations used in other papers.

The following games have been selected for the evaluation:

Game	$\lvert H\rvert$	$\lvert\mathcal{I}\rvert$	$\lvert\Delta_u\rvert$	$\lvert A\rvert$
PerfectRecall{LatentTicTacToe{}}	4854291	342196	2	9
IIGoofspiel{6}[1]	2006323	166002	2	6
PerfectRecall{LeducPoker{100,100,3,30}}	4362901	50640	38	3
PerfectRecall{PrincessAndMonster{8}}	2570339	3218	28	4

Table 5.1: Command-line configuration keys for selected games and their properties.[2] $\Delta_u = \Delta_{u,1} = \Delta_{u,2}$ for zero-sum games. Additionally, for all selected games $|A_1| = |A_2|$, where $|A_i| = \max_{h \in H:\, P(h)=i} |A(h)|$.

Each solver was tested in all games with time limit of 30 minutes, which allowed us to reach approximately 10^9 visited states in all games. The accumulated strategy was evaluated every 30 seconds.

5.1.1 CFR

Our CFR implementation has the following hyperparameters: a regret matching algorithm, alternating/simultaneous updates and cumulative strategy discounting exponent. Results for each hyperparameter are averaged over four runs.

[1] II-Goofspiel is implemented with perfect recall.

[2] Game parameters are explained in 3.2.

We start by comparing alternating and simultaneous updates. We used the basic regret matching and no cumulative strategy discounting for both cases.

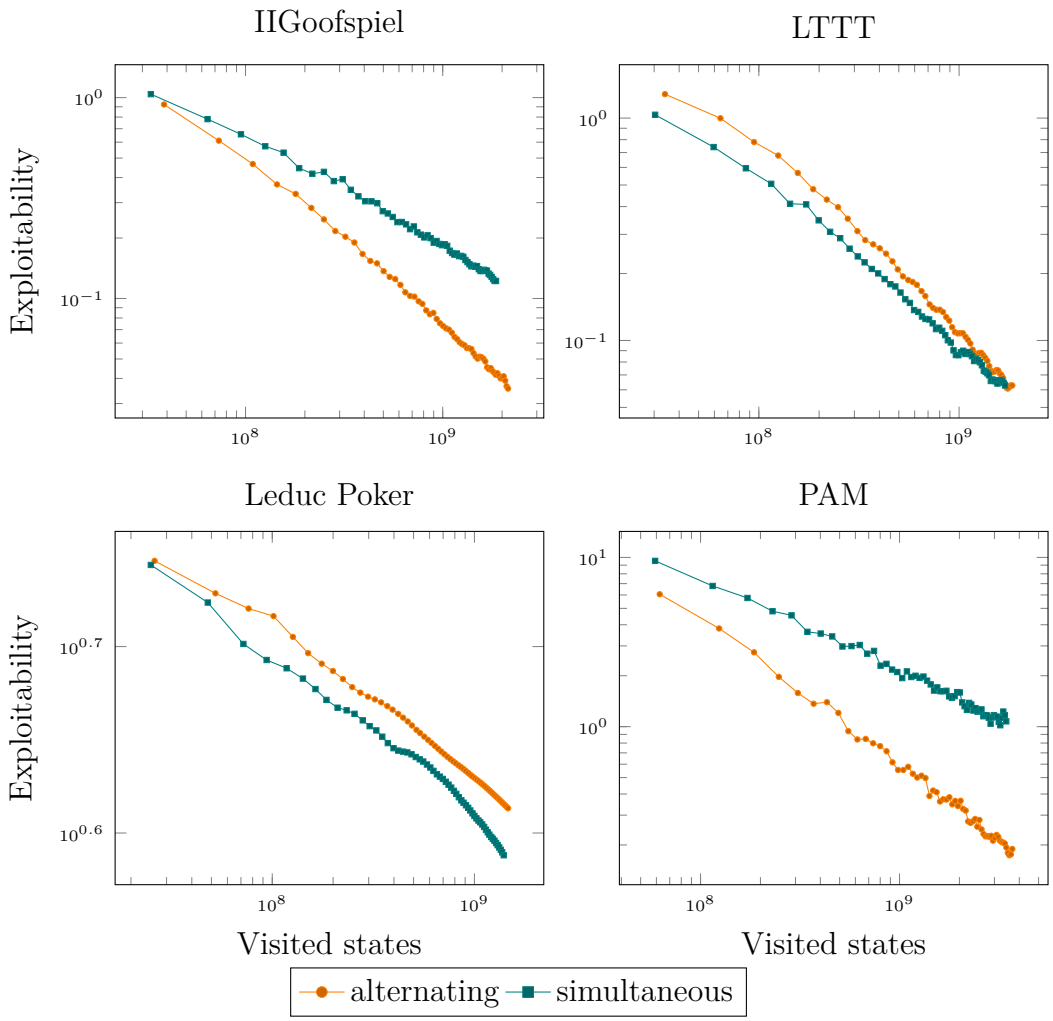

Figure 5.1: Comparison of alternating and simultaneous updates in CFR.

In Figure 5.1 we can see that neither option is clearly better than the other, however given the much better performance of alternating updates in goofspiel, and princess and monster we chose alternating updates for our solver candidates.

Next we compare different values of cumulative strategy discounting exponents. When cumulative strategy σ is updated at iteration $t+1$ with discounting exponent γ, it is multiplied by $\left(\frac{t}{t+1}\right)^{\gamma}$ before accumulating the current strategy. Sensible values for γ are from the interval $[0, +\infty)$. We try values from 0 to 3 with 0.3 increments. We use regret matching^{+} and alternating updates for the comparison.

Figure 5.2 shows that apart from zero, the other values have similar performance. Upon closer inspection we concluded that values between one and two seem to work best, we selected 1.5.

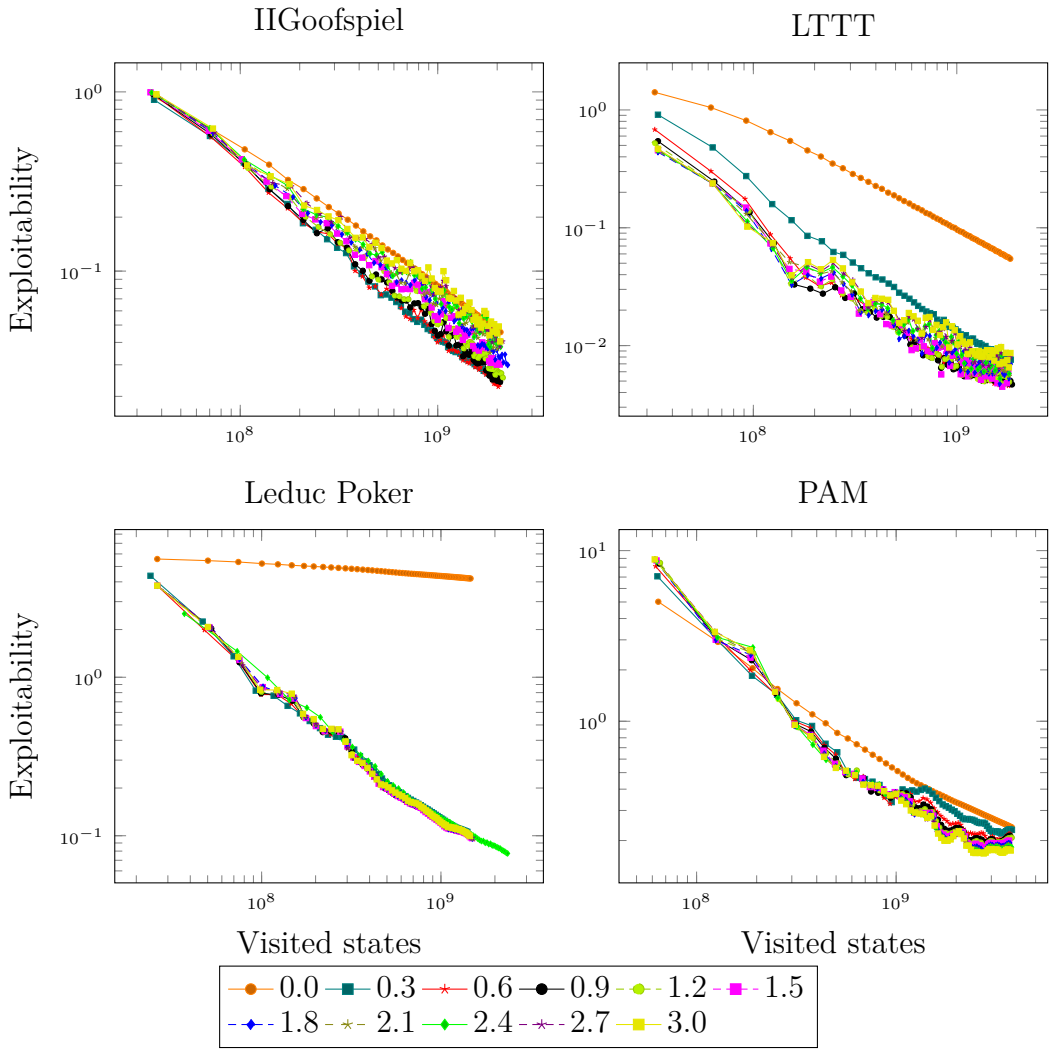

Figure 5.2: Comparison of different cumulative strategy discount exponents for CFR.

Finally, we have to choose a regret matching algorithm. We do not consider explorative regret matching at this stage. That leaves us with regret matching, regret matching$^+$, and discounting regret matching. Discounting regret matching (DRM) has two hyperparameters α and β. At each iteration t accumulated positive regret is multiplied by $\frac{t^\alpha}{t^\alpha+1}$ before adding new regret, similarly for β and negative regret. Therefore, the higher the value, the less is the corresponding regret discounted. This formulation allows DRM to emulate both regret matching ($\text{DRM}_{\infty,\infty}$) and regret matching$^+$ ($\text{DRM}_{\infty,-\infty}$).

We evaluate each DRM hyperparameter separately with cumulative strategy discounting exponent equal to two and alternating updates and the other hyperparameter fixed to ∞. The results for α are in Figure 5.3. We selected $\alpha = 1.5$, which is among the best values in three of the tested games. The results for β are in Figure 5.4. We selected $\beta = 0.5$, which is among the best values in all four tested games.

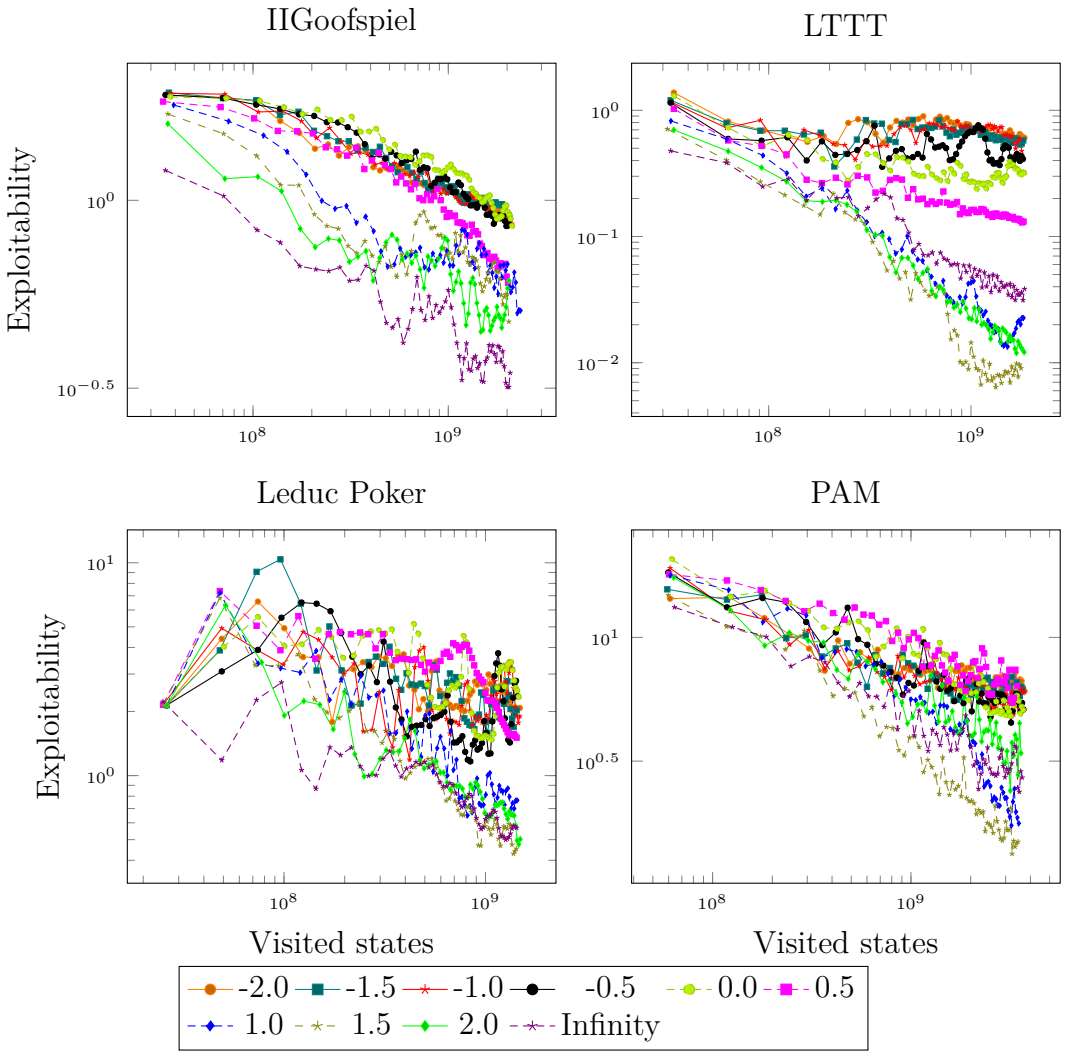

Figure 5.3: Comparison of different α for DCFR$_{\alpha,\infty,2}$.

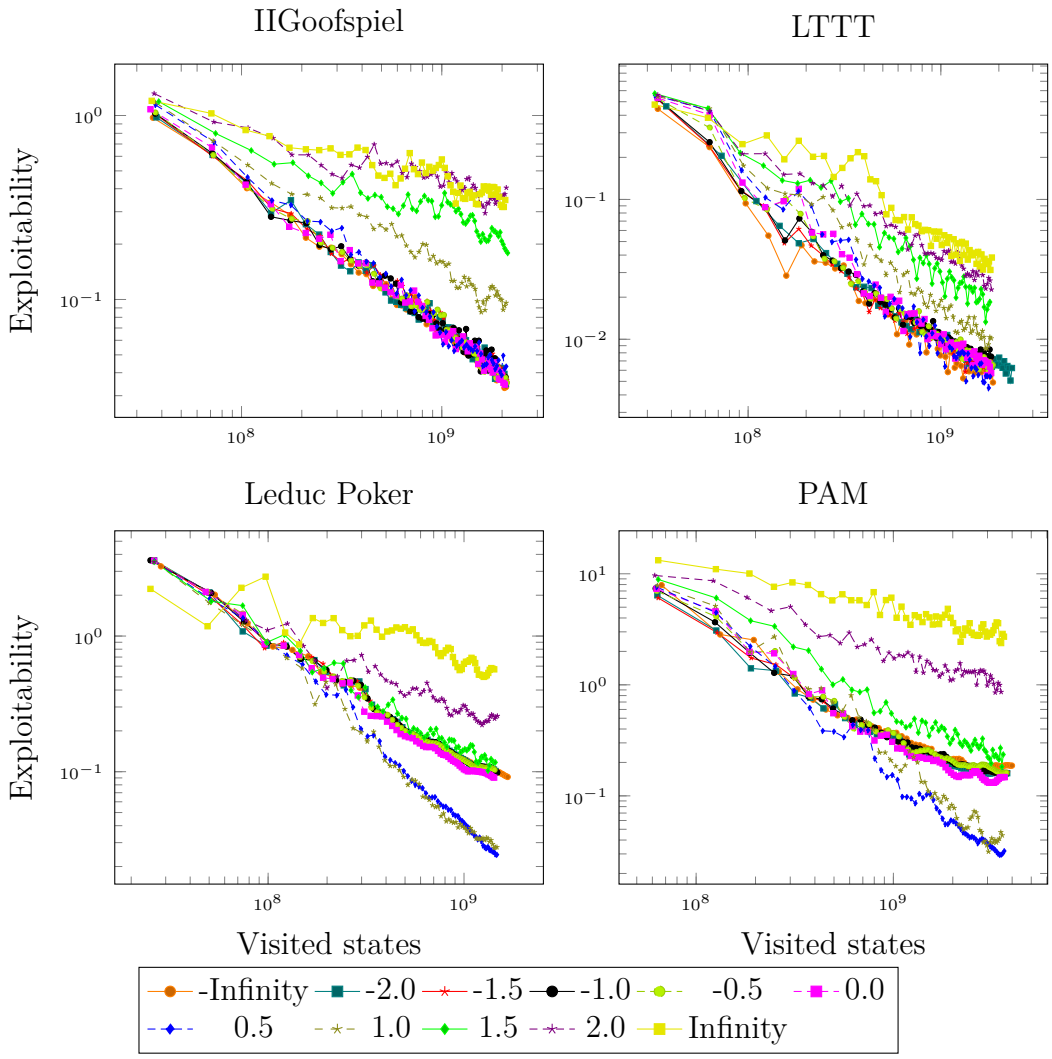

Figure 5.4: Comparison of different β for DCFR$_{\infty,\beta,2}$.

If we put all the selected hyperaparameters together, we get our candidate – $\text{DCFR}_{1.5,\,0.5,\,1.5}$ with alternating updates. We now compare it with configurations we found in other papers. The CFR configurations are further described in Table 5.2.

The results are in Figure 5.5. We can see, that both CFR^+ variants are among the best solvers in all the games we tested. Our DCFR candidate also performed quite well, and it managed to beat CFR^+ in some games. In the end we selected $\text{DCFR}_{1.5,\,0.5,\,1.5}$ and CFR^+ 1 for online evaluation.

CFR	Original CFR with simultaneous updates
CFR au	Original CFR with alternating updates
Deepstack CFR	CFR with regret matching$^+$, simultaneous updates and no cumulative strategy discounting. Deepstack additionally discards strategy from early iterations, which we do not support.
CFR^+ 1	Original CFR^+ (with cumulative strategy discounting exponent equal to 1)
CFR^+ 2	As above but with cumulative strategy discounting exponent equal to 2 (as suggested by Brown and Sandholm [2018])
$\text{DCFR}_{1.5,\,0,\,2}$	Discounted CFR configuration suggested by Brown and Sandholm [2018]
LCFR	Linear CFR [Brown and Sandholm, 2018]

Table 5.2: Table of standard CFR configurations.

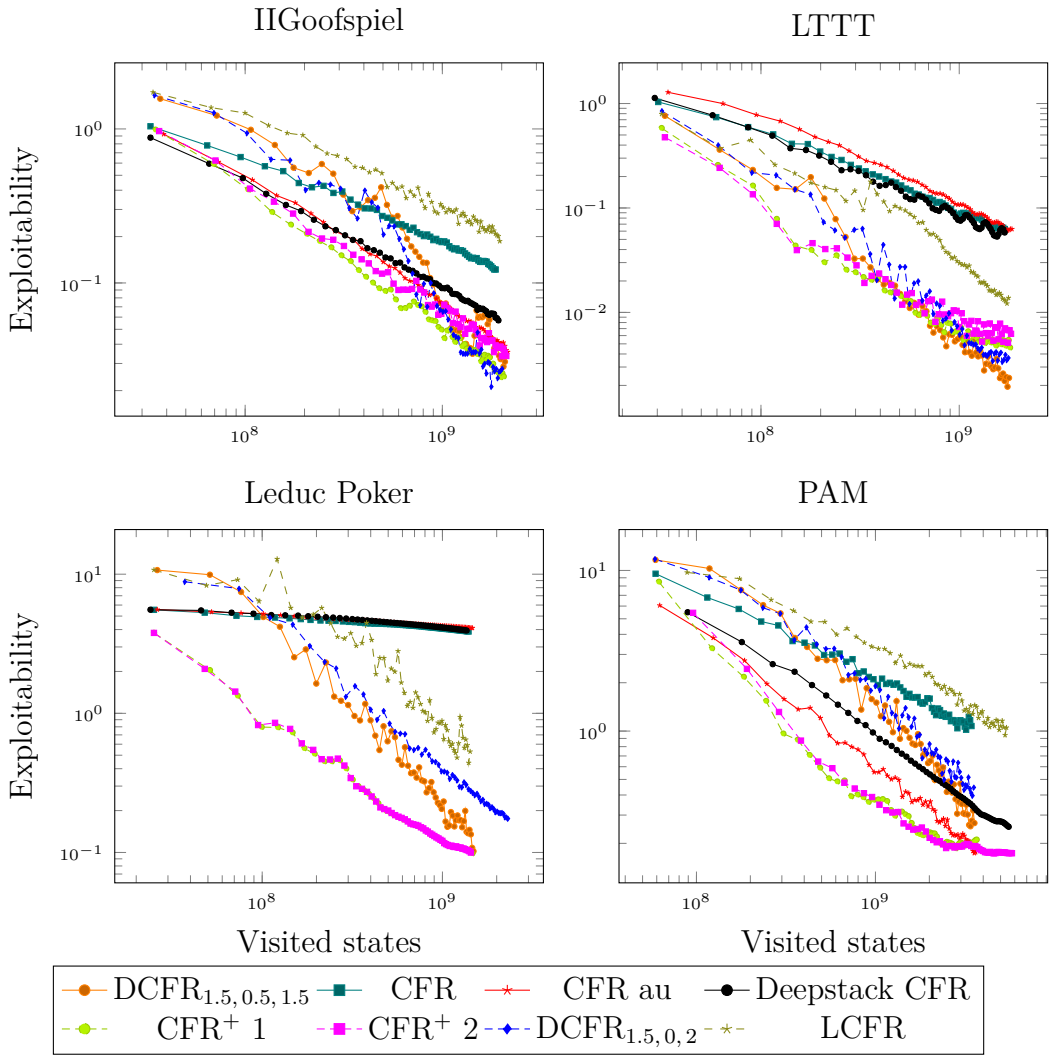

Figure 5.5: Comparison of our CFR candidate to options used by other papers.

5.1.2 MC-CFR

Our implementation of MC-CFR with outcome sampling supports the following hyperparameters: a regret matching algorithm, a cumulative strategy discounting exponent, an exploration probability, and a baseline for variance reduction in MC-CFR. Results for each hyperparameter are averaged over 20 runs.

We start by comparing different values of the cumulative strategy discounting exponent. We do this with regret matching, exploration probability equal to one and no baseline.

The results in Figure 5.6 indicate that values 0 and 0.3 are among the best. We chose 0.3, so that we have some discounting to better support in-match targeting in the online setting (since we do not increase the weight of future iterations when the targeting is changed).

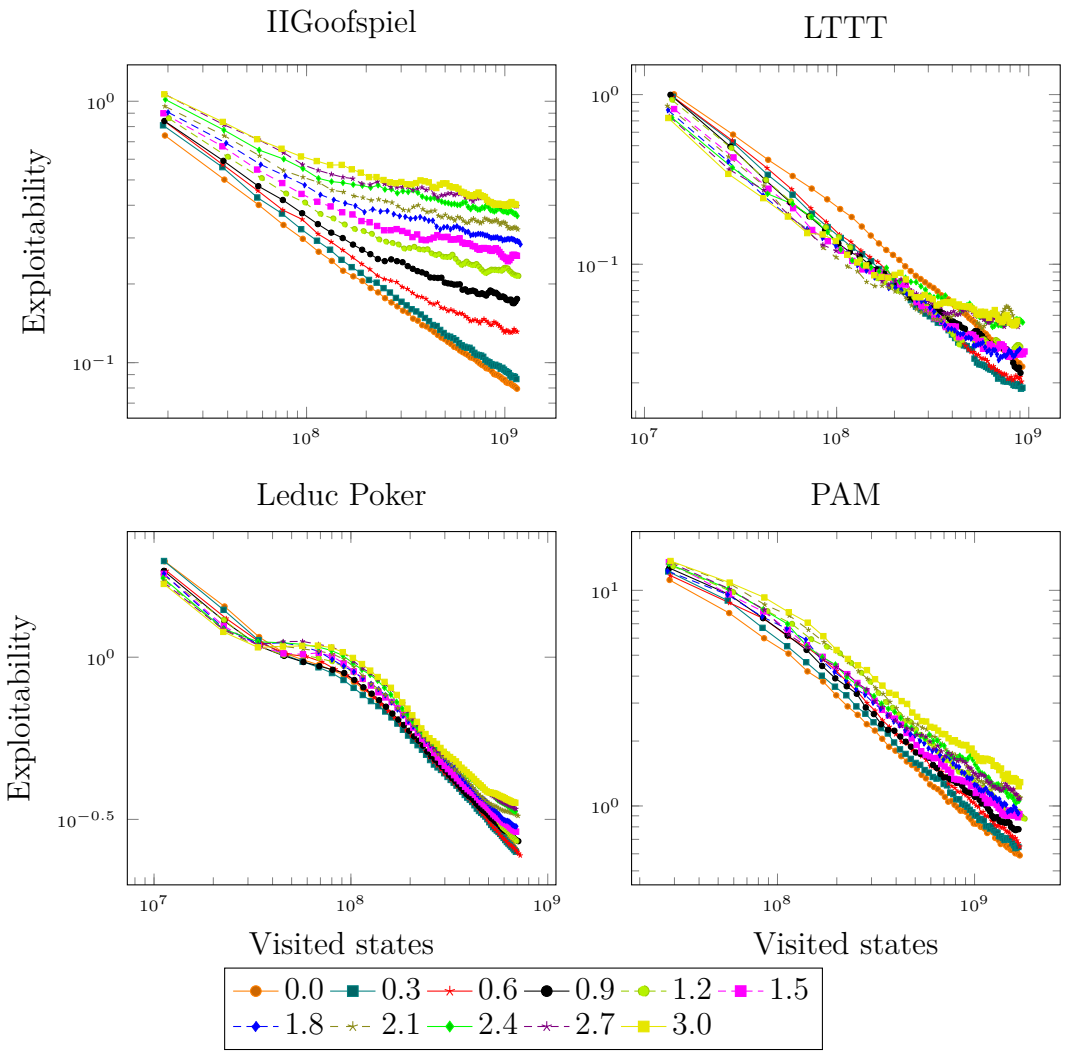

Figure 5.6: Comparison of different cumulative strategy discount exponents for MC-CFR.

Next we compare different values of the exploration probability. Our MC-CFR implementation uses alternating updates. Actions at iteration t, which updates player i, are sampled from the distribution described in Equation 1.2. The evaluation is done with regret matching, no cumulative strategy discounting and no baseline.

The results are shown in Figure 5.7. We chose exploration probability 0.7, which is among the best in all tested games.

As with CFR, we only evaluate different DRM configurations, since it can emulate the other regret matching algorithms we considered. The evaluation for both α and β was done with exploration probability 1, no cumulative strategy discounting, no baseline and the other hyperparameter fixed to ∞.

The results are in Figure 5.8 and Figure 5.9. We selected 1 as the best value for both hyperparameters.

The final hyperparameter is the decay factor for exponentially-decaying average baseline for VR-MCCFR. The evaluation was done with regret matching, exploration probability 1 and no cumulative strategy discounting. The results are in Figure 5.10. We selected 0.3 as the best value.

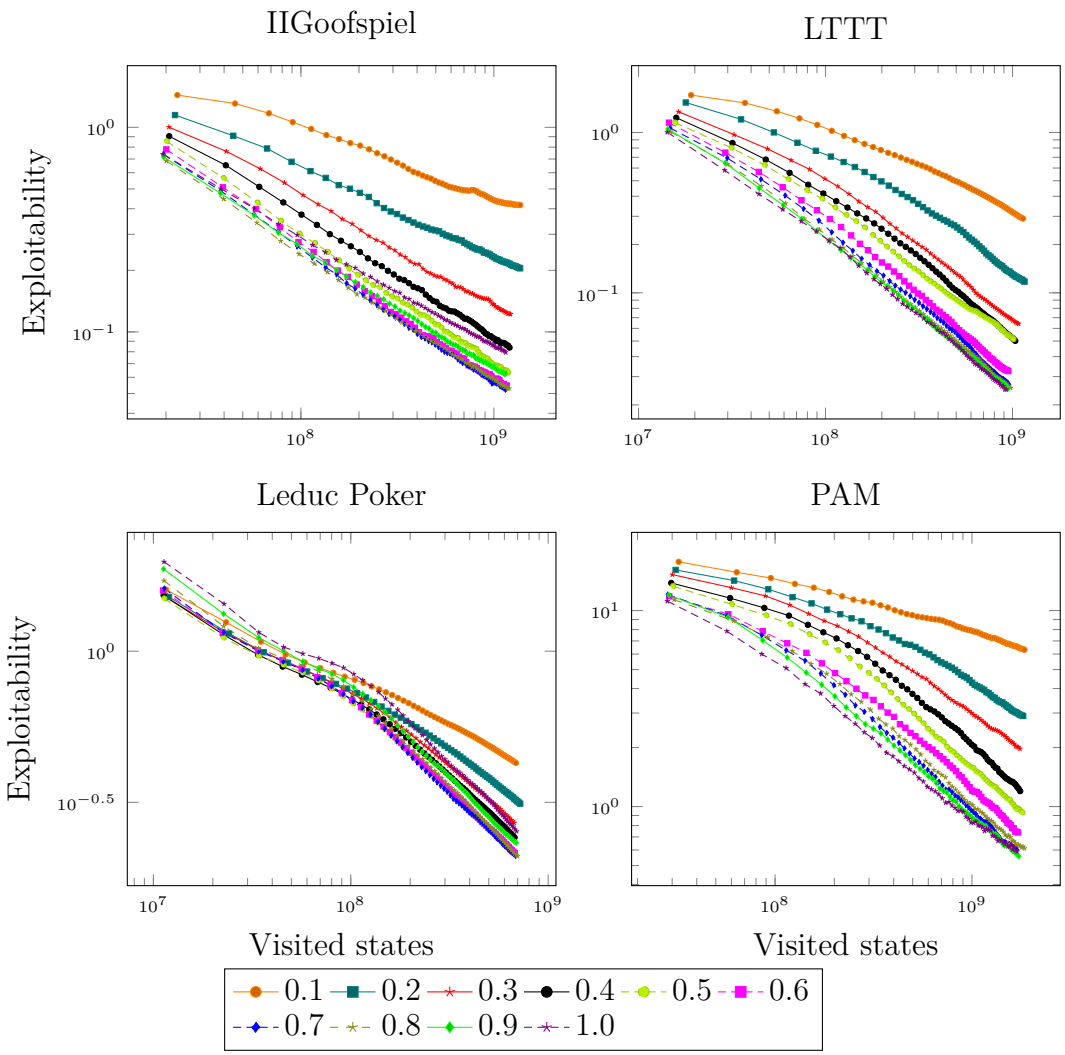

Figure 5.7: Comparison of different exploration probabilities for MC-CFR.

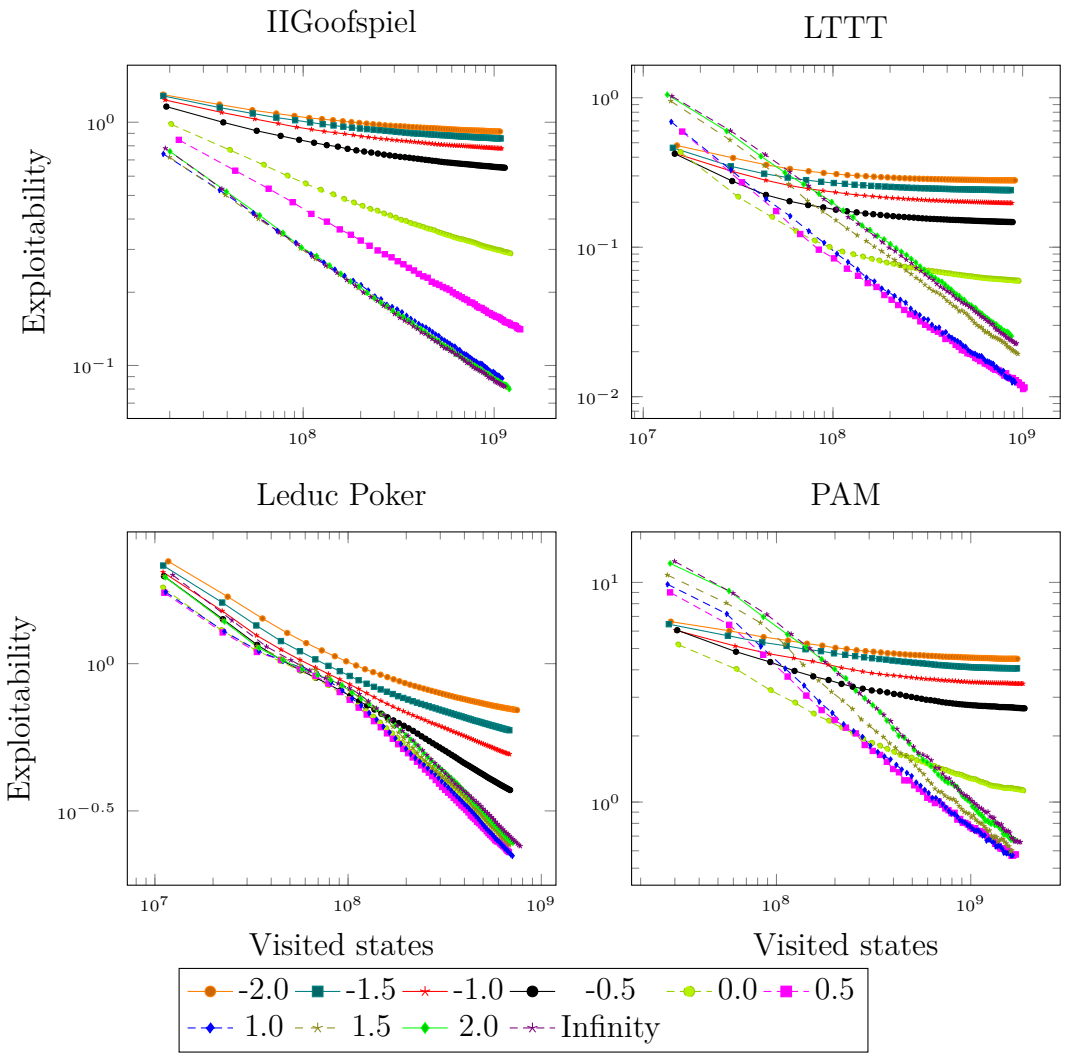

Figure 5.8: Comparison of different α for MC-CFR with $\mathrm{DRM}_{\alpha,\infty}$.

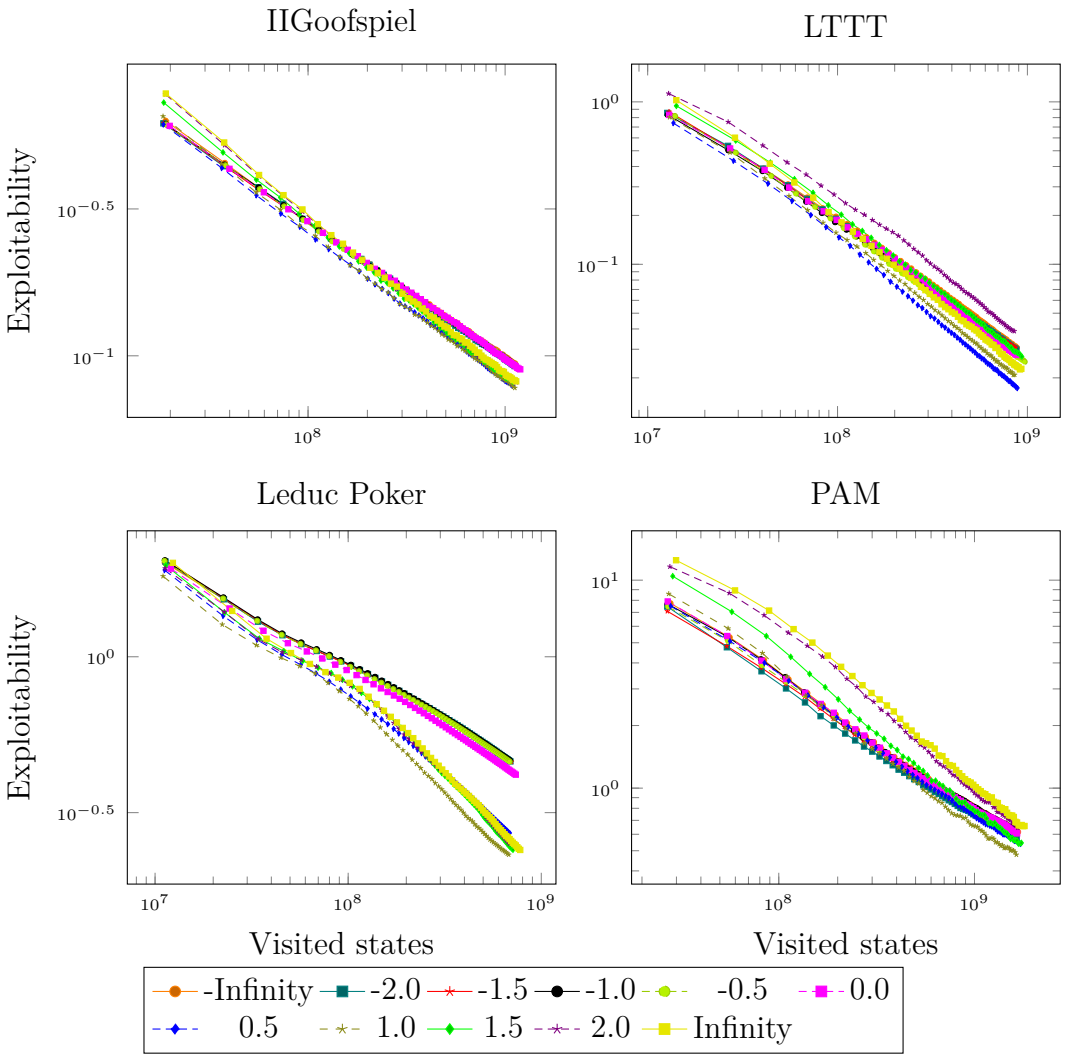

Figure 5.9: Comparison of different β for MC-CFR with $\mathrm{DRM}_{\infty,\beta}$.

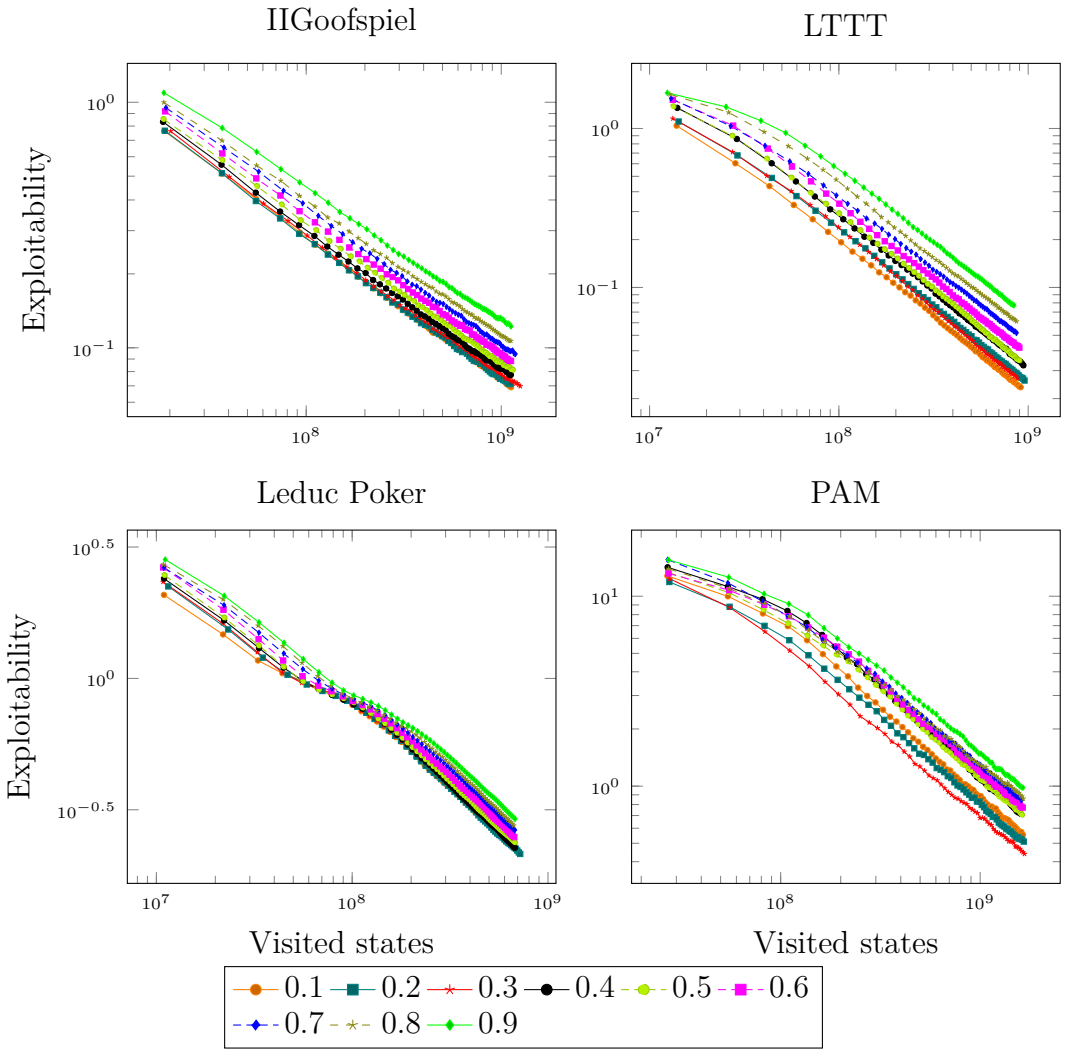

Figure 5.10: Comparison of different decay factors for exponentially-decaying average baseline for VR-MCCFR.

This gives us our two candidates – MC-CFR with $\text{DRM}_{1,1}$, cumulative strategy discounting exponent 0.3, exploration probability 0.7, and either no baseline, or exponentially-decaying average with decay 0.3. We again wanted it to compare to some standard configurations, however in this case there were not any clear candidates. In the end, we used mostly approximations of configurations mentioned in Schmid et al. [2018], however since the paper did not specify details such as the cumulative strategy discounting, we had to guess there. The configurations are listed in Table 5.3. All the configurations use exploration probability 1.

MC-CFR	Basic MC-CFR
MC-CFR$^+$	MC-CFR with regret matching$^+$
MC-DCFR$_{1.5,0,2}$	MC-CFR with $\text{DRM}_{1.5,0}$ and cumulative strategy discounting exponent 2
MC-LCFR	MC-CFR with $\text{DRM}_{1,1}$ and cumulative strategy discounting exponent 1
VR-MCCFR	VR-MCCFR with regret matching and exponentially decaying average baseline with decay 0.5
VR-MCCFR+	VR-MCCFR with regret matching$^+$, cumulative strategy discounting exponent 1, and exponentially decaying average baseline with decay 0.5

Table 5.3: Table of standard MC-CFR configurations.

Figure 5.11 shows that our candidates were the two best configurations in all the games we tested, with the VR-MCCFR candidate being better than the MC-CFR candidate. However, we used both candidates for the online evaluation.

All four candidates are compared in Figure 5.12. The MC-CFR candidates have a clear advantage in early convergence, however they get overtaken by CFR in the later stages. Note that the CFR lines are longer, because we limited the evaluation by time, rather than visited states, and CFR traverses the game tree more efficiently than MC-CFR.

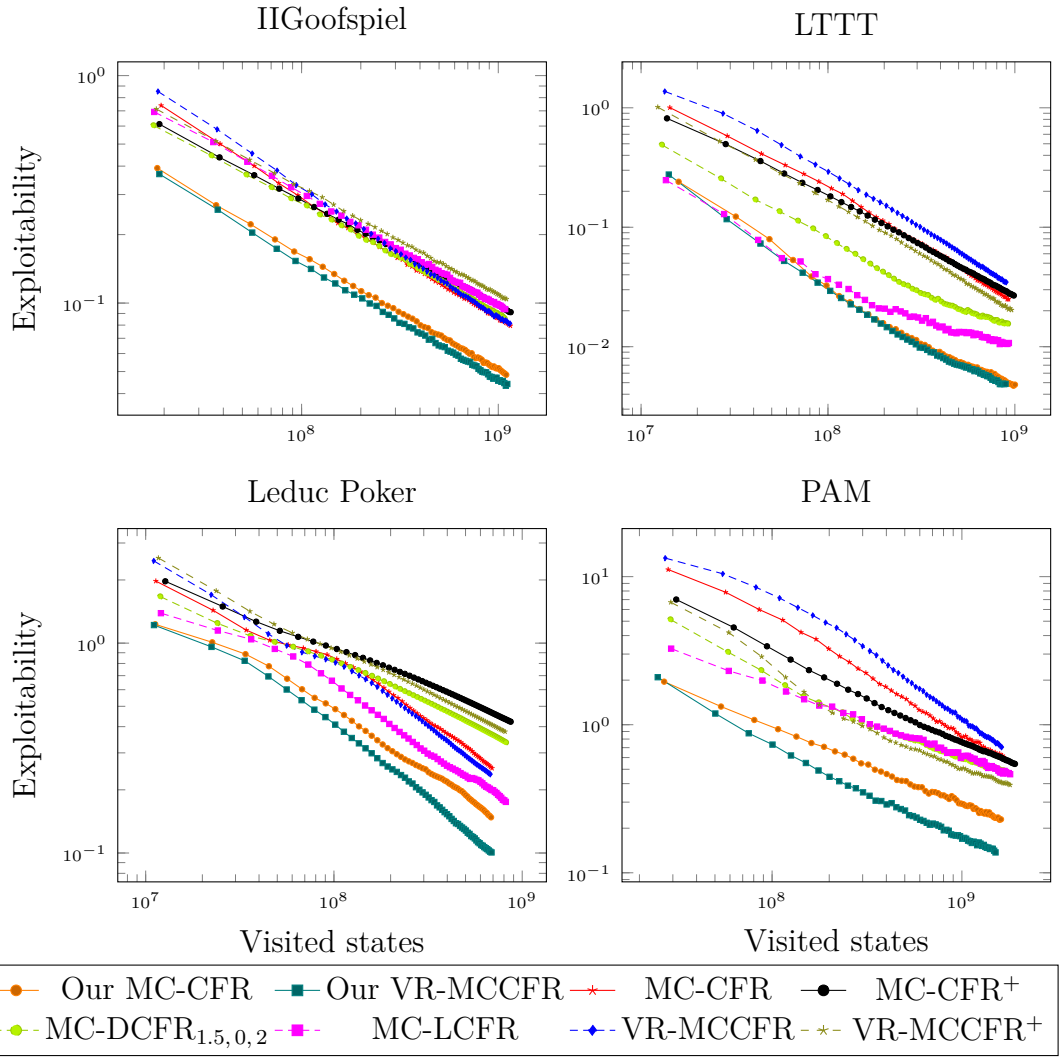

Figure 5.11: Comparison of our MC-CFR candidates to options used by other papers.

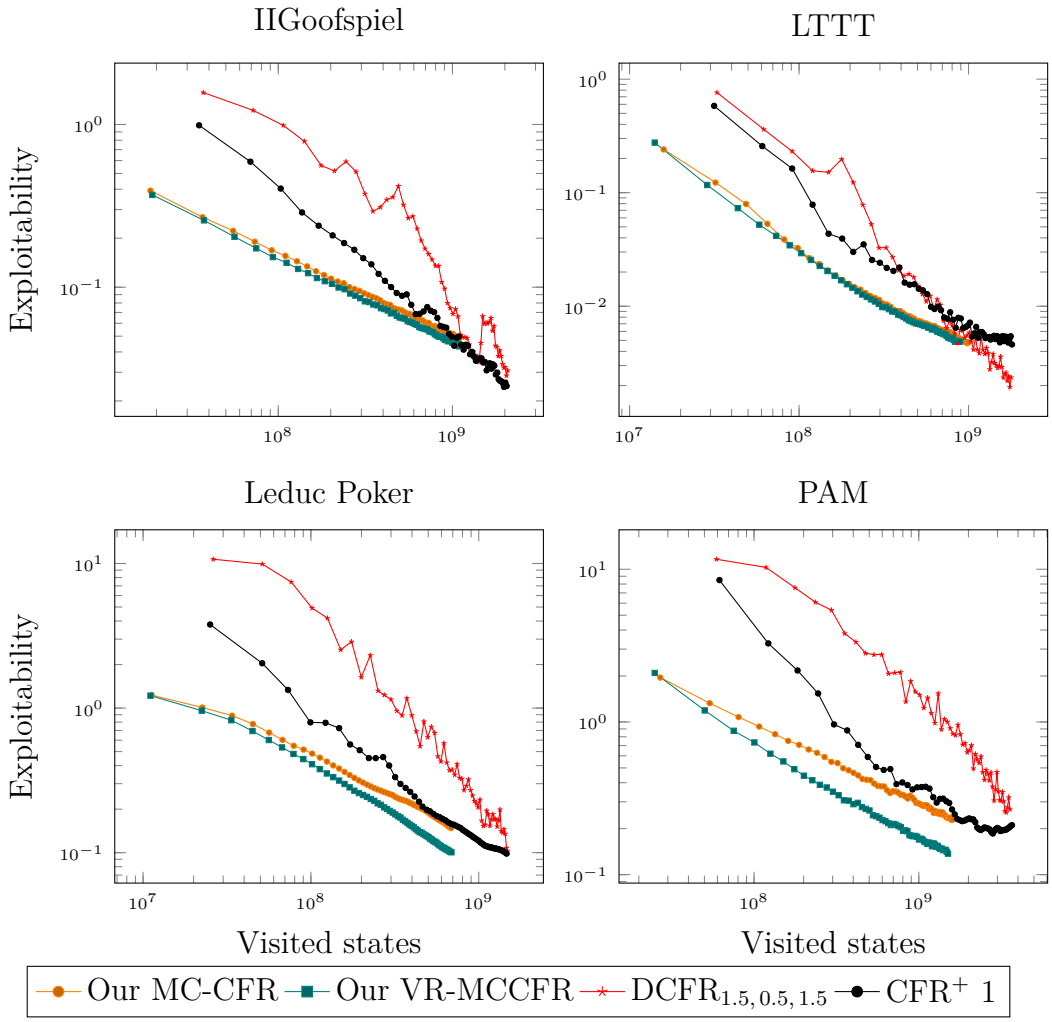

Figure 5.12: Comparison of our solver candidates for online evaluation.

5.2 Player hyperparameter selection

Our CFR and MC-CFR implementations both have online-specific hyperparameters. That is the depth-limit and utility estimation function for CFR, and information set targeting probability for MC-CFR. On top of that we also consider explorative regret matching as it was previously used for OOS MC-CFR [Lisý et al., 2015]. We again optimize each hyperparameter independently, and separately for both players (continual resolving and online solving).

The evaluation was done on two sets of games – one set of smaller games for the traversing evaluator (Table 5.4), and another set of larger games for the game-playing evaluator (Table 5.5). Since latent tic-tac-toe was too large, we include two variants of leduc poker – one with higher number of cards in each suite, and another one with higher number of allowed raises in each betting round – to keep the number of games the same. Strategies aggregated by game-playing evaluator were restricted the intersection of information sets contained by all strategies for given game (that is strategies for all players and their time-limits). This is especially important when comparing continual resolving to online solving, which produces a strategy for the whole game and is therefore able to cover a bigger part of the game-tree in a single match.

Game	$\lvert H\rvert$	$\lvert\mathcal{I}\rvert$	$\lvert\Delta_u\rvert$	$\lvert A\rvert$
IIGoofspiel{4}	2229	738	2	4
PerfectRecall{LeducPoker{100,100,4,3}}	6529	840	50	3
PerfectRecall{LeducPoker{100,100,1,7}}	11635	616	14	3
PerfectRecall{PrincessAndMonster{5}}	11283	146	16	4

Table 5.4: Command-line configuration keys for selected games and their properties. $\Delta_u = \Delta_{u,1} = \Delta_{u,2}$ for zero-sum games. Additionally, for all selected games $|A_1| = |A_2|$, where $|A_i| = \max_{h \in H:\, P(h)=i} |A(h)|$.

Game	$\lvert H\rvert$	$\lvert\mathcal{I}\rvert$	$\lvert\Delta_u\rvert$	$\lvert A\rvert$
IIGoofspiel{5}	55731	9948	2	5
PerfectRecall{LeducPoker{1000,1000,14,3}}	62749	7920	170	3
PerfectRecall{LeducPoker{100,100,1,12}}	58225	1776	14	3
PerfectRecall{PrincessAndMonster{6}}	72531	402	20	4

Table 5.5: Command-line configuration keys for selected games and their properties. $\Delta_u = \Delta_{u,1} = \Delta_{u,2}$ for zero-sum games. Additionally, for all selected games $|A_1| = |A_2|$, where $|A_i| = \max_{h \in H:\, P(h)=i} |A(h)|$.

5.2.1 Explorative regret matching

Explorative regret matching adds a uniform strategy to the regret-matched strategy with a small weight γ. We evaluate different values of γ separately for continual resolving and online solving. We use CFR$^+$ as the underlying solver in both cases. The evaluation is done on the games from Table 5.4 using traversing evaluator.

The results are in Figure 5.13 and Figure 5.14 for continual resolving and online solving respectively. However, there does not seem to be any benefit in using explorative regret matching with either player.

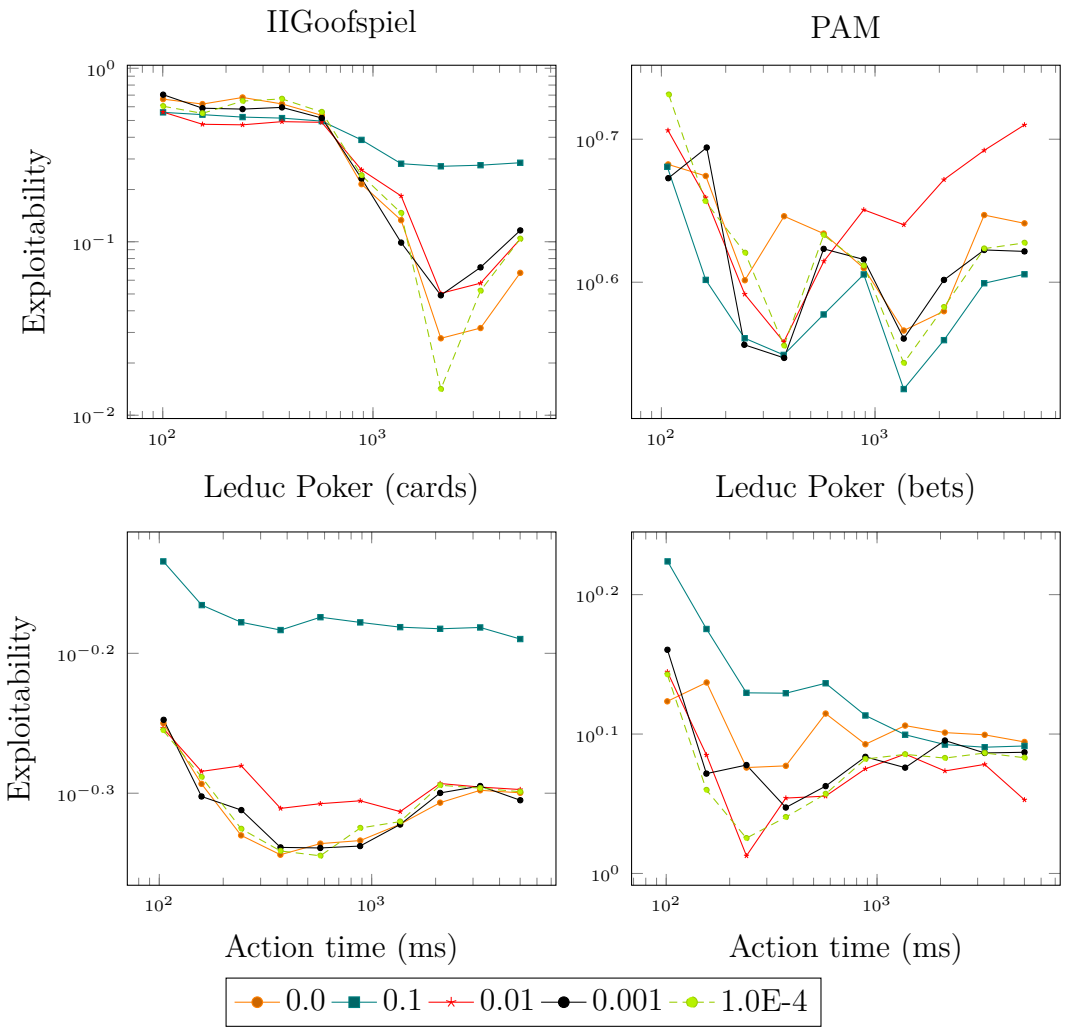

Figure 5.13: Comparison of different exploration weights for explorative regret matching with continual resolving.

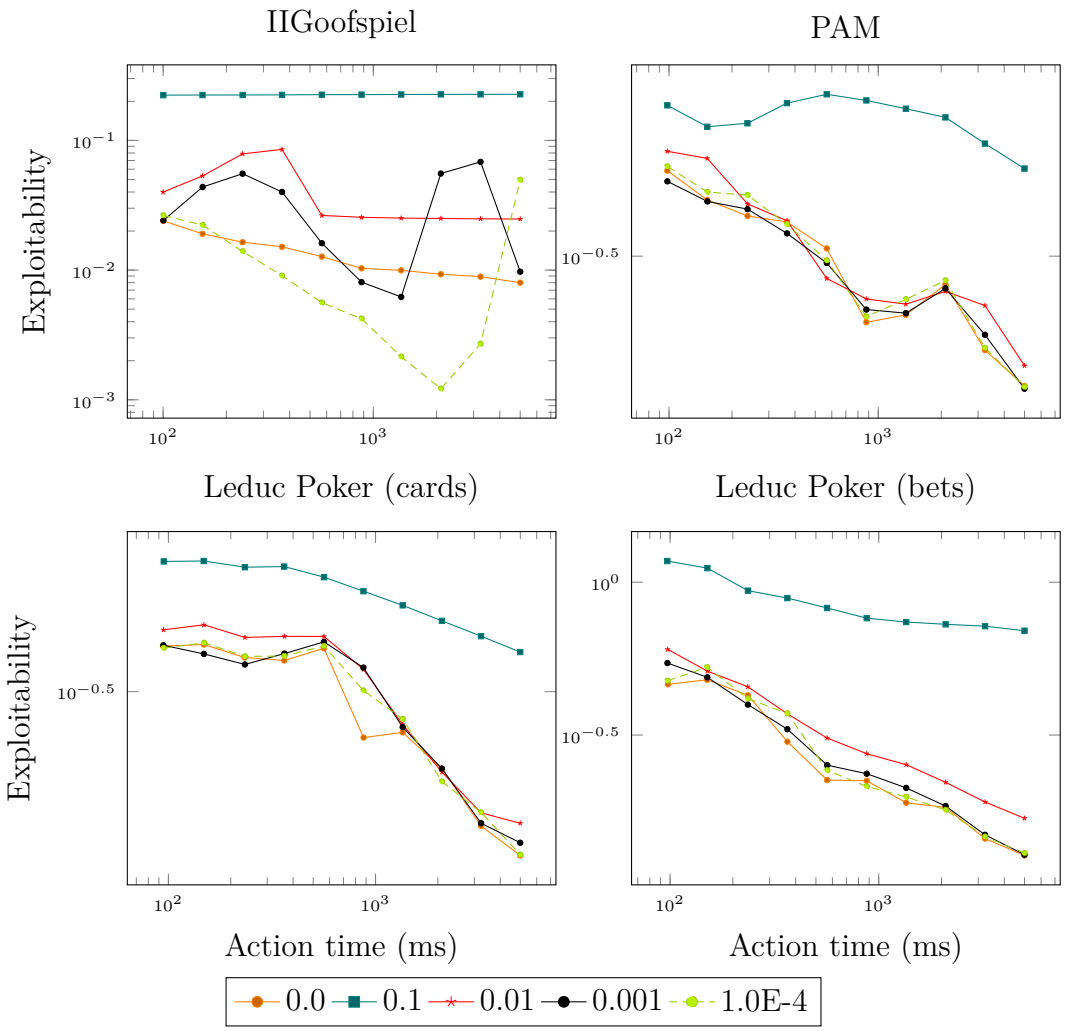

Figure 5.14: Comparison of different exploration weights for explorative regret matching with online solving.

5.2.2 Information set targeting

Information set targeting is only applicable for MC-CFR solvers. It works by sampling the targeted portion of the game (in this case the states contained in a current information set and their descendants) with higher probability. At the beginning of each MC-CFR iteration a scenario is decided – with probability t the iteration will be targeted, otherwise it will be not be targeted. If the iteration is targeted, only states from the targeted portion of the game can be sampled.

We again evaluate different values of t for both players separately. We use the VR-MCCFR candidate from the offline evaluation as the underlying solver. The evaluation is done on the games from Table 5.4 using traversing evaluator.

The results for continual resolving are in Figure 5.15. The results seem to be very game-dependent and there is no clear best option. We chose $t = 0$ which is among the reasonably behaving options.

The results for online solving are in Figure 5.16. They are certainly more consistent than the results for continual resolving. While $t = 0$ seems to be the best option, we chose $t = 0.1$ which is very close in terms of performance, however it should give the player a better chance in larger games.

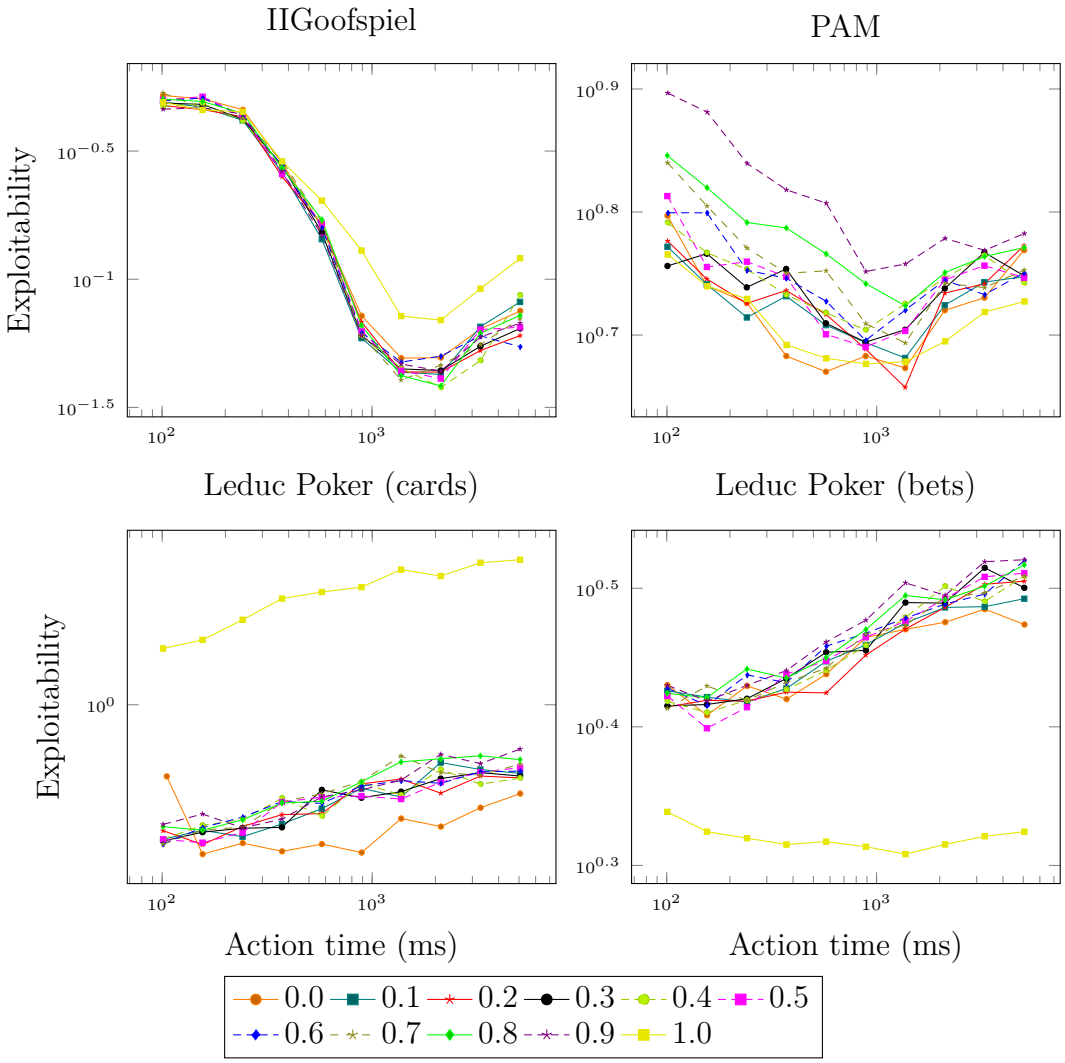

Figure 5.15: Comparison of different information set targeting probabilities with continual resolving.

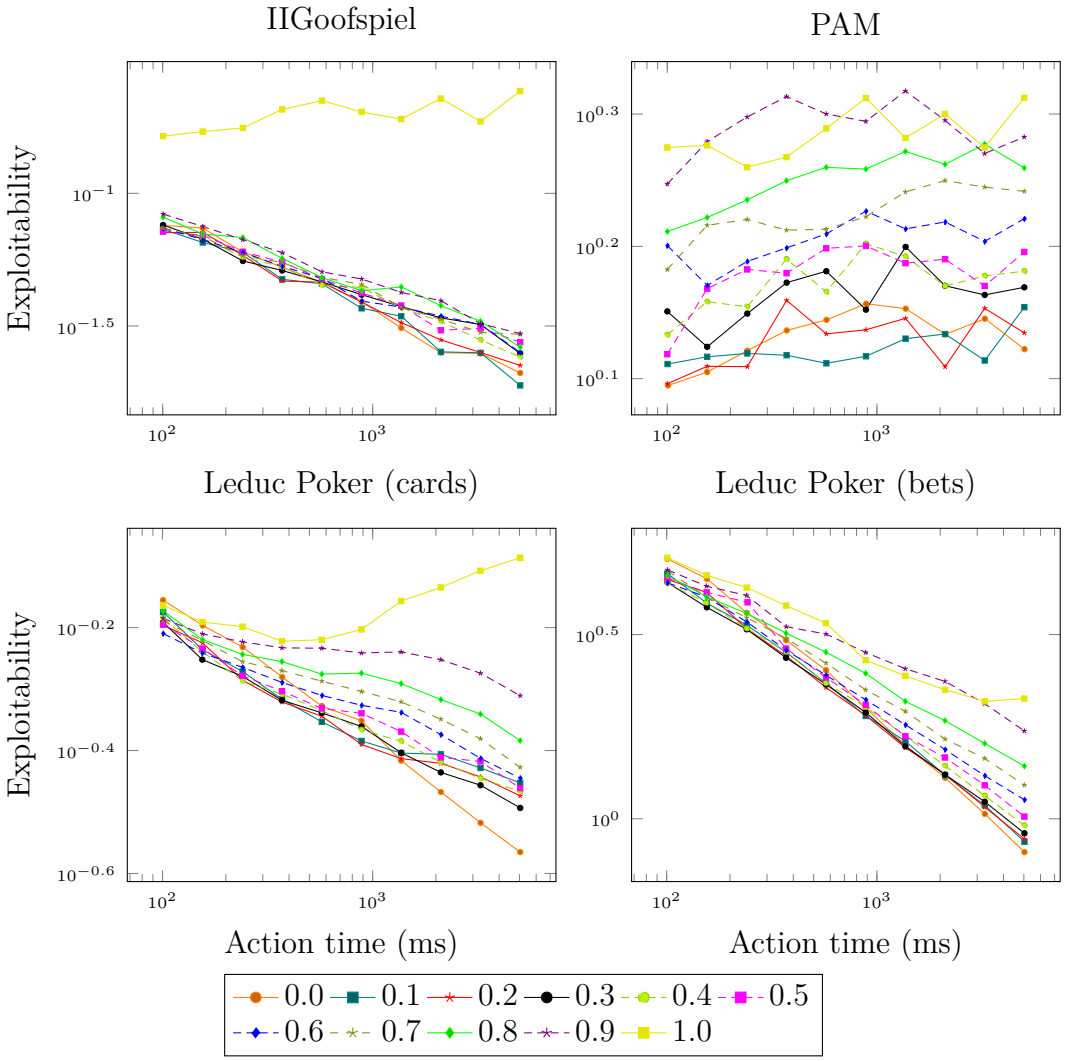

Figure 5.16: Comparison of different information set targeting probabilities with online solving.

5.2.3 Depth-limited CFR

Depth-limited CFR works by replacing the computation below certain depth with an evaluation function. It therefore cannot be used with online solving, since there we need to compute a strategy for the whole game. In our case we only have a uniform random playout evaluation function. We evaluate different numbers of samples used for each evaluation and different depth-limits (higher depth-limit = less iterations, but more precise estimates). We use CFR$^+$ as the underlying solver. The evaluation is done on the games from Table 5.5 using game-playing evaluator with 1000 matches per evaluated value.

First we evaluate the different numbers of samples. The results are in Figure 5.17. However, there seems to be almost no difference between the options. Therefore, we chose to use only 1 sample for the evaluation function.

Next we evaluate different depth-limits. A depth-limit for continual resolving must be high enough to reach our next turn in another subgame (in order to collect the information necessary to advance to another subgame). This is ensured automatically (see Appendix A.5 for more details). A depth limit d therefore means that the solver will go to at least such depth, but possibly more. The minimal depth-limit for a CFR-D gadget is 4 (initial chance node $\rightarrow$ opponent's choice node $\rightarrow$ our node $\rightarrow$ opponent's node $\rightarrow$ our node in another subgame).

The results are in Figure 5.18. There again seems to be almost no difference between the options. We therefore chose $d = 0$ (as little as necessary).

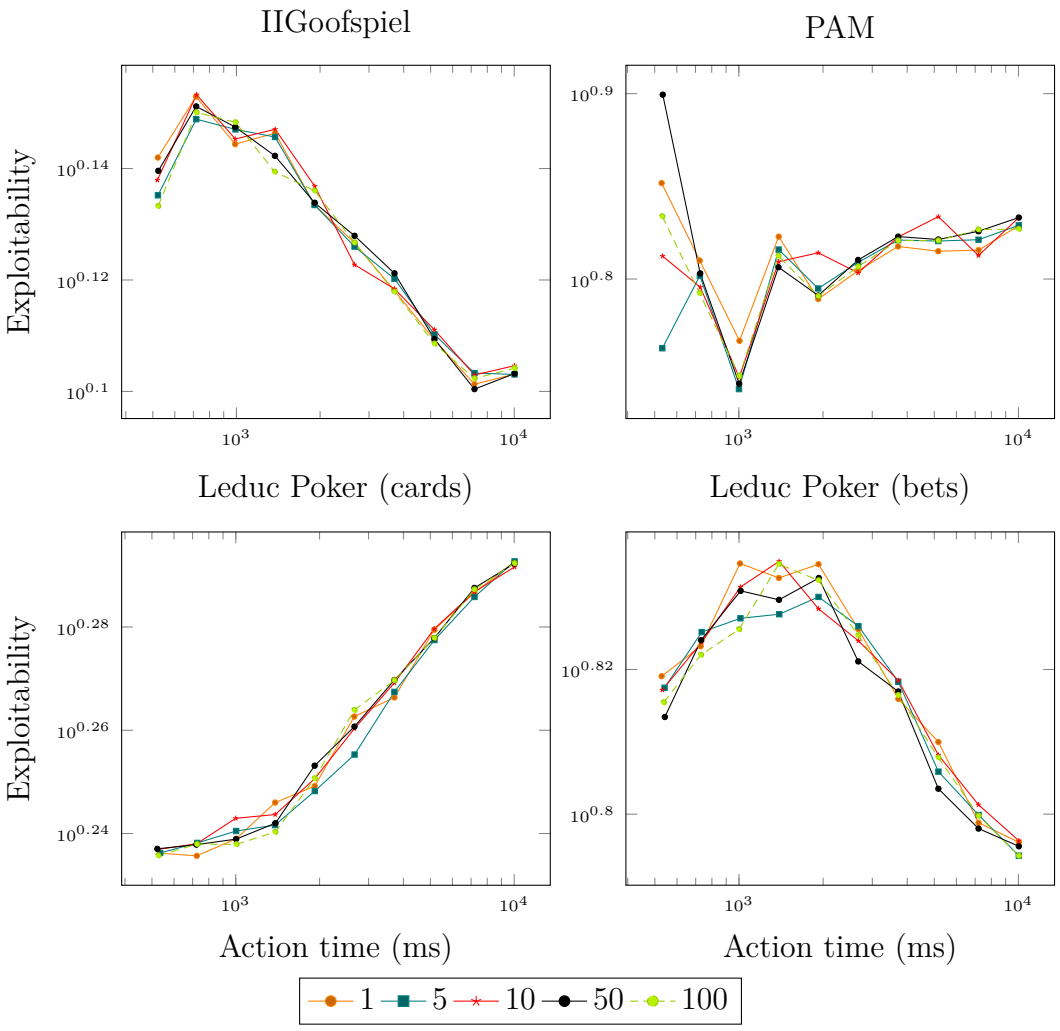

Figure 5.17: Comparison of different numbers of random samples for utility estimation for depth-limited CFR with continual resolving.

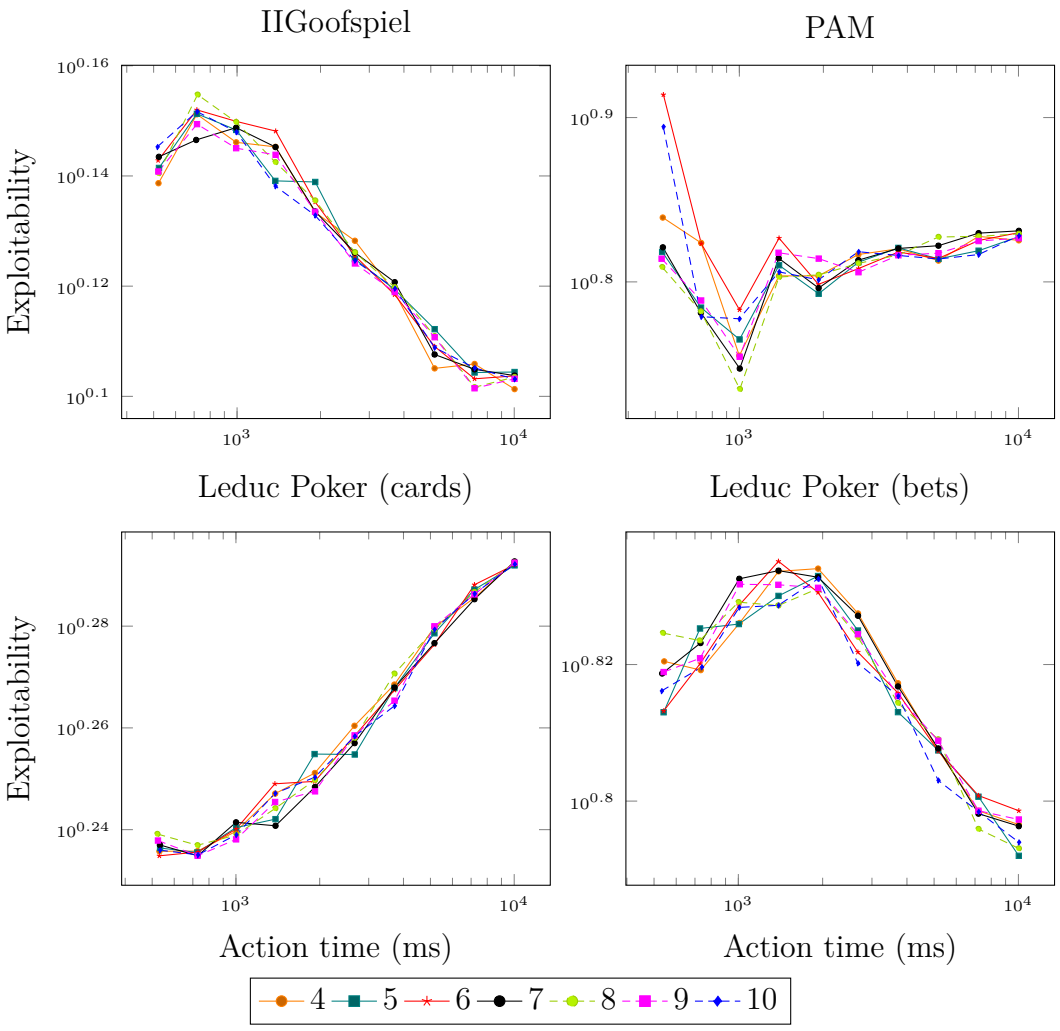

Figure 5.18: Comparison of different depth-limits for depth-limited CFR with continual resolving.

5.3 Final evaluation

In the previous section we selected online-specific hyperparameters for our players. Now we can compare their relative performance. But first we compare the MC-CFR variants to CFR ones for baseline comparison. The evaluation is done on the games listed in Table 5.5 using the game-playing evaluator with 1000 matches per player configuration. Players had 1 second of pre-play time and between 0.5 and 20 seconds of additional time per move.

Figure 5.19 shows that except for Leduc poker with higher amount of possible bets, the MC-CFR variants of continual resolving outperform the ones using depth-limited CFR with the uniform random playout as an evaluation function. This is not unexpected, given that the evaluation function is very naive.

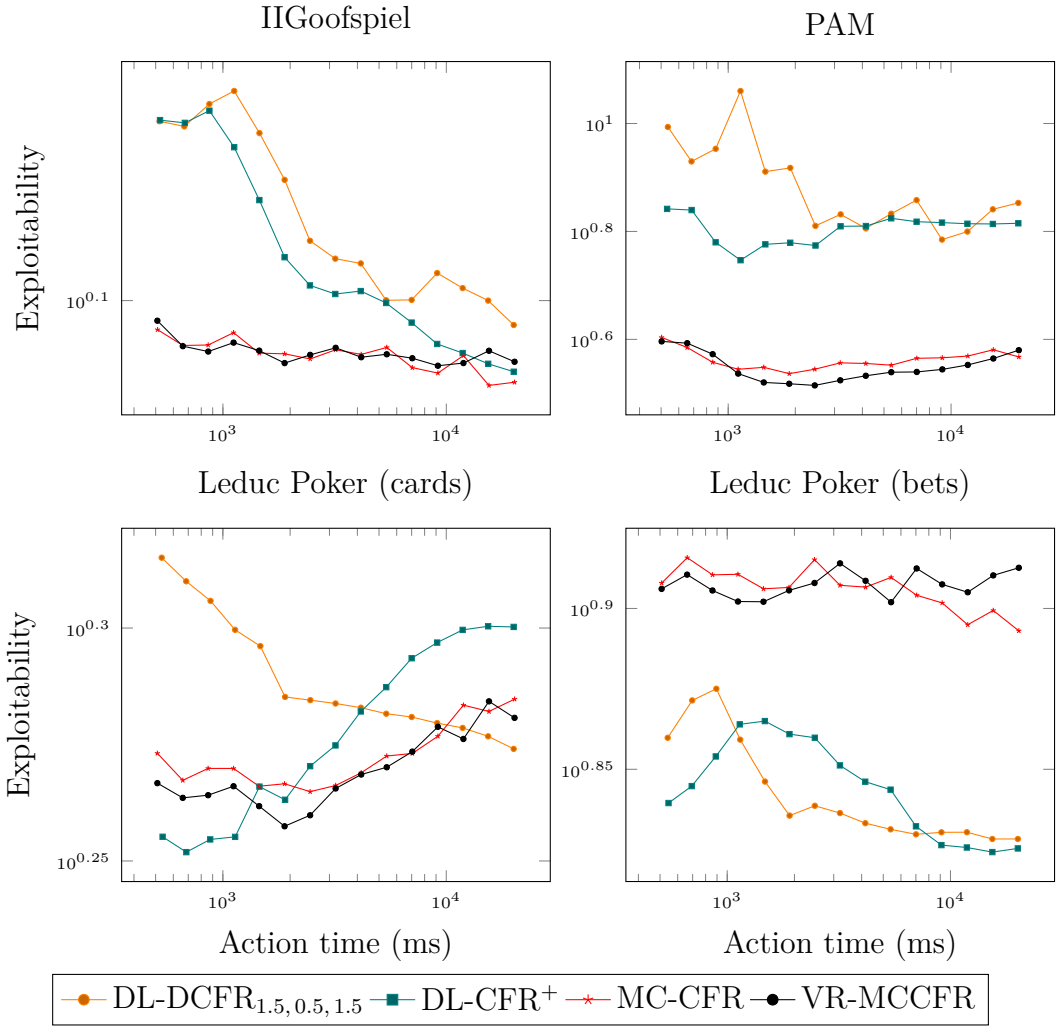

Figure 5.19: Comparison of continual resolving with different solvers.

On the other hand, Figure 5.20 shows that online solving performs better with CFR solvers. However, this is likely due to the smaller size of the tested games.

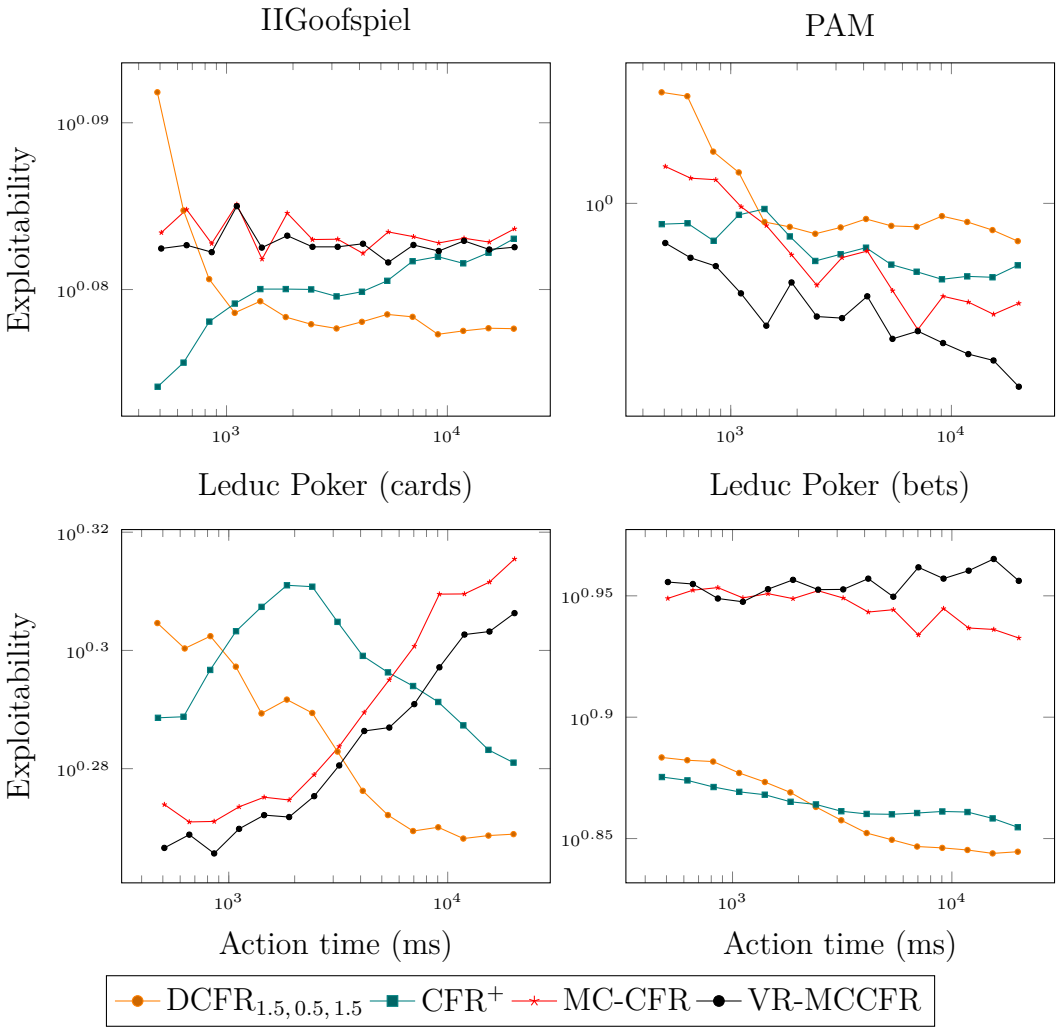

Figure 5.20: Comparison of online solving with different solvers.

Figure 5.21 compares continual resolving and online solving, with both using the VR-MCCFR solver. Continual resolving seems to perform better in both variants of Leduc poker, which unlike the other two games have public actions and consequently smaller subgames. This is especially visible in the Leduc poker with higher amount of possible bets, which has even smaller subgames than the other Leduc poker variant, but has much more of them. On the other hand, in princess and monster the only public information is how many turns have been played so far, leading to subgames with roots corresponding to states with the same depth.

However, it is possible that the advantage of continual resolving in games with smaller subgames could be diminished by using public subgame targeting for the solving player.

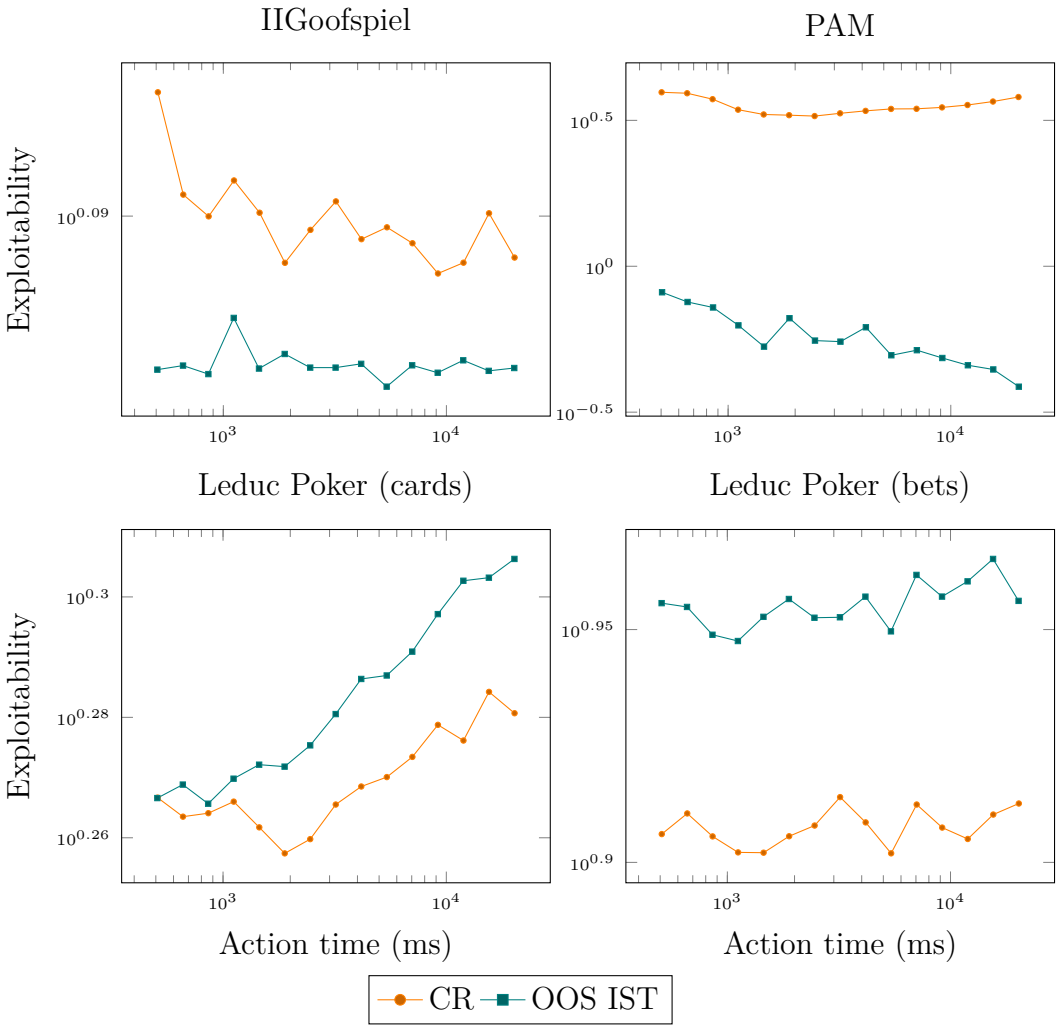

Figure 5.21: Comparison of continual resolving and online solving with VR-MCCFR as the underlying solver.

Finally, we compare head-to-head performance of different players on games from Table 5.1, which we used for the offline evaluation. The players had 10 seconds of pre-play time and additional 30 seconds per move. We ran 20,000 matches for each game and each pair of players (10,000 for both role assignments). The players are described in Table 5.6.

Player	Description
CR	Continual resolving (our VR-MCCFR candidate).
OOS IST	Our online solving candidate.
RND	A uniform random player.
VR-MCCFR	A player based on a fixed strategy precomputed for each game by our offline VR-MCCFR candidate in 40 seconds.

Table 5.6: Description of players in the head-to-head comparison.

LP	RND	OOS IST	VR-MCCFR
CR	0.704 ± 0.046	-0.037 ± 0.048	-0.056 ± 0.047
RND		-0.873 ± 0.048	-0.854 ± 0.048
OOS IST			-0.006 ± 0.052
IIGS	RND	OOS IST	VR-MCCFR
CR	0.653 ± 0.010	-0.008 ± 0.013	-0.006 ± 0.013
RND		-0.547 ± 0.011	-0.548 ± 0.011
OOS IST			-0.011 ± 0.013
PAM	RND	OOS IST	VR-MCCFR
CR	2.564 ± 0.144	-0.268 ± 0.149	-0.615 ± 0.150
RND		-1.523 ± 0.147	-1.531 ± 0.146
OOS IST			0.028 ± 0.148
LTTT	RND	OOS IST	VR-MCCFR
CR	0.852 ± 0.005	0.022 ± 0.007	0.011 ± 0.007
RND		-0.750 ± 0.007	-0.757 ± 0.007
OOS IST			-0.009 ± 0.007

Table 5.7: Head-to-head performance of different players in larger games. The first number in each cell indicates mean payoff for the row player when playing against the column player in the given game. The second number is a 95% confidence interval.

6. Discussion

In the previous chapter we evaluated the performance of continual resolving and compared it to online solving. However we have seen that the performance of continual resolving can be disappointing and often worse than that of online solving, which is a much simpler algorithm. In this chapter we examine this in greater detail.

One of the obvious disadvantages of our implementation of continual resolving compared to online solving is that it cannot use the pre-play time as efficiently as online solving. Online solving is able to improve the strategy for the whole game directly during the pre-play time, whereas in continual resolving we throw the trunk strategy away when we enter a new subgame. In this way the improvement to the trunk strategy can only indirectly improve the subgame strategies by creating better estimates of the opponent's counterfactual values required for the construction of the CFR-D gadgets.

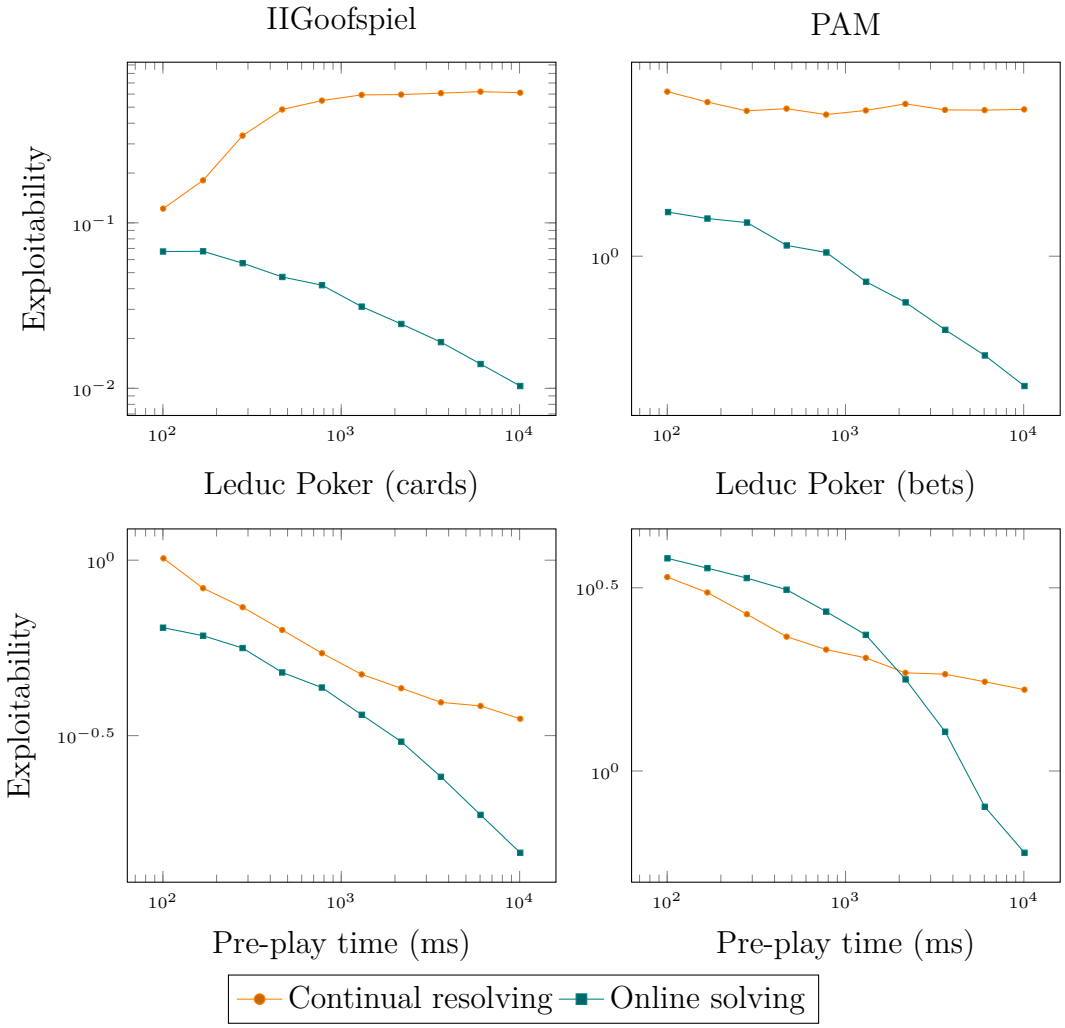

Figure 6.1: Comparison of impact of pre-play time on exploitability of continual resolving and online solving.

Figure 6.1 shows how increasing the pre-play time affects the exploitability of the overall strategy for both continual resolving and online solving when the

action time remains fixed to 300 ms. Both the continual resolving and the online solving players use our VR-MCCFR candidate from the offline evaluation round. The evaluation is done in the games from Table 5.4 using traversing evaluator.

While the online solving player reliably improves with more pre-play time, the continual resolving player only really improves in the two Leduc poker games. This could be because the estimates of the opponent's counterfactual values for the subgame gadgets have already converged (probably PAM), or because of some more complex interaction between an improving trunk strategy and a subgame strategy that remains similar regardless of the improvement to the trunk strategy.

Another aspect to the player's performance is if the extra time used for each action selection actually helps compared to just constructing a strategy for the whole game during the pre-play time and the time the for the first action selection (we refer to this as the first action strategy). We would most definitely want player's overall strategy to improve with each additional action selection (even though the benefit of the later action selections may be smaller because they can no longer change the strategy in the previous steps).

We compared the overall strategies produced by continual resolving and online solving separately against their first action strategies. It can be done because our implementation of continual resolving always uses the whole game to compute a strategy for the first action, even if it could use a smaller subgame (e.g. when it is acting second). We do this, because otherwise the pre-play time would be of very limited value to the continual resolving player.

Both the continual resolving player and the online solving players are our VR-MCCFR based candidates from the online evaluation round. The evaluation is done in the games from Table 5.4 using traversing evaluator with pre-play time of 300 ms. Given the small size of the games, this is a sort of worst-case scenario for this comparison, because both players are able to construct a first act strategy for the whole game. In larger games where this would not be possible with the given time constraints, the additional action selection steps would likely have more benefit.

Figures 6.2 and 6.3 compare the exploitabilities of the first action strategy and the final strategy for continual resolving and online solving respectively. We can see that neither player is making consistent improvements to their first action strategy, however the continual resolving player seems to be almost consistently bad in this regard.

We suspect that this could be at least partially caused by the CFR-D gadget. While it guarantees that combining a Nash equilibrium strategy for the subgame gadget with the resolving player's trunk strategy will not increase the exploitability for the resolving player (provided that the assumptions made by CFR-D are met), this may not be the case when the subgame strategy is not a Nash equilibrium. Furthermore, we do not use a counterfactual best response (CBR) strategy for the opponent when calculating their counterfactual values for the subgame gadget (which is required by CFR-D). We instead use their average strategy from the trunk and instead of the exact counterfactual values for the average strategy, we use the average of the counterfactual values from each iteration.

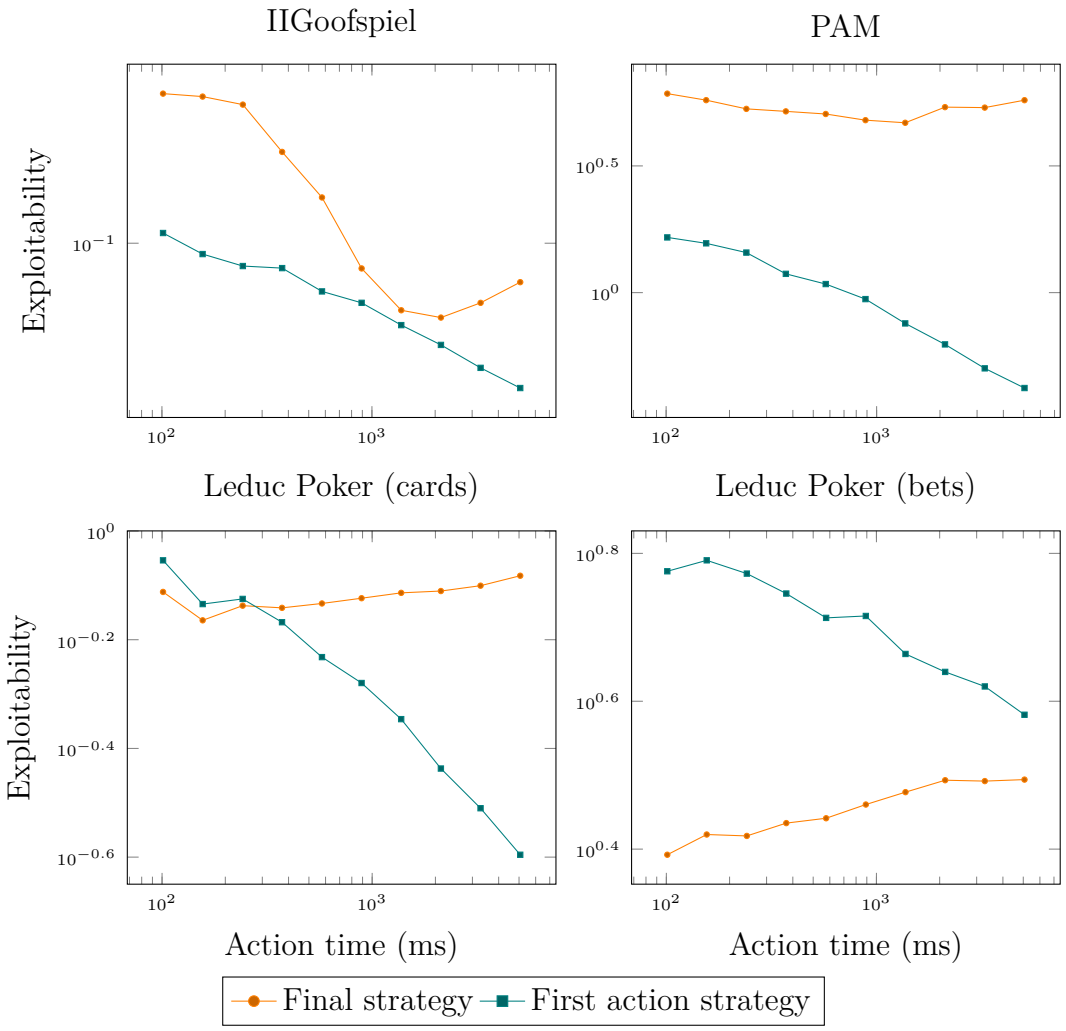

Figure 6.2: Comparison of exploitability of first action's strategy and final strategy for continual resolving.

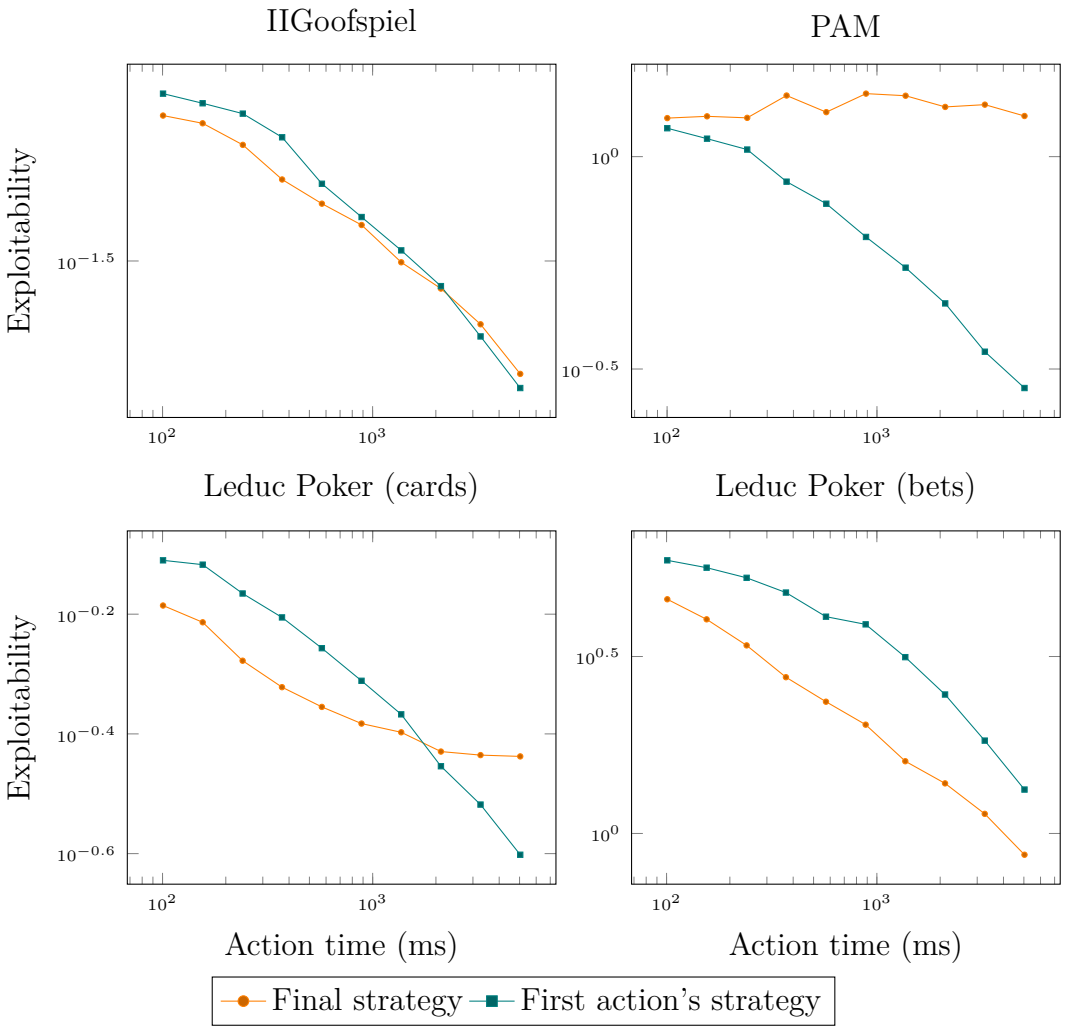

Figure 6.3: Comparison of exploitability of first action's strategy and final strategy for online solving.

To illustrate possible behaviors of CFR-D gadgets in different subgames, we evaluated a single step of CFR-D in a given subgame with a fixed trunk strategy. The trunk strategies were precomputed separately and used to compute the values necessary for the CFR-D gadget. The opponent's counterfactual values for the CFR-D gadget were computed either using the counterfactual best response (these results are marked with CBR), or just using the opponent's strategy in the trunk. The combined strategy was then created by taking the trunk strategy, and replacing strategies for each of the resolving player's information set inside subgame with the corresponding subgame strategy. The opponent's strategy remains the same as in the trunk.

We use CFR$^+$ in these experiments to avoid the non-determinism of MC-CFR. Apart from the exploitability plots for the combined strategies, we also include exploitability plots of the subgame strategies in the corresponding subgame gadget. A subgame is identified by a sequence of actions leading from the root of the game to one of the root states in the subgame.[1] The resolving player is the acting player in that state.

Figure 6.4 shows that the combined strategy may not properly converge to the trunk when the opponent's counterfactual values are not computed from the opponent's CBR strategy. The results were measured in latent tic-tac-toe, with subgame given by the following sequence of actions: `Mark(0,0)`, `Mark(0,0)`, `Mark(0,1)`. The trunk strategy was computed using CFR$^+$ in 23 iterations (that is approximately 10^8 visited states) with exploitability equal to 0.0969.

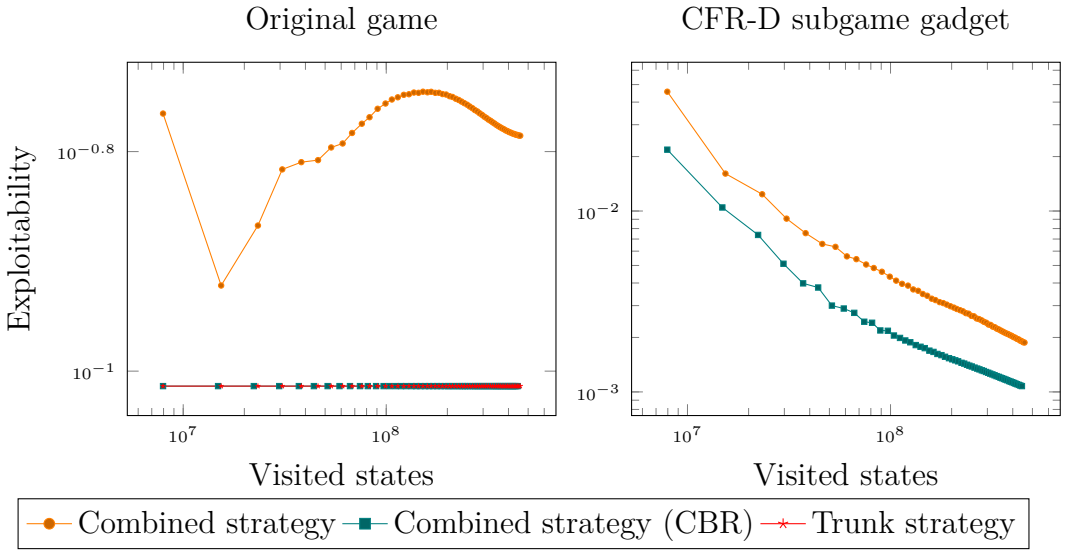

Figure 6.4: Example of a subgame where the combined strategy does not properly converge with approximate counterfactual values in the subgame gadget.

We can see that while both subgame gadgets converge at a similar rate, the subgame gadget constructed using CBR recovers the trunk strategy early, while the subgame gadget which uses just the opponent's trunk strategy does not seem to be able to recover the trunk strategy.

However, as shown in Figure 6.5, this does not always have to be the case.

[1] While the command-line interface expects the sequence to be encoded as an array of action indices, we report them here as pseudo-code for better readability.

This result is again from latent tic-tac-toe, but this time the subgame is given by the sequence: `Mark(0,0)`, `Mark(0,0)`. We used the same trunk strategy as in the previous case. We can see that in this larger subgame the combined strategy manages to improve the trunk strategy even when not using the CBR values.

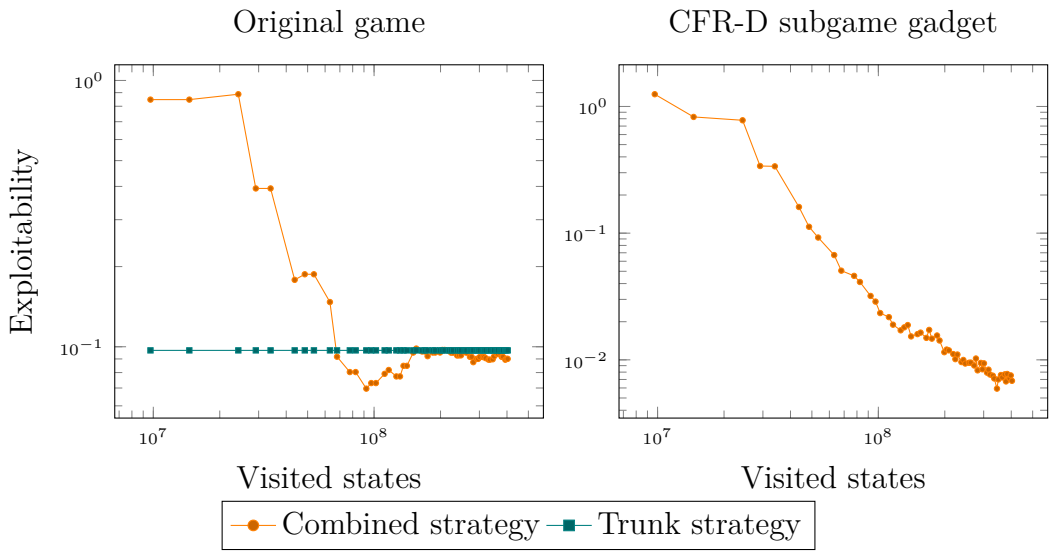

Figure 6.5: Example of a subgame where the combined strategy improves the trunk strategy.

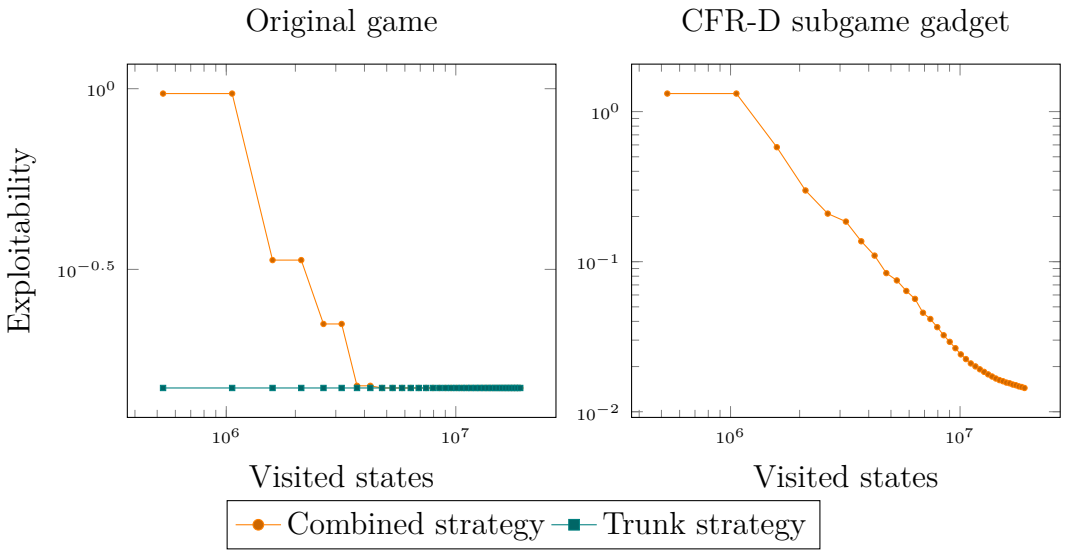

Figure 6.6: Example of a subgame where the combined strategy quickly recovers the trunk strategy, but does not improve it.

Figure 6.6 shows a result where the trunk strategy is recovered very quickly (in just 9 iterations of CFR$^+$ without even using CBR values). Even more interestingly it is the same subgame as in Figure 6.4, but with a different trunk strategy. Here we used a strategy which we computed using VR-MCCFR for the tournament in the previous chapter. The strategy was computed in about 4.4 million iterations of VR-MCCFR (or $3.2 \cdot 10^7$ visited states) and it's exploitabil-

ity is 0.1485. This result suggest that even with similar exploitabilities, some strategies may be easier to resolve than others.

Figure 6.7 shows a result where the combined strategy's exploitability does not seem to converge to the exploitability of the trunk strategy even when using CBR values. However since the combined strategy's exploitability cannot be higher that the trunk strategy's exploitability when the subgame strategy reaches a Nash equilibrium and CBR is used, we can only conclude that it would likely take a long time. The result is measured in II-Goofspiel with 5 cards. The subgame is given by this sequence of actions: `Bet(0)`, `Bet(0)`. The trunk strategy was computed using CFR$^+$ in 420 iterations (or $2.3 \cdot 10^7$ visited states) and it's exploitability is 0.0289.

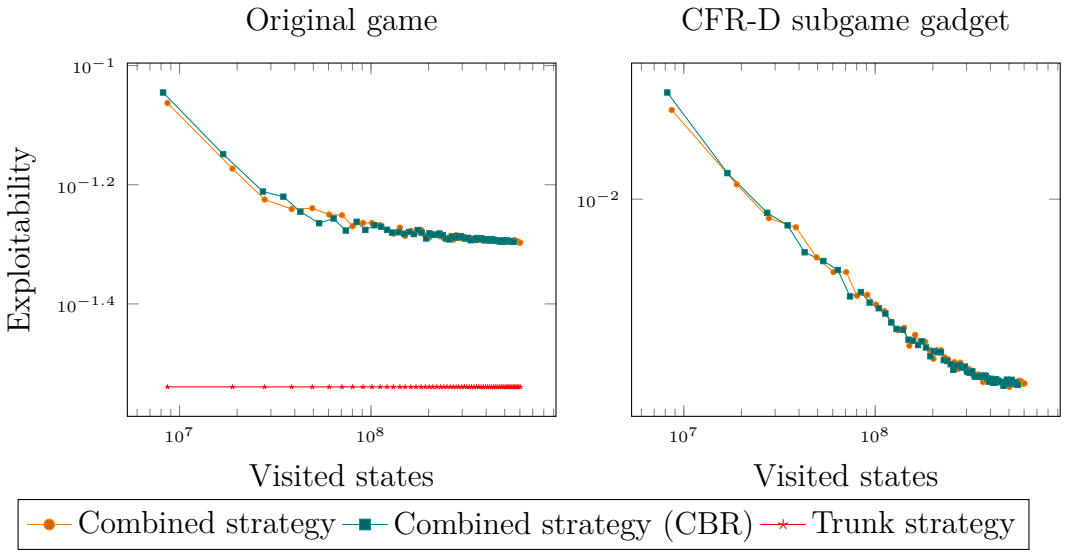

Figure 6.7: Example of a subgame where the combined strategy improves over time, but it is not able to recover the trunk strategy.

Finally, we wanted to see the impact of randomly noised counterfactual values in the subgame gadget on the exploitability of the combined strategy. First we computed the standard opponent's counterfactual values (using CBR) for the CFR-D subgame gadget. Then each of these values was multiplied by a factor drawn independently for each counterfactual value from $\mathcal{N}(1, \sigma)$ for a given standard deviation σ. We repeated this for 50 iterations (each iteration starting with the original counterfactual values) for each value of σ. The relative noise allows us to use the same standard deviation for all the games, even though their utilities can be quite different.

We again used CFR$^+$ to resolve the subgames. The evaluation was done on games from Table 5.1. We used the strategies we precomputed for the tournament in the previous section as trunk strategies. The action sequences specifying the subgames are in Table 6.1.

The results in Figure 6.8 are as expected – more precise counterfactual values lead to better performance. As the standard deviation of the noise gets higher, the convergence slows down and in extreme cases the exploitability of the combined strategy might converge to a higher value than that of the trunk strategy. It is also noteworthy that even without the noise, the convergence in princess and monster is slow.

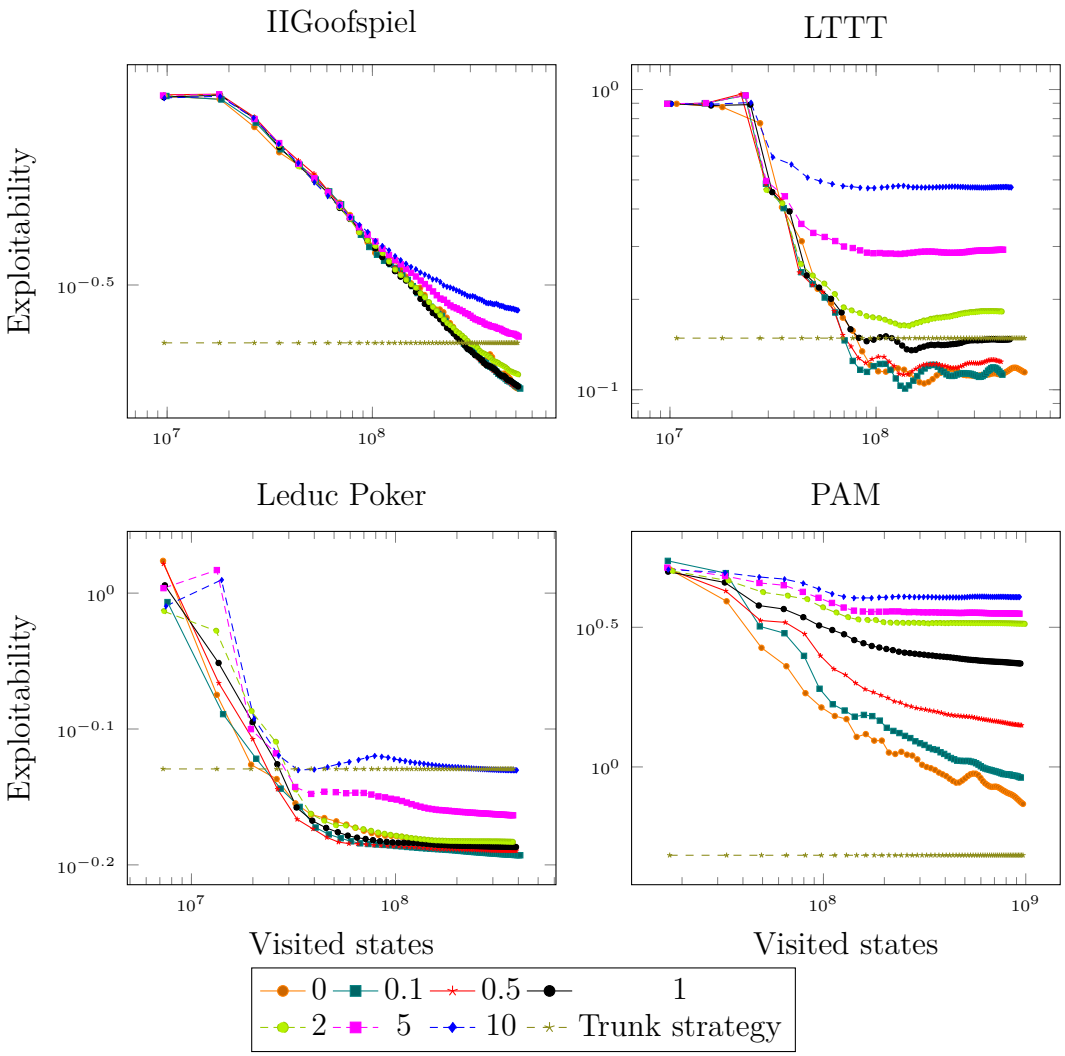

Figure 6.8: Comparison of how randomly noised counterfactual values in CFR-D gadget affect exploitability of the combined strategy.

Game	Subgame specification
IIGoofspiel	`Bet(0)`, `Bet(0)`
Latent Tic-Tac-Toe	`Mark(0)`, `Mark(0)`
Leduc Poker	`Deal(0,1)`, `Deal(0,2)`, `Call()`, `Raise()`
Princess and Monster	`Move(1,0)`, `Move(1,2)`

Table 6.1: Table of subgame specifications for the evaluation of the impact of randomly noised counterfactual values on the exploitability of resulting combined strategies.

7. Related work

The most closely related work is Monte Carlo Continual Resolving for Online Strategy Computation in Imperfect Information Games [Sustr et al., 2018]. It adapts continual resolving for general imperfect information games and uses MCCFR as the underlying solver. Overall it is similar to our approach with some minor differences in the implementation of continual resolving (eg. it uses a weighted average to estimate opponent's counterfactual values for the CFR-D gadget).

Additionally, it provides a theoretical analysis of the algorithm. For us the most interesting part is the probabilistic upper bound on exploitability of continual resolving's strategy quoted in Theorem 1. Each additional resolving adds a non-negative member to the upper bound on overall exploitability. Our own results from the previous chapter [Figure 6.2], where we compare the exploitability of the first action's strategy to the overall strategy produced by continual resolving, point to a similar behavior.

Theorem 1 (Sustr et al. [2018])**.** *With probability at least* $(1-p)^{N+1}$*, the exploitability of strategy* σ *computed by MCCR satisfies*

$$expl_i(\sigma) \leq \left(\sqrt{2}/\sqrt{p}+1\right)|\mathcal{I}_i|\frac{\Delta_{u,i}\sqrt{A_i}}{\delta}\left(\frac{2}{\sqrt{T_0}}+\frac{2N-1}{\sqrt{T_R}}\right),$$

where T_0 *and* T_R *are the numbers of MCCR's iterations in pre-play and each resolving,* N *is the number of required resolvings,* $\delta = \min_{z,t} q_t(z)$ *where* $q_t(z)$ *is the probabilty of sampling* $z \in Z$ *at iteration* t*,* $\Delta_{u,i} = \max_{z,z'} |u_i(z) - u_i(z')|$ *and* $A_i = \max_{I \in \mathcal{I}_i} |A(I)|$*.*

Another algorithm for online gameplay of imperfect information games is Information Set Monte Carlo Tree Search (ISMCTS) [Cowling et al., 2012]. It adapts Monte Carlo tree search to imperfect information games by searching a tree of information sets instead of a game tree. However, unlike continual resolving, this algorithm is not guaranteed to find a Nash equilibrium strategy.

Finally, there is the Hyperplay algorithm [Schofield et al., 2012], which is more focused on general game-playing. It maintains a collection of perfect information models of the game. It then uses a perfect information algorithm on those models, combines their decisions to obtain an action selection for the game and updates the model collection. It was further improved by HyperPlay-II [Schofield et al., 2013], which calculates expected payoff of a strategy in the imperfect information game, instead of a perfect information sample. However, even the improved version is short-sighted when valuing information [Chitizadeh and Thielscher, 2018].

Conclusion

In this thesis we presented a generalized version of the continual resolving algorithm from Deepstack [Moravčík et al., 2017]. We evaluated its performance across different games and compared it to the performance of online solving. However, continual resolving only seems to be better than online solving in games with narrow subgames (like poker). As the subgames grow larger, the advantage of continual resolving diminishes. Furthermore, online solving would likely perform better in games with narrow subgames if public subgame targeting was used.

The main advantage of continual resolving seems to be in the original Deepstack use-case – playing a large game by resolving strategy for each step using a pre-computed evaluation function (which can be compactly represented by a neural network) and depth-limited CFR. On the contrary, CFR and MC-CFR would have to store a strategy for the whole game, which makes them impractical for large games. This is however not as useful in general game playing because it is difficult to create a suitable evaluation function for a general game.

As a side result, we have also compared the performance of various CFR and MC-CFR variants, and demonstrated possible convergence issues with the CFR-D gadget.

As part of the thesis, we developed CFRPlayground – a Java application which provides tools for evaluating CFR-based solvers and players in general games. This application is open source and could be a useful starting point for evaluating other CFR-based algorithms in both online and offline setting.

Future work could be focused on improving the CFR-D gadget (or trying other gadgets, as was already suggested by [Moravčík et al., 2017]). Another possibility is to explore other sampling schemes for the underlying VR-MCCFR solver to try to improve the long-term convergence of outcome sampling (and its variance), while maintaining its early convergence.

Bibliography

Antlr. `https://www.antlr.org/`. Accessed: 2019-04-15.

Gradle build tool. `https://gradle.org/`. Accessed: 2019-04-12.

picocli - a mighty tiny command line interface. `https://picocli.info/`. Accessed: 2019-04-12.

Noam Brown and Tuomas Sandholm. Solving imperfect-information games via discounted regret minimization. *CoRR*, abs/1809.04040, 2018. URL `http://arxiv.org/abs/1809.04040`.

Neil Burch, Michael Johanson, and Michael Bowling. Solving imperfect information games using decomposition. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence*, AAAI'14, pages 602–608. AAAI Press, 2014.

Murray Campbell, A Joseph Hoane Jr, and Feng-hsiung Hsu. Deep blue. *Artificial intelligence*, 134(1-2):57–83, 2002.

Armin Chitizadeh and Michael Thielscher. Iterative tree search in general game playing with incomplete information. In *Computer Games Workshop at IJCAI*, 2018.

Peter I Cowling, Edward J Powley, and Daniel Whitehouse. Information set monte carlo tree search. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2):120–143, 2012.

Michael Genesereth and Michael Thielscher. General game playing. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 8(2):1–229, 2014.

Richard Gibson, Neil Burch, Marc Lanctot, and Duane Szafron. Efficient monte carlo counterfactual regret minimization in games with many player actions. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'12, pages 1880–1888, USA, 2012. Curran Associates Inc.

Martin J. Osborne and Ariel Rubinstein. *A course in Game Theory*. January 1994. ISBN 0-262-65040-1.

Marc Lanctot. *Monte Carlo Sampling and Regret Minimization for Equilibrium Computation and Decision-Making in Large Extensive Form Games*. PhD thesis, University of Alberta, 2013.

Marc Lanctot, Kevin Waugh, Martin Zinkevich, and Michael Bowling. Monte carlo sampling for regret minimization in extensive games. In *Proceedings of the 22Nd International Conference on Neural Information Processing Systems*, NIPS'09, pages 1078–1086, USA, 2009. Curran Associates Inc. ISBN 978-1-61567-911-9.

Viliam Lisý, Marc Lanctot, and Michael Bowling. Online monte carlo counterfactual regret minimization for search in imperfect information games. In *Proceedings of the 2015 International Conference on Autonomous Agents and Multiagent Systems*, AAMAS '15, pages 27–36, Richland, SC, 2015. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 978-1-4503-3413-6.

Nathaniel Love, Timothy Hinrichs, David Haley, Eric Schkufza, and Michael Genesereth. General game playing: Game description language specification. 2008.

Matej Moravčík, Martin Schmid, Neil Burch, Viliam Lisý, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael Bowling. Deepstack: Expert-level artificial intelligence in no-limit poker. *Science*, 356, 01 2017. doi: 10.1126/science.aam6960.

Jonathan Schaeffer, Robert Lake, Paul Lu, and Martin Bryant. Chinook the world man-machine checkers champion. *AI Magazine*, 17(1):21–21, 1996.

Martin Schmid, Neil Burch, Marc Lanctot, Matej Moravcik, Rudolf Kadlec, and Michael Bowling. Variance reduction in monte carlo counterfactual regret minimization (VR-MCCFR) for extensive form games using baselines. *CoRR*, abs/1809.03057, 2018. URL `http://arxiv.org/abs/1809.03057`.

Michael Schofield, Timothy Cerexhe, and Michael Thielscher. Hyperplay: A solution to general game playing with imperfect information. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*, AAAI'12, pages 1606–1612. AAAI Press, 2012.

Michael Schofield, Timothy Cerexhe, and Michael Thielscher. Lifting hyperplay for general game playing to incomplete-information models. In *Proc. GIGA 2013 Workshop*, pages 39–45, 2013.

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, 2016.

Michal Sustr, Vojtech Kovarík, and Viliam Lisý. Monte carlo continual resolving for online strategy computation in imperfect information games. *CoRR*, abs/1812.07351, 2018. URL `http://arxiv.org/abs/1812.07351`.

Oskari Tammelin, Neil Burch, Michael Johanson, and Michael Bowling. Solving heads-up limit texas hold'em. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, pages 645–652. AAAI Press, 2015. ISBN 978-1-57735-738-4.

Ole Tange. *GNU Parallel 2018*. Ole Tange, March 2018. ISBN 9781387509881. doi: 10.5281/zenodo.1146014. URL `https://doi.org/10.5281/zenodo.1146014`.

Michael Thielscher. A general game description language for incomplete information games. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*, 2010.

Martin Zinkevich, Michael Johanson, Michael H. Bowling, and Carmelo Piccione. Regret minimization in games with incomplete information. Technical Report TR07-14, University of Alberta, September 2007.

List of Figures

List of Tables

A. Appendix

A.1 Attached files

The thesis comes with files attached to it. They are structured as follows:

- `/bin/` – CFRPlayground's executables,
- `/data/` – aggregated experiment results used to draw plots in Chapter 5 and Chapter 6,
- `/docs/` – javadoc documentation for the source code,
- `/experiment-results/` – raw experiment results as produced by CFRPlayground, and job files with the corresponding CLI arguments,
- `/src/` – CFRPlayground's source code,
- `agg_results.py` – a python script which aggregates the raw results from CFRPlayground.

A.2 Experiment results data model

The experimental results are outputs of 4 different CFRPlayground's commands – `solve`, `evaluate`, `tournament`, and `cfrd-eval`. Each of these commands reports slightly different data. The final results are stored as CSV files in a directory structure, which captures some of the command's settings.

A.2.1 Solve command

The `solve` command is used to evaluate a solver in a game with time limit given in seconds and evaluation frequency in milliseconds. The results are stored in the following path relative to the specified results directory (brackets are replaced by corresponding parameters):

`/{game}/{solver}/{timeLimitS}-{evalFreqMs}-{dateKey}-{resultPostfix}.csv`

The game and solver parameters are replaced by the canonical configuration key of the selected game/solver. The result postfix can be used to add a comment to the result file (we used it to prevent file-name collision during the parallel evaluation).

The CSV file has the following columns:

- `intended_time` – time (ms) after which the solver should have been evaluated (a multiple of evaluation frequency),
- `time` – time (ms) after which the solver was actually evaluated,
- `iterations` – number of iterations the solver has done,
- `states` – number of states the solver has visited,

- `exp` – exploitability of the solver's average strategy,
- `avg_regret` – the solver's average regret.

A.2.2 Evaluate command

The `evaluate` command is used to evaluate a player in a game with given pre-play time and several time-limits for action selection (all in milliseconds). The result's format depends on the specified evaluator.

Traversing evaluator stores results directly in the following CSV file:

`/{game}/{player}/tr-{initMs}-{dateKey}-{resultPostfix}.csv`

Game-playing evaluator instead stores results as a serialized object, which includes the aggregated strategy. This allows the evaluation to be split into multiple jobs and then merged together using the `merge-gp-results` command. Finally, the merged results are converted to a CSV file using the `gp-to-csv` command, which stores in the following path:

`/{game}/{player}/gp-{initMs}-{intendedTimeMs}-{gameCount}-{dateKey}-conv.csv`

The CSV files from both of these evaluators have the following columns:

- `intended_time` – intended time limit for each action selection (ms),
- `time` – average action selection time (ms),
- `intended_init_time` – intended pre-play time (ms),
- `init_time` – average pre-play time,
- `states` – total number of states visited during the evaluation (depends on the evaluator, only relative comparison with the same evaluator is significant),
- `init_states` – average number of states visited during the pre-play time,
- `path_states` – average number of states visited during action selections for a single match (excluding pre-play time),
- `path_states_min` – minimum number of states visited during actions selections for a single match,
- `path_states_max` – maximum number of states visited during actions selections for a single match,
- `exp` – exploitability of the overall strategy,
- `first_act_exp` – exploitability of the first action strategy.

A.2.3 Tournament command

The `tournament` command is used to run multiple matches of a game between two players with pre-play time and action selection time given in milliseconds. The results are stored in the following CSV file:

`/{game}/{initMs}-{timeLimit}-{dateKey}-{resultPostfix}.csv`

Each data row in the CSV file is a result of a single match. The CSV file contains the following columns:

- `player1` and `player2` – configuration keys for players 1 and 2,
- `intended_init_time` – intended pre-play time (ms),
- `intended_time` – intended action selection time (ms),
- `init1` and `init2` – actual pre-play times for players 1 and 2,
- `time_sum1` and `time_sum2` – sum of action selection times for players 1 and 2,
- `actions1` and `actions2` – number of action selections for players 1 and 2 in the match,
- `payoff1` and `payoff2` – payoffs for players 1 and 2 (so that the command works even for non-zero-sum games).

A.2.4 CFRD-Eval command

The `cfrd-eval` command is used to evaluate a single step of CFR-D. The results are stored in the following CSV file:

`/{game}-{subgame}/{solver}-{useCBR}-{cfvNoise}/{timeLimitS}-{evalFreqMs}-{dateKey}-{resultPostfix}.csv`

The CSV file contains the following columns:

- `intended_time` – time (ms) after which the solver should have been evaluated (a multiple of evaluation frequency),
- `time` – time (ms) after which the solver was actually evaluated,
- `iterations` – number of iterations the solver has done,
- `states` – number of states the solver has visited,
- `exp` – exploitability of the combined strategy in the whole game,
- `avg_regret` – the solver's average regret,
- `trunk_exp` – exploitability of the trunk strategy in the whole game,
- `subgame_exp` – exploitability of the solver's average strategy in the subgame gadget.

A.2.5 Aggregated data

To plot the data, we need to aggregate results from multiple runs. The results can be aggregated using the attached `agg_results.py` script. It requires python 3 with pandas and numpy libraries installed. The results are grouped by the directory structure (game and solver/player), and by intended time (and intended init time – if available), the values in the columns are replaced by the average of the grouped results. Columns for 5th and 95th percentile values are added for exploitability and several other attributes.

A.3 Building from source

To build CFRPlayground executable from source code you need to have appropriate versions of Java development kit (8+) and Gradle installed (4.9+). Then you can simply run the following command in the source directory:

```
$ ./gradlew installDist
```

It will automatically download the dependencies, build the executable and place it in the `/build/install/CFRPlayground/bin/` directory.

A.4 Usage examples

We will now go over some the most common usage scenarios which we used during the evaluation. The experiment results also contain job files with the CLI arguments used to generate them, which can provide additional help. The job files do not contain the executable name (in order to work with different paths). A simple way to execute the jobs using GNU parallel is like this:

```
$ parallel {} -q --res-postfix {#} ::: ./build/install/CFRPlayground/
    bin/CFRPlayground :::: jobs.txt
```

This will execute CFRPlayground for each line in `jobs.txt` and pass additional arguments – `-q` for quiet mode, and `--res-postfix` set to sequence number of the job (to prevent conflicts in names of the result files).

To get the list of available CLI commands run:

```
$ ./CFRPlayground help
```

The parameters of a command can be obtained by running:

```
$ ./CFRPlayground help COMMAND
```

Most commands also have non-primitive parameters – such as games and players/solvers. These are constructed from a string specification using one of the factories for the type requested by the command. Information about the available types and implementations is provided by the `config-help` command. Without any parameters this command will list available configurable types. The list of implementations for a given type can be obtained by adding `-t TYPE` parameter to the command. The `TYPE` can be a substring of the full type name – the command will simply list implementations for each matching type. Finally, each implementation can have multiple factories with different parameters. The list of available factories and their parameters for an implementation can be obtained by

running the `config-help` command with parameter `-i IMPLEMENTATION` where `IMPLEMENTATION` is the full implementation name. Note that configuration keys are case-sensitive.

Let us assume, that we want to run a single match. By using the `help` command with no parameters, we find out, that this is done by the `run` command. We then run `help run` to obtain the list of parameters:

```
$ ./CFRPlayground help run
Usage: ./CFRPlayground run [-hV] --player1=<player1> --player2=<player2
    > -g=<game>
Runs given game with given players

Options:
  -g, --game=<game>       game to be played (IGameDescription)
      --player1=<player1> player 1 (IPlayerFactory)
      --player2=<player2> player 2 (IPlayerFactory)
  -i, --init=<init>       Init time (ms)
  -t, --time-limit=<timeLimit>
                          Time limit per move (ms)
  -h, --help              Show this help message and exit.
  -V, --version           Print version information and exit.
```

We can see that we need to specify game (configuration string for one of the implementations of `IGameDescription`) and two players (implementations of type `IPlayerFactory`). Using `config-help -t` we can list the available implementations for these two types, select the ones we want, and find their parameters using `config-help -i`. The final command can look like this:

```
$ ./CFRPlayground run -g "PerfectRecall{LeducPoker{7}}" --player1=
    RandomPlayer --player2="ContinualResolving{CFR{rm=RM+,cse=1}}" -i
    1000 -t 1000
```

Which runs a single match of perfect-recall Leduc poker where the first player is a uniform random player and the second player is a continual resolving player with CFR$^+$ solver. Note that depending on your shell, you may have to quote the configuration strings to prevent unwanted shell expansion.

A.5 Continual resolving with a depth-limit

When continual resolving is used together with depth-limited CFR, it must be ensured, that the depth-limit is sufficiently high, so that the player's next turn in another subgame is visited during the resolving. This is necessary to aggregate the values required by the CFR-D gadget.

However, no fixed depth-limit is sufficient for all possible games. To solve this issue, CFRPlayground includes a utility estimator wrapper (`ContinualResolvingUtilityEstimatorWrapper`), which ensures that the condition is met, before it allows the rest of the computation to be replaced by the underlying utility estimator. It can be used in a configuration string like this:

```
ContinualResolving{CFR{ue=CRUEW{RandomPlayout{1}},dl=0,rm=RM+,au=true,
    cse=1.0}}
```

Note that the depth-limit can now be 0. In that case the wrapper ensures, that the effective depth-limit is the minimal depth-limit required by continual resolving in the particular game.