Algorithms for Computing Strategies in Two-Player Simultaneous Move Games

Branislav Bošanský[1a], Viliam Lisý[a], Marc Lanctot[2b], Jiří Čermák[a], Mark H.M. Winands[b]

[a] *Agent Technology Center, Department of Computer Science, Faculty of Electrical Engineering, Czech Technical University in Prague*
Technicka 2, 166 27 Prague 6, Czech Republic
{branislav.bosansky,viliam.lisy,jiri.cermak}@agents.fel.cvut.cz
[1] *Corresponding author; phone: +420 22435 7581*
[b] *Games and AI Group, Department of Data Science and Knowledge Engineering, Maastricht University,*
P.O. Box 616, 6200 MD, Maastricht, The Netherlands
{marc.lanctot,m.winands}@maastrichtuniversity.nl

Abstract

Simultaneous move games model discrete, multistage interactions where at each stage players simultaneously choose their actions. At each stage, a player does not know what action the other player will take, but otherwise knows the full state of the game. This formalism has been used to express games in general game playing and can also model many discrete approximations of real-world scenarios. In this paper, we describe both novel and existing algorithms that compute strategies for the class of two-player zero-sum simultaneous move games. The algorithms include exact backward induction methods with efficient pruning, as well as Monte Carlo sampling algorithms. We evaluate the algorithms in two different settings: the offline case, where computational resources are abundant and closely approximating the optimal strategy is a priority, and the online search case, where computational resources are limited and acting quickly is necessary. We perform a thorough experimental evaluation on six substantially different games for both settings. For the exact algorithms, the results show that our pruning techniques for backward induction dramatically improve the computation time required by the previous exact algorithms. For the sampling algorithms, the

[2] This author has a new affiliation: Google DeepMind, London, United Kingdom.

results provide unique insights into their performance and identify favorable settings and domains for different sampling algorithms.

Keywords: simultaneous move games, Markov games, backward induction, Monte Carlo Tree Search, alpha-beta pruning, double-oracle algorithm, regret matching, counterfactual regret minimization, game playing, Nash equilibrium

1. Introduction

Strategic decision-making in multiagent environments is an important problem in artificial intelligence. With the growing number of agents interacting with humans and with each other, the need to understand these strategic interactions at a fundamental level is becoming increasingly important. Today, agent interactions occur in many diverse situations, such as e-commerce, social networking, and general-purpose robotics, each of which creates complex problems that arise from conflicting agent preferences.

Much research has been devoted to developing algorithms that reason about or learn in sequential (multi-step) interactions. As an example, adversarial search has been a central topic of artificial intelligence since the inception of the field itself, leading to very strong rational behaviors in Chess [1] and Checkers [2]. Advances in machine learning for multi-step interactions (e.g., reinforcement learning) have led to self-play learning of evaluation functions achieving master level play in Backgammon [3] and super-human level in Atari [4].

The most common model for these multistage environments is one with strictly sequential interactions. This model is sufficient in many settings [5], such as in the examples used above. On the other hand, it is not a good representation of the environment when agents are allowed to act simultaneously. These situations occur in many real-world scenarios such as auctions (e.g., [6]), autonomous driving, and many video and board games in the expanding gaming industry (e.g., [7, 8], including games we use for our experiments). In all of these scenarios, the simultaneity of the decision-making is crucial and we have to include it directly into the model when computing strategies. One of the fundamental differences of simultaneous move games versus strictly sequential games is that the agents may need to use randomized (or *mixed*) strategies in order to play optimally [9], i.e., to maximize their worst-case expected utility. This means that agents may need to randomize over several actions in some states of the game to guarantee the worst-case expected utility, even though the only information that is hidden is each player's action as they play it.

This paper focuses specifically on algorithms for decision-making in simultaneous move games. We cover the offline case, where the computation time is abundant and the optimal strategies are computed and stored, as well as the online case, where the computation time is limited and agents must choose an action quickly. We are concerned both with the quality of strategies based on their worst-case expected performance in theory and their observed performance in practice. We compare and contrast the algorithms and parameter choices in the offline and the online cases, and thoroughly evaluate each algorithm on a suite of games. Our collection covers Biased Rock-Paper-Scissors, Goofspiel, Oshi-Zumo, Pursuit-Evasion Games, and Tron, all of which have been used for benchmark purposes in previous work. We also perform experiments on randomly generated games. These games differ in the number of possible actions, the number of moves before the game ends, the variance of the utility values, and the proportion of states in which mixed strategies are required for optimal play.

Our experimental comparison shows that the algorithms perform differently in each case. The exact algorithms based on the backward induction are significantly better in the offline setting, where they are able to find the optimal strategy very quickly compared to the sampling algorithms. In some cases, our novel algorithm (DO$\alpha\beta$) solves the game in less than 2% of the time required by the standard backward induction algorithm. However, the exact algorithms are less competitive in the online setting. In contrast, the approximative sampling algorithms can perform very well in the online setting and find good strategies to play within a few seconds, however, they are not well-suited for offline solving of games.

The paper is structured as follows. First, we make explicit the contributions of the paper in Subsection 1.1. In Section 2, we present a formal introduction of the simultaneous move games that we will use throughout the paper. Section 3 follows with a list and discussion of the existing algorithms in the related work. In Section 4, we describe in detail selected exact and approximative algorithms. We first describe the algorithms in the offline setting, followed by the necessary modifications used in the online case described in Section 5. In Section 6, we present our experimental results comparing the algorithms. Finally, we conclude in Section 7.

1.1. Novel Contributions

This paper presents detailed descriptions and analysis of recent state-of-the-art exact [10] and approximative algorithms [11, 12, 13] that compute strategies for the class of two-player simultaneous move games. Furthermore, it presents the following original contributions:

- We present the latest variants of state-of-the-art algorithms under a single unified framework and combine the offline and online computation perspectives that have been previously analyzed separately.

- We describe the first adaptation of backward induction and the double-oracle algorithm with serialized bounds (DO$\alpha\beta$) [10] to the online search setting in simultaneous move games using iterative deepening and heuristic evaluation functions.

- We describe a novel variant of Online Outcome Sampling [13] tailored for two-player simultaneous move games (SM-OOS) and provide its formal analysis.

- We provide a wide experimental analysis and a comparison of these and other algorithms on five different specific games and on randomly generated games.

- We replicate an experimental convergence analysis for approximative algorithms that is often used in the literature as a demonstration that sampling-based algorithms are not guaranteed to converge to an optimal solution [14], and we identify the sensitivity of the existing approximative algorithms to tie-breaking rules.

Our algorithms thus allow computing offline strategies in larger games than previously possible (using DO$\alpha\beta$). In online game-playing, our algorithms are less sensitive to chosen parameters (SM-MCTS-RM) or guarantee to closely approximate the optimal strategies given enough time (SM-OOS). Since we describe each algorithm in a domain-independent manner, they can be further tailored to specific domains to achieve additional improvements in the scalability and/or game-playing performance.

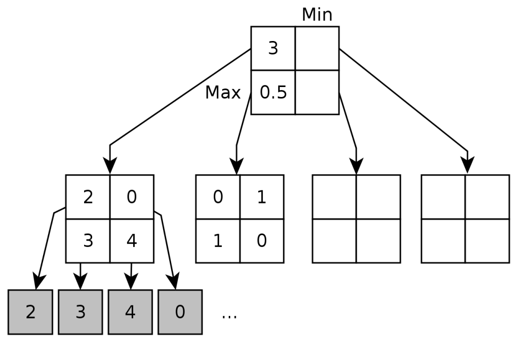


Figure 1: An example of a two-player simultaneous move game. Each white matrix corresponds to a state of the game where both players (a maximizing player with actions in rows and a minimizing player with actions in columns) act simultaneously. The dark squares are terminal states. The values shown in the matrices correspond to the values of subgames (e.g., calculated by backward induction).

2. Simultaneous Move Games

A finite two-player game with simultaneous moves and chance events (also called *Markov games*, or *stacked matrix games*) is a tuple $(\mathcal{N}, \mathcal{S}, \mathcal{A}, \mathcal{T}, \Delta_\star, u_i, s_0)$, where $\mathcal{S} = \mathcal{D} \cup \mathcal{C} \cup \mathcal{Z}$. The player set $\mathcal{N} = \{1, 2, \star\}$ contains player labels, where $\star$ denotes the chance player, and by convention a player is denoted $i \in \mathcal{N}$. $\mathcal{S}$ is a set of states, with $\mathcal{Z}$ denoting the terminal states, $\mathcal{D}$ the states where players make decisions, and $\mathcal{C}$ the possibly empty set of states where chance events occur. $\mathcal{A} = \mathcal{A}_1 \times \mathcal{A}_2$ is the set of joint actions of individual players. We denote by $\mathcal{A}_i(s)$ the actions available to player i in state $s \in \mathcal{S}$. The number of actions available to player i, $|\mathcal{A}_i(s)|$, is called the *branching factor for player i*. When the player is not specified, we mean the joint branching factor $|\mathcal{A}(s)|$. The transition function $\mathcal{T} : \mathcal{S} \times \mathcal{A}_1 \times \mathcal{A}_2 \mapsto \mathcal{S}$ is a partial function that defines the successor state given a current state and actions for both players. $\Delta_\star : \mathcal{C} \mapsto \Delta(\mathcal{S})$ describes a probability distribution over possible successor states of the chance event. Induced by $\Delta_\star$, we also define $\mathcal{P}_\star(s, r, c, s')$ as the probability of transitioning to s' after choosing joint action (r, c) from s, or simply 1 when $\mathcal{T}(s, r, c) \notin \mathcal{C}$. The utility function $u_i : \mathcal{Z} \mapsto [v_{\min}, v_{\max}] \subseteq \mathbb{R}$ gives the utility of player i, with v_{min} and $v_{\max}$ denoting the minimum and maximum possible utility respectively. We assume zero-sum games: $\forall z \in \mathcal{Z}, u_1(z) = -u_2(z)$. The game begins in an

	H	T
H	1	0
T	0	1

	A	B
a	0	1
b	-1	0

Figure 2: Matrix games of Matching Pennies (left), and one with a pure Nash equilibrium (right). Payoffs for the row player are shown.

initial state s_0 and a subset of a game that starts in some node s is called a *subgame*. An example of such a game is depicted in Figure 1, more examples can be found in [15, Chapter 5].

In two-player zero-sum games, a (subgame perfect) Nash equilibrium strategy is often considered to be optimal (the formal definition follows). It guarantees an expected payoff of at least V against any opponent. Any non-equilibrium strategy has its nemesis, which makes it gain less than V in expectation. Moreover, a subgame perfect Nash equilibrium strategy can earn more than V against weak opponents. After the opponent makes a sub-optimal move, the strategy will never allow it to gain the loss back. The value V is known as the *value of the game* and it is the same for every equilibrium strategy profile by von Neumann's minimax theorem [16].

A *matrix game* is a single step simultaneous move game with action sets $\mathcal{A}_1$ and $\mathcal{A}_2$ (see Figure 2). Each entry in the matrix A_{rc} where $(r, c) \in \mathcal{A}_1 \times \mathcal{A}_2$ corresponds to a utility value reached if row r is chosen by player 1 and column c by player 2. For example, in Matching Pennies in the left side of Figure 2, each player has two actions (heads or tails). The row player receives a payoff of 1 if both players choose the same action and 0 if they do not match. In simultaneous move games, at every decision state $s \in \mathcal{D}$ there is a joint action set $\mathcal{A}_1(s) \times \mathcal{A}_2(s)$. Each joint action (r, c) leads to another state $\mathcal{T}(s, r, c)$ that is either a terminal state or a subgame which is itself another simultaneous move game. A chance event is a state $s \in \mathcal{C}$ with a fixed set of outcomes, each of which leads to a possible successor state. In simultaneous move games, A_{rc} refers to the value of the subgame rooted in state $\mathcal{T}(s, r, c)$.

A *behavioral strategy* for player i is a mapping from states $s \in \mathcal{S}$ to a probability distribution over the actions $\mathcal{A}_i(s)$, denoted $\sigma_i(s)$. We denote by $\sigma_i(s, a)$ the probability that strategy σ_i assigns to a in s. These strategies are often called *randomized*, or *mixed* because they represent a mixture over *pure* strategies, each of which is a point in the Cartesian product space

$\prod_{s \in \mathcal{S}} \mathcal{A}_i(s)$.[1] Let $\mathcal{H}$ be a global set of histories (sequences of actions from the start of the game). Given a strategy profile $\sigma = (\sigma_1, \sigma_2)$, we define the probability of reaching a history h under σ as $\pi^\sigma(h) = \pi_1^\sigma(h)\pi_2^\sigma(h)\pi_\star^\sigma(h)$, where each $\pi_i^\sigma(h)$ is a product of probabilities of the actions taken by player i along the path to h ($\pi_\star$ being chance's probabilities). Finally, we define Σ_i to be the set of all behavioral strategies for player i. We adopt a standard convention that the index $-i$ refers to the opponent of player i.

In order to define optimal behavior for this class of games, we now provide definitions of some fundamental concepts.

Definition 2.1 (Strictly Dominated Action). *In a matrix game, an action $a_i \in \mathcal{A}_i$ is strictly dominated if $\forall a_{-i} \in \mathcal{A}_{-i}, \exists a'_i \in \mathcal{A}_i \setminus \{a_i\} : u_i(a_i, a_{-i}) < u_i(a'_i, a_{-i})$.*

No rational player would want to play a strictly dominated action, because there is always a better action to play independent of the opponent's action. The concept also extends naturally to behavioral strategies. For example, in the game on the right of Figure 2, both b and B are strictly dominated. In this paper we refer to the dominance always in this strict sense.

Definition 2.2 (Best Response). *Suppose $\sigma_{-i} \in \Sigma_{-i}$ is a fixed strategy of player $-i$. Define the set of best response strategies $BR_i(\sigma_{-i}) = \{\sigma_i \mid u_i(\sigma_i, \sigma_{-i}) = \max_{\sigma'_i \in \Sigma_i} u_i(\sigma'_i, \sigma_{-i})\}$. A single strategy in this set, e.g., $\sigma_i \in BR_i(\sigma_{-i})$, is called a best response strategy to σ_{-i}.*

Note that a best response can be a mixed strategy, but a pure best response always exists [9] and it is often easier to compute.

Definition 2.3 (Nash Equilibrium). *A strategy profile (σ_i, σ_{-i}) is a Nash equilibrium profile if and only if $\sigma_i \in BR_i(\sigma_{-i})$ and $\sigma_{-i} \in BR_{-i}(\sigma_i)$.*

In other words, in a Nash equilibrium profile each strategy is a best response to the opponent's strategy. In two-player zero-sum games, the set

[1] Notice that a pure strategy is also a mixed strategy that assigns probability 1 to a single pure strategy and probability 0 to every other pure strategy. However, as it is common in the literature, we sometimes refer to a mixed strategy to specifically mean not a pure strategy. This is mostly clear from the context, but we clarify where necessary.

of Nash equilibria corresponds to the set of minimax-optimal strategies. That is, a Nash equilibrium profile is also a pair of behavioral strategies optimizing

$$V = \max_{\sigma_1 \in \Sigma_1} \min_{\sigma_2 \in \Sigma_2} \mathbb{E}_{z \sim \sigma}[u_1(z)] = \max_{\sigma_1 \in \Sigma_1} \min_{\sigma_2 \in \Sigma_2} \sum_{z \in \mathcal{Z}} \pi^{\sigma}(z) u_1(z). \tag{1}$$

None of the players can improve their utility by deviating unilaterally. For example, the game of Rock, Paper, Scissors (depicted in Figure 3) modeled as a matrix game has a single state and the only equilibrium strategy is to mix equally between all actions, i.e., both players play with a mixed strategy $\sigma_i = \sigma_{-i} = (1/3, 1/3, 1/3)$ giving the expected payoff of $V = 0$. Note that using a mixed strategy is necessary in this game to achieve the guaranteed payoff of V. Any pure strategy of one player can be exploited by the opponent; so while a pure best response to a fixed strategy always exists, it is not always possible to find a Nash equilibrium for which both strategies are pure. For the same reason, randomized strategies are often necessary also in the multi-step simultaneous move games. If the strategies also optimize Equation (1) in every subgame, the equilibrium strategy is termed *subgame perfect.*

Finally, a two-player simultaneous move game is a specific type of two-player extensive-form game with imperfect information. In imperfect information games, states are grouped into *information sets*: two states s, s' are in an information set I if the player to act at I cannot distinguish whether she is in s or s'. Any simultaneous move game can be modeled using information sets to represent half-completed transitions, i.e., $\mathcal{T}(s, a_1, ?)$ or $\mathcal{T}(s, ?, a_2)$. The matrix game of Rock, Paper, Scissors can also be thought of as a two-step process where the first player commits to a choice, writing it on a face-down piece of paper, and then the second player responds. Figure 3 shows this transformation, which can generally be applied to every state in a simultaneous move game. Therefore, algorithms intended for two-player zero-sum imperfect information games may also be applied to the simultaneous move game using this equivalent form.

3. Related Work

There has been a number of algorithms designed for simultaneous move games. They can be classified into three categories: (1) iterative learning algorithms, (2) exact backward induction algorithms, (3) approximative sampling algorithms. The first type computes strategies through iterated self-play. The second type computes strategies in a game state recursively based

	r	p	s
R	0	−1	1
P	1	0	−1
S	−1	1	0

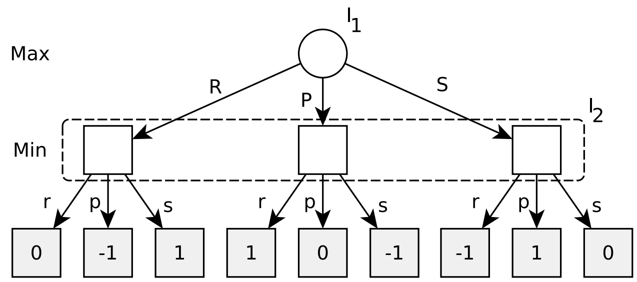


Figure 3: The matrix game of Rock, Paper, Scissors (left) and its equivalent extensive-form game representation (right). The extensive game has four states, two information sets (I_1 and I_2), and nine terminal histories: $\{Rr, Rp, Rs, Pr, Pp, Ps, Sr, Sp, Ss\}$.

on the values of its successors. The third type computes strategies by approximating utilities using sampling.

3.1. Iterative Learning Algorithms

A significant amount of interest in simultaneous move games was generated by initial work on multiagent reinforcement learning. In multiagent reinforcement learning, each agent acts simultaneously and the joint action determines how the state changes. Littman introduced Markov games to model these interactions as well as a variant of Q-learning called Minimax-Q to compute strategies [17, 18]. Minimax-Q modifies the learning rule so that the value of the next state (the subgame) is obtained by solving a linear program using the estimated values of that subgame's root. As it is common in these settings, the goal of each agent is to maximize their expected utility. In two-player zero-sum Markov games, an optimal policy corresponds to a Nash equilibrium strategy, which assures the agent the highest worst-case expected payoff. Initial results provided conditions under which approximate dynamic programming could be used to guarantee convergence to the optimal value function and policies [19]. Later, in [20], Lagoudakis and Parr provided stronger bounds and convergence guarantees for least squares temporal difference learning using linear function approximation. Bounds on the approximation error for sampling techniques in discounted Markov games are presented in [21], and new bounds for approximate dynamic programming have also been recently shown [22].

In early 2000s, gradient ascent methods were introduced for playing repeated games [23, 24]. These algorithms update strategies in a direction of the strategy space that increases the expected payoff with respect to the opponent's strategy. These were then generalized and combined, and shown to minimize regret over time [25, 26], leading to strong convergence guarantees in multiagent learning. More no-regret algorithms followed and were applied to imperfect information games in sequence-form (One-Card Poker) [27]. Later, counterfactual regret (CFR) minimization was introduced for large imperfect information games [28]. CFR has gained much attention due to its success in computing Poker AI strategies, and recently an application of CFR has solved Heads Up Limit Texas Hold'em Poker [29]. In this paper we analyze the effectiveness of a specific form of Monte Carlo CFR for the first time in simultaneous move games.

As we focus on zero-sum simultaneous move games in this paper, the work on multiagent learning in general-sum and cooperative games has been omitted. For surveys of the relevant previous work in multiagent reinforcement learning and game theory (including the zero-sum case), see [30, 31, 32].

3.2. Exact Backward Induction Algorithms

The techniques in this section are based on the backward induction algorithm (cf. [33]), a form of dynamic programming [34] often presented for purely sequential games. A modified variant of the algorithm can also be applied to simultaneous move games (e.g., see [35, 36, 37]). The algorithm enumerates states of the game tree in a depth-first manner and after computing the values of all the succeeding subgames of state $s \in \mathcal{S}$, it solves the normal-form game corresponding to s (i.e., computes a NE of the matrix game in s), and propagates the calculated game value to the predecessor. Backward induction then outputs a subgame perfect NE.

There are two notable algorithms that improve the standard backward induction in simultaneous move games. First is an algorithm by Saffidine et al. [38] termed simultaneous move alpha-beta algorithm (SMAB). The main idea of the algorithm is to reduce the number of the recursive calls of the backward induction algorithm by removing dominated actions in every state of the game. The algorithm keeps bounds on the utility value for each successor in a game state. The lower and upper bounds represent the threshold values, for which neither of the actions of the player is dominated by any other action in the current matrix game. These bounds are calculated by linear programs in the state given existing exact values (or appropriate bounds) of the utility values of all the other successors of the state. If they form an empty interval (the lower bound is greater than the upper bound), pruning takes place and the dominated action is no longer considered in this state afterward.

While SMAB outperforms classical backward induction, the speed-up is less significant in comparison to the second exact algorithm introduced in [10], a description of which is given in detail in Subsection 4.3.1. The main idea is to integrate two key components: (1) instead of evaluating all successors in each state of the game and solving a normal-form game, the algorithm exploits the iterative framework known in game theory as double-oracle algorithm [39]; (2) the algorithm computes bounds on the utility values of the successors by serializing the subgames and running the classic alpha-beta algorithm.

Finally, since simultaneous move games can be seen as extensive-form games with imperfect information, one can use techniques designed for large imperfect information games. An algorithm that is also built on double-oracle is the Range-of-Skill algorithm [40]. However, the number of iterations required by this algorithm in the worst case can be large [41]. There are also state-of-the-art algorithms for solving generic extensive-form games with imperfect information, based on sequence-form optimization problems [42, 43, 44]. However, these algorithms do not exploit the specific structure of simultaneous move games and could require memory that is linear in the size of the game tree. In practice, this prohibits scaling to larger games (see, e.g., [38]) and causes weak performance compared to tailored algorithms.

3.3. Approximative Sampling Algorithms

Monte Carlo Tree Search (MCTS) is a simulation-based state space search technique often used in extensive-form games [45, 46]. Having first seen practical success in computer Go [47, 48], MCTS has since been applied successfully to simultaneous move games and to imperfect information games [49, 50, 13]. Most of the successful applications use the Upper Confidence Bounds (UCB) formula [51] as a selection strategy. These variants of MCTS are also known as UCT (UCB applied to trees). The first application of MCTS to simultaneous move games was in general game playing (GGP) [52] programs: CadiaPlayer [53, 54] uses UCB selection strategy for each player in a single game tree. The success of MCTS was demonstrated by the success of CadiaPlayer which was the top-ranked player of the GGP competition between 2007 and 2009, and also in 2012.

Despite this success, Shafiei et al. in [14] provide a counter-example showing that this straightforward application of UCT does not converge to an equilibrium even in the simplest simultaneous move games and that a player playing a NE can exploit this strategy. Another variant of UCT, which has been applied to Tron [55], builds the tree as if the players were moving sequentially giving one of the players an informational advantage. This approach also cannot converge to an equilibrium in general. For this reason, other variants of MCTS were considered for simultaneous move games. Teytaud and Flory describe a search algorithm for games with short-term imperfect information [8], which are a generalization of simultaneous move games. Their algorithm uses a different selection strategy, called Exp3 [56], and was shown to work well in the Internet card game Urban Rivals. We provide details of these two main existing selection functions in Subsections 4.4.1 and 4.4.2. A more thorough experimental investigation of different selection policies including UCB, UCB1-Tuned, UCB1-greedy, Exp3, and more is reported in the game of Tron [57]. The work by Lanctot et al. [11] compares some of these variants and proposes Online Outcome Sampling, a search version of Monte Carlo CFR [58], which computes an approximate equilibrium strategy with high probability. We describe a new formulation of this algorithm in Subsection 4.5.1. Finally, [12, 59] present variants of MCTS that provably converge to Nash equilibria in simultaneous move games, using any regret-minimizing algorithm at each stage. We elaborate on these results in Subsection 4.4.4.

There have been two recent studies that examine the head-to-head performance of these variants in practice. The first [60] builds on previous work in Tron by varying the shape of the initial board, comparing previous serialized variants of simultaneous move MCTS. The authors found that UCB1-Tuned worked particularly well in Tron when using knowledge-based playout policies. The success of UCB1-Tuned differed in a similar study of the same variants across nine domains [61] without domain knowledge. In this work, the chosen games were ones inspired by previous work in general game playing and did not include chance elements. Results indicate that parameter-tuning landscapes do not seem as smooth as in the purely sequential case.

3.3.1. Simulation-Based Search in Real-time Games

Real-time games are not turn-based and represent realistic physical situations where agents can move freely in space. The state of the game is a continuous function of time and the effect of some actions may only be realized some time after the decision is made. These games are often appropriately modeled as a simultaneous move game with very short delays (e.g., 40 milliseconds) between frames.

MCTS has enjoyed some success in these types of games, in the single-agent setting [62, 63] and multiagent setting [64]. Much of this work is inspired by video games [65, 66, 67]. Few of these works have considered MCTS in the simultaneous move game directly. In one of the first papers on real-time strategy games, the authors used a randomized serialization of the game [68], or a strategy simulation from scripts was used to build a single matrix of values from which an equilibrium strategy was computed using linear programming [69]. This method can be extended to multiple nodes where internal nodes would correspond to scripts being interrupted to replan, similarly to [70]. MCTS-style multistage replanning was also applied to a real-time battle scenario which was also accurately represented as a discrete simultaneous move game [7]. Results of this work show that the multistage forward replanning can improve upon the single-stage forward planning, and can produce approximate Nash equilibrium strategies when mixed strategies are computed at each stage during the search. Around the same time, a serialized (sequential) version of the alpha-beta algorithm was proposed for simultaneous move games and run on combat scenarios [71]. This algorithm is described in greater detail in Subsection 4.2 as it forms the basis of the follow-up algorithm enhanced by double-oracle, presented in Subsection 4.3.

In this paper, we focus on the analysis of different algorithms for two-player simultaneous move games. Therefore, the problems arising from discrete modeling of continuous time and space remain outside the scope of this paper.

4. Offline Strategy Computation

This section focuses on algorithms that compute strategies for simultaneous move games. The baseline algorithm for solving simultaneous move games exactly is backward induction (BI) (Subsection 4.1). Afterwards we present a modification that exploits a fast computation of upper and lower bounds in a simultaneous move game (Subsection 4.2). Then, we further improve the algorithm by speeding up the computation of NE in matrix games, exploiting the iterative framework of double-oracle algorithms (Subsection 4.3). In Subsection 4.4 we describe Monte Carlo Tree Search for simultaneous move games. Finally, we present counterfactual regret minimization and its adaptation Online Outcome Sampling in Subsection 4.5.

input : s – current matrix game; i – searching player
1 **if** $s \in \mathcal{Z}$ **then**
2   **return** $u_i(s)$
3 **for** $r \in \mathcal{A}_1(s)$ **do**
4   **for** $c \in \mathcal{A}_2(s)$ **do**
5         $A_{rc} \leftarrow \sum_{s' \in \mathcal{S} \,:\, \mathcal{P}_\star(s,r,c,s')>0} \mathcal{P}_\star(s,r,c,s') \cdot \mathrm{BI}(s',i)$
6 $\langle v_s, \sigma_i(s)\rangle \leftarrow$ solve matrix game A
7 **return** v_s

Algorithm 1: Backward Induction algorithm (BI).

4.1. Backward Induction

The standard backward induction algorithm, first described for simultaneous move games in [35], enumerates the states in depth-first order. At each state of the game, it creates a matrix game for the current state using child subgame values, solves the matrix game, and propagates back the value of the matrix game. The pseudocode of the algorithm is given in Algorithm 1. If the successor node $\mathcal{T}(s,r,c)$ is a chance node, the algorithm directly evaluates all successors of this chance node and computes an expected utility: the value of each subgame rooted in node s' computed by the recursive call is weighted by the probability of the stochastic transition $\mathcal{P}_\star(s,r,c,s')$ (line 5).

Once the algorithm computes the value of each possible subgame following the current state s, matrix game A is well-defined and the algorithm solves matrix game A by solving the standard linear program (LP) for normal-form games[2]:

$$\begin{aligned}
\max \quad & v_s && && (2)\\
\text{s.t.} \quad & \textstyle\sum_{a_i \in \mathcal{A}_i} A_{a_i,a_{-i}} \cdot \sigma_i(s,a_i) \geq v_s && \forall a_{-i} \in \mathcal{A}_{-i}(s) && (3)\\
& \textstyle\sum_{a_i \in \mathcal{A}_i} \sigma_i(s,a_i) = 1 && && (4)\\
& \sigma_i(s,a_i) \geq 0 && \forall a_i \in \mathcal{A}_i(s) && (5)
\end{aligned}$$

A linear programming algorithm computes both the value v_s of the matrix game A, as well as the optimal strategy to play in this matrix game (variables $\sigma_i(s,a_i)$). Value v_s is then propagated to the predecessor (line 7 of Algorithm 1) and the optimal strategy $\sigma_i(s,a_i)$ is stored for this state. If the

[2]By solving a game we mean computing both the optimal value and the strategy that achieves it.

algorithm evaluates a terminal state, it directly returns the utility value of the state (line 1).

Evaluating each successor and solving an LP in each state of the game is the main computational bottleneck of the backward induction algorithm. The following algorithms try to prune some of the branches of the game tree in order to reduce this bottleneck even at the cost of multiple traversals of the game tree.

4.2. Backward Induction with Serialized Alpha-Beta Bounds

Solving computationally expensive linear programs in the backward induction algorithm is necessary in game states that require mixed strategies. However, many realistic games include subgames where it is sufficient to use only pure strategies. These subgames can be found efficiently by transforming the simultaneous move game into a perfect information extensive-form game with sequential moves and subsequently using some of the algorithms developed for this more standard setting. We call this purely alternating form a *serialization* of the original simultaneous move game. Consider a matrix game representing a single joint action of both players. This matrix can be serialized by discarding the notion of information sets; hence, letting one player play first, followed by the second player. The difference between a serialized and a simultaneous move matrix game is that the second player to move has an advantage of knowing what action the first player chose.

Given this advantage, the value of a serialized game consisting of a single simultaneous move where player i is second to move is greater than or equal to the value of the original simultaneous move game from the perspective of player i, formally shown by the following lemma.

Lemma 4.1. *Let A be a single step simultaneous move game for state s with value v_s for player i. Let v_s^i be the value of the serialized game created from game A by letting player $-i$ move first and player i move second with the knowledge of the move played by the first player. Then*

$$v_s \leq v_s^i.$$

Proof

$$
\begin{aligned}
v_s &= \min_{\sigma_{-i}\in\Sigma_{-i}} \max_{\sigma_i\in\Sigma_i} \sum_{a_i\in\mathcal{A}_i(s)} \sum_{a_{-i}\in\mathcal{A}_{-i}(s)} \sigma_i(s,a_i)\sigma_{-i}(s,a_{-i})A_{a_i a_{-i}} \\
&= \min_{a_{-i}\in\mathcal{A}_{-i}(s)} \max_{\sigma_i\in\Sigma_i} \sum_{a_i\in\mathcal{A}_i(s)} \sigma_i(s,a_i)A_{a_i a_{-i}} \\
&\le \min_{a_{-i}\in\mathcal{A}_{-i}(s)} \max_{a_i\in\mathcal{A}_i(s)} A_{a_i a_{-i}} = v_s^i.
\end{aligned}
$$

The first equality is the definition of the value of a zero-sum game. The second equality is from the fact that a best response can always be found in pure strategies: if there was a mixed strategy best response with expected utility v_s and some of the actions from its support would have lower expected utility, removing those actions from the support would increase the value of the best response, which is a contradiction. The inequality is due to the fact that a maximization over each action of player $-i$ can only increase the value. □

We can now generalize this lemma to game trees with multiple simultaneous moves.

Lemma 4.2. *Consider a simultaneous move subgame defined by state s and a serialized variant of this subgame, where in each state player i is second to move. The value of the serialized game is an upper bound on the value of the simultaneous move subgame for player i.*

Proof We use Lemma 4.1 inductively. Let s be the current state of the game and let A be the exact matrix game corresponding to s with utilities of player i. By induction we assume that the algorithm computes for state s some A' so that each value in matrix A' is greater than or equal to A:

$$\forall a_i \in \mathcal{A}_i(s) \forall a_{-i} \in \mathcal{A}_{-i}(s)\ A'_{a_i a_{-i}} \ge A_{a_i a_{-i}}.$$

Therefore, the value of matrix game $v_{A'} \ge v_A$. Finally, by Lemma 4.1 the algorithm returns value $v^i_{A'} \ge v_{A'} \ge v_A$. □

An example of this serialization is depicted in Figure 4. There is a simple matrix game for two players (the circle and the box player; the utility values are depicted for the circle player; the box player in the column is minimizing

	b_1	b_2
a_1	2	0
a_2	3	4

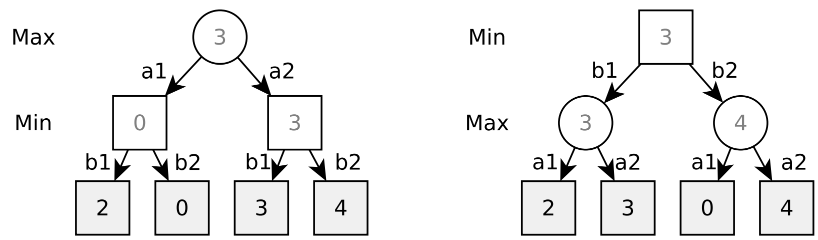


Figure 4: Different serializations of a simple simultaneous move game. Utility values are in the leaf nodes, the grey values correspond to the value propagation when solving the serialized game.

this value). There are two ways this game can be transformed into a serialized extensive-form game with perfect information. If the circle player moves first (the left game tree), then the value of this serialized game is the lower bound of the value of the game. If this player moves second (the right game tree), then the value of this serialized game is the upper bound of the value of the game. Since the serialized games are zero-sum perfect information games in the extensive form, they can be solved quite quickly by using some of the classic AI algorithms such as alpha-beta or negascout [72]. If the values of both serialized games are equal, then this value is also equal to the value of the original simultaneous move game. This situation occurs in our example in Figure 4, where both serialized games have value $V = 3$.

We can speed up the backward induction algorithm using bounds that are computed by the alpha-beta algorithm (denoted BI$\alpha\beta$). Algorithm 2 depicts the pseudocode. The BI$\alpha\beta$ algorithm first serializes the game and solves the serialized games using the standard alpha-beta algorithm; if the bounds are equal then this value is returned directly (line 3). Note that in Algorithm 2 the call alpha-beta(s,i), i is the second player to move in the serialized game rooted at s. If the bounds are not equal, the algorithm starts evaluating successors of the current state. As before, the algorithm computes upper and lower bounds using the alpha-beta algorithm on serialized variants of the subgame rooted at the successor s' (lines 9-10). Then, the algorithm uses the value directly if the bounds are equal (line 14), or performs a recursive

input : s – current matrix game; i – searching player

1 **if** $s \in \mathcal{Z}$ **then**

2   **return** $u_i(s)$

3 **if** (s *is root*) **and** ($alpha\text{-}beta(s,i) = alpha\text{-}beta(s,-i)$) **then**

4   **return** alpha-beta$(s,-i)$

5 **for** $r \in \mathcal{A}_1(s)$ **do**

6   **for** $c \in \mathcal{A}_2(s)$ **do**

7         $A_{rc} \leftarrow 0$

8   **for** $s' \in \mathcal{S} \ : \ \mathcal{P}_\star(s,r,c,s') > 0$ **do**

9             $v^i_{s'} \leftarrow$ alpha-beta(s', i)

10             $v^{-i}_{s'} \leftarrow$ alpha-beta$(s', -i)$

11   **if** $v^{-i}_{s'} < v^i_{s'}$ **then**

12                 $A_{rc} \leftarrow A_{rc} + \mathcal{P}_\star(s,r,c,s') \cdot \mathrm{BI}\alpha\beta(s', i)$

13   **else**

14                 $A_{rc} \leftarrow A_{rc} + \mathcal{P}_\star(s,r,c,s') \cdot v^i_{s'}$

15 $\langle v_s, \sigma_i \rangle \leftarrow$ solve matrix game A

16 **return** v_s

Algorithm 2: Backward Induction with Serialized Bounds (BI$\alpha\beta$).

call otherwise (line 12).

We distinguish two cases when extracting equilibrium strategies from the BI$\alpha\beta$ algorithm. In the first case, when a state is fully evaluated by the algorithm (i.e., an LP was built and solved for this state), we proceed as before and keep the pair of equilibrium strategies in this state. However, in the other case, the algorithms prunes certain branches and does not create an LP in some of the subgames. The algorithm then keeps the strategy computed by the serialized alpha-beta algorithm in those subgames. More precisely, for player i the algorithm keeps the pure strategy computed by alpha-beta$(s,\ -i)$, where the opponent has an advantage of knowing the moves of player i. Such a strategy provides a guarantees for player i (it is not exploitable) and due to the alpha-beta cut-offs we know that there is no better strategy for player i with a higher expected utility.

Theorem 4.3. *The algorithm BI$\alpha\beta(s,i)$ computes the value of the subgame from state s for player i.*

Proof The correctness of the algorithm follows immediately from the correctness of the standard BI algorithm and the correctness of using the values computed by serialized alpha-beta (Lemma 4.2). Moreover, values computed by the serialized alpha-beta algorithm are used only if the upper bound equals the lower bound. □

The performance of BI$\alpha\beta$ depends on the existence of a pure NE in the simultaneous move game. In the best case (i.e., there exists a pure NE), the algorithm finds the solution by solving each serialization exactly once starting from the root state. In the worst case, all NE require mixed strategies in every state of the game. In this case, the algorithm not only solves the LP in each state similarly to BI, but also repeatedly attempts to solve serialized subgames by calling the alpha-beta algorithm. However, this case was very rarely encountered during our experiments.

4.3. Backward Induction with Double-Oracle and Serialized Bounds

The computational complexity of solving a matrix game by linear programming can be reduced by their incremental construction using the iterative double-oracle algorithm [39]. The following algorithm incorporates this idea to BI$\alpha\beta$, which leads to additional pruning of the game tree. First of all, we describe the main principles of the double-oracle algorithm for matrix games, followed by the description of the integration of the double-oracle algorithm in simultaneous move games [10] (denoted DO$\alpha\beta$).

4.3.1. Double-Oracle Algorithm for Matrix Games

The goal of the double-oracle algorithm is to find a solution of a matrix game without necessarily constructing the complete LP that solves the game. The main idea is to create a restricted game where the players can choose only from a limited set of actions. The algorithm iteratively expands the restricted game by allowing the players to choose from new actions. The new actions are added incrementally: in each iteration, a best response (chosen from the unrestricted action set) to an optimal strategy of the opponent in the current restricted game, is added to restricted game.

Figure 5 shows a visualization of the main structure of the algorithm, where the following three steps repeat until convergence:

1. Create a restricted matrix game by limiting the set of actions that each player is allowed to play.
2. Compute a pair of Nash equilibrium strategies in this restricted game using linear programming.

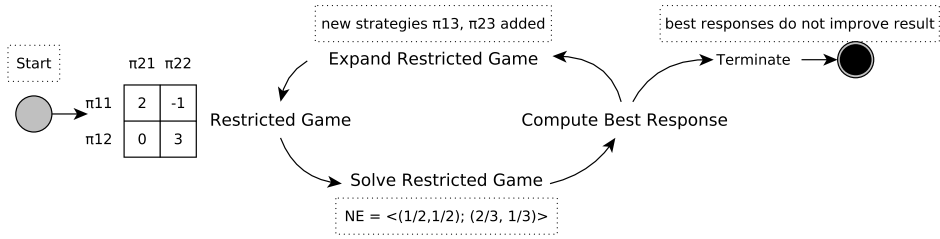


Figure 5: Schematic of the double-oracle algorithm for a normal-form game.

3. For each player, compute a pure best response strategy against the equilibrium strategy of the opponent; pure best response can be *any* action from the original unrestricted game.

The best response strategies computed in step 3 are added to the restricted game, the game matrix is expanded by adding new rows and columns, and the algorithm follows with the next iteration. The algorithm terminates if neither of the players can improve the outcome of the game by adding a new strategy to the restricted game; hence, both players play best response strategies to the strategy of the opponent. The algorithm maintains the values of the best expected utilities of the best-response strategies for each player throughout the iterations of the algorithm. These values provide bounds on the value of the original game V (from Equation 1), and their sum represents the error of the algorithm which converges to zero.

4.3.2. Integrating Double-Oracle with Backward Induction

The double-oracle algorithm for matrix games can be directly incorporated into the backward induction algorithm: instead of immediately evaluating each of the successors of the current game state and solving the linear program, the algorithm can exploit the double-oracle algorithm. Pseudocode in Algorithm 3 details this integration.

Similarly to BI$\alpha\beta$, the algorithm first tests, whether the whole game can be solved by using the serialized variants of the game (line 3). If not, then in each state of the game the algorithm initializes the restricted game with an arbitrary action (line 5)[3] – A' represents the restricted matrix game, $\mathcal{A}'_i$ represents the restricted set of available actions to player i. The algorithm

[3] In practice we use the first action of a shuffled ordered set $\mathcal{A}_i$ for each player i. This initialization step can be improved with domain knowledge and by adding more actions.

then starts the iterations of the double-oracle algorithm. First, the algorithm needs to compute the value for each of the successors of the restricted game, for which the current value is not known (lines 8-16). This evaluation is the same as in the case of BI$\alpha\beta$. Once all values for restricted game A' are known, the algorithm solves the restricted game and keeps the optimal strategies σ' of the restricted game (line 17). Next, the algorithm computes best responses for each of the player (lines 18,19) using Algorithm 4 below, and updates the lower and upper bounds (line 20). Finally, the algorithm expands the restricted game with the new best response actions (line 21) until the lower and upper bound are equal. Once the bounds are equal, neither of the best responses improves the current solution from the restricted game; hence, the algorithm has found an equilibrium of the complete unrestricted matrix game corresponding to state s.

Now we describe the algorithm for computing the best responses on lines 18 and 19. The pseudocode of this step is depicted in Algorithm 4. The goal of the best response algorithm is to find the best action from the original unrestricted game against the current strategy of the opponent σ'_{-i}. Throughout the algorithm we use, as before, $v^i_{s'}$ to denote the upper bound of the value of the subgame rooted in state s' computed using alpha-beta(s', i). These values are computed on demand, i.e., they are computed once needed and cached until the game for state s is solved. Moreover, once the algorithm computes the exact value of a particular subgame, both upper and lower bounds are updated to be equal to the exact value of the game.

The best response algorithm iteratively examines all actions of player i from the unrestricted game (line 3). Every action a_i is evaluated against the actions of the opponent that are used in the optimal strategy from the restricted game (line 5). Before evaluating the successors, the algorithm determines whether the current action a_i of the searching player i can still be the best response action against the strategy of the opponent σ'_{-i}. In order to determine this, the algorithm computes value λ_{a_i} that represents the lower bound on the expected utility this action must gain against the current action of the opponent a_{-i} in order for action a_i to be a best response. λ_{a_i} is calculated (line 7) by subtracting the upper bound of the expected value against all other actions of the opponent ($v^i_{\mathcal{T}(s,a_i,a'_{-i})}$) from the current best response value (v_i^{BR}) and normalizing with the probability that the action a_{-i} is played by the opponent ($\sigma'_{-i}(a_{-i})$). This calculation corresponds to a situation where player i achieves the best possible utility by playing action

input : s – current matrix game; i – searching player; α_s, β_s – bounds for the game value rooted in state s

if $s \in \mathcal{Z}$ **then**
    return $u_i(s)$
if (s *is root*) **and** ($\textit{alpha-beta}(s,i) = \textit{alpha-beta}(s,-i)$) **then**
    return alpha-beta$(s,-i)$
initialize $\mathcal{A}'_i$, $\mathcal{A}'_{-i}$ with arbitrary actions from $\mathcal{A}_i, \mathcal{A}_{-i}$
repeat
    for $r \in \mathcal{A}'_i$, $c \in \mathcal{A}'_{-i}$ **do**
        if A'_{rc} *is not initialized* **then**
            $A'_{rc} \leftarrow 0$
            for $s' \in \mathcal{S} \,:\, \mathcal{P}_\star(s,r,c,s') > 0$ **do**
                $v^i_{s'} \leftarrow$ alpha-beta(s',i)
                $v^{-i}_{s'} \leftarrow$ alpha-beta$(s',-i)$
                if $v^{-i}_{s'} < v^i_{s'}$ **then**
                    $A'_{rc} \leftarrow A'_{rc} + \mathcal{P}_\star(s,r,c,s') \cdot \mathrm{DO}\alpha\beta(s', i, v^{-i}_{s'}, v^i_{s'})$
                else
                    $A'_{rc} \leftarrow A'_{rc} + \mathcal{P}_\star(s,r,c,s') \cdot v^i_{s'}$
    $\langle v_s, \sigma' \rangle \leftarrow$ solve matrix game A'
    $\left\langle v^{BR}_i, a^{BR}_i \right\rangle \leftarrow \mathrm{BR}(s, i, \sigma'_{-i}, \beta_s)$
    $\left\langle v^{BR}_{-i}, a^{BR}_{-i} \right\rangle \leftarrow \mathrm{BR}(s, -i, \sigma'_i, -\alpha_s)$
    $\alpha_s \leftarrow \max(\alpha_s, -v^{BR}_{-i})$, $\beta_s \leftarrow \min(\beta_s, v^{BR}_i)$
    $\mathcal{A}'_i \leftarrow \mathcal{A}'_i \cup \{a^{BR}_i\}$, $\mathcal{A}'_{-i} \leftarrow \mathcal{A}'_{-i} \cup \{a^{BR}_{-i}\}$
until $\alpha_s = \beta_s$
return v_s

Algorithm 3: Double-Oracle with Serialized Bounds (DO$\alpha\beta$).

a_i against all other actions from the strategy of the opponent and it needs to achieve at least λ_{a_i} against a_{-i} so that the expected value for playing a_i is at least v^{BR}_i. If λ_{a_i} is strictly higher than the upper bound on the value of the subgame rooted in the successor (i.e., $v^i_{\mathcal{T}(s,a_i,a_{-i})}$) then the algorithm knows that the action a_i can never be the best response action, and can proceed with the next action (line 9). Note that λ_{a_i} is recalculated for each action of the opponent since the upper bound values can become tighter when the exact values are computed for successor nodes s' (line 13).

If the currently evaluated action a_i can still be a best response, the value of the successor is determined (first by comparing the bounds). Once the expected outcome against all actions of the opponent is known, the expected

input : s – current matrix game; i – best-response player; σ'_{-i} – strategy of the opponent; λ – bound for the best-response value

1 $v_i^{BR} \leftarrow \lambda$
2 $a_i^{BR} \leftarrow$ **null**
3 **for** $a_i \in \mathcal{A}_i$ **do**
4 $\quad v_{a_i} \leftarrow 0$
5 $\quad$ **for** $a_{-i} \in \mathcal{A}'_{-i} \ : \ \sigma'_{-i}(a_{-i}) > 0$ **do**
6 $\quad\quad v_{a_i,a_{-i}} \leftarrow 0$
7 $\quad\quad \lambda_{a_i} \leftarrow \frac{v_i^{BR} - \sum_{a'_{-i} \in \mathcal{A}'_{-i} \setminus \{a_{-i}\}} \sigma'_{-i}(a'_{-i}) \cdot v^i_{\mathcal{T}(s,a_i,a'_{-i})}}{\sigma'_{-i}(a_{-i})}$
8 $\quad\quad$ **if** $\lambda_{a_i} > v^i_{\mathcal{T}(s,a_i,a_{-i})}$ **then**
9 $\quad\quad\quad$ continue from line 3 with next a_i
10 $\quad\quad$ **else**
11 $\quad\quad\quad$ **for** $s' \in \mathcal{S} \ : \ \mathcal{P}_\star(s, a_i, a_{-i}, s') > 0$ **do**
12 $\quad\quad\quad\quad$ **if** $v_{s'}^{-i} < v_{s'}^{i}$ **then**
13 $\quad\quad\quad\quad\quad v_{a_i,a_{-i}} \leftarrow v_{a_i,a_{-i}} + \mathcal{P}_\star(s, a_i, a_{-i}, s') \cdot$ DO$\alpha\beta(s', i, v_{s'}^{-i}, v_{s'}^{i})$
14 $\quad\quad\quad\quad$ **else**
15 $\quad\quad\quad\quad\quad v_{a_i,a_{-i}} \leftarrow v_{a_i,a_{-i}} + \mathcal{P}_\star(s, a_i, a_{-i}, s') \cdot v_{s'}^{i}$
16 $\quad\quad\quad v_{a_i} \leftarrow v_{a_i} + \sigma'_{-i}(a_{-i}) \cdot v_{a_i,a_{-i}}$
17 $\quad$ **if** $v_{a_i} \geq v_i^{BR}$ **then**
18 $\quad\quad v_i^{BR} \leftarrow v_{a_i}$
19 $\quad\quad a_i^{BR} \leftarrow a_i$
20 **return** $\langle v_i^{BR}, a_i^{BR} \rangle$

Algorithm 4: Best Response with Serialized Bounds (BR)

value of action a_i is compared against the current best response value (line 17) and saved if the expected utility is higher (line 19). These best response actions are allowed in the next iteration of the double-oracle algorithm and the algorithm progresses further as described.

When extracting strategies from DO$\alpha\beta$, we proceed exactly as in the case of BI$\alpha\beta$: either a double-oracle is initialized and solved for a certain matrix game and we keep the equilibrium strategies from the final restricted game, or the strategy is extracted from the serialized alpha-beta algorithms as before.

Theorem 4.4. *The DO$\alpha\beta(s, i, \alpha_s, \beta_s)$ algorithm computes the value of the subgame defined by state s for player i.*

Proof The correctness of the algorithm follows from the correctness of the standard BI algorithm, Lemma 4.2, and the correctness of the double-oracle algorithm for matrix games [39]. We use them inductively for state s and assume DO$\alpha\beta$ for all the children of s returned correct values when called. Since we are using the classical double-oracle on a matrix game corresponding to state s with correct values, we only need to show that the best-response algorithm with serialized bounds cannot return **null** action due to setting the bounds incorrectly.

Without loss of generality, consider a lower bound $-\alpha_s$ for state s to be λ in the best response algorithm. Value λ thus corresponds either to a value calculated by serialized alpha-beta and propagated via bounds when calling DO$\alpha\beta(s, i, \alpha_s, \beta_s)$, or it was updated during the iterations of the double-oracle algorithm for state s (line 20). In either case there exists a pure best response strategy corresponding to this value; hence, the best response has to find the strategy that achieves this value and cannot return **null**. □

Similarly to BI$\alpha\beta$, the performance of DO$\alpha\beta$ also depends on the existence of a pure NE in the simultaneous move game. The best case is identical to BI$\alpha\beta$ and the algorithm finds the solution by solving each serialization exactly once starting from the root state. In the worst case, neither of the serialized games yield useful bounds and the algorithm needs to call the double-oracle algorithm in every state. Moreover, the worst case for the double-oracle algorithm occurs when all actions in this state must be added and an action for only a single player is added in each iteration causing the largest number of iterations repeatedly resolving the linear program. Again in practical games used for benchmark purposes, or in real-world applications this is rarely the case. Moreover, the computational overhead from repeatedly solving a LP is relatively small. This is due to the size of each LP that is determined by the number of actions in each state (the number of constraints and variables is bounded by the number of actions in each state). Therefore, the size of each LP is small compared to the number of states DO$\alpha\beta$ can prune out, especially if the pruning occurs close to the root of the game tree.

4.4. Simultaneous Move Monte Carlo Tree Search (SM-MCTS)

In the following subsections we move to the approximative algorithms. Monte Carlo Tree Search (MCTS) is a simulation-based state space search algorithm often used in game trees. In its simplest form, the tree is initially empty and a single leaf is added each iteration. Each iteration starts by

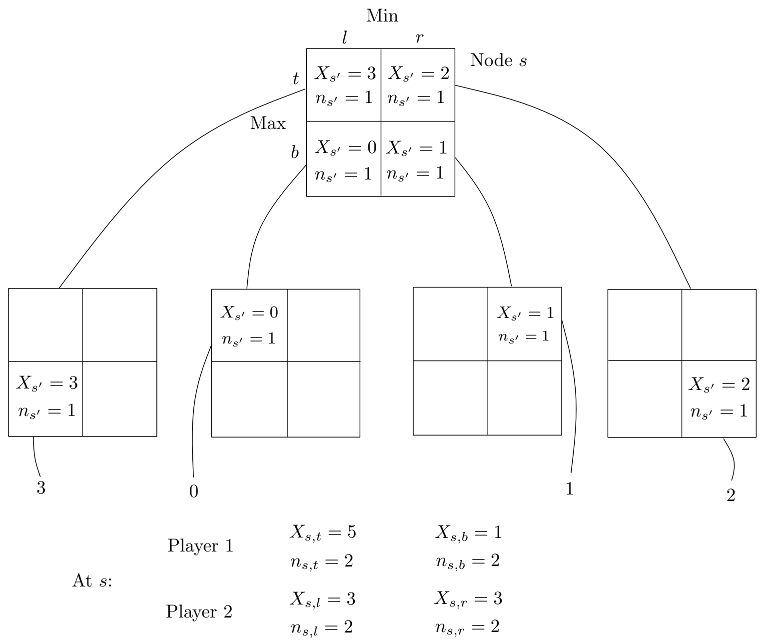


Figure 6: Simultaneous Move MCTS example. Here, $X_{s'}$ represents the cumulative payoff of all simulations that have passed through the cell, while $n_{s'}$ represents the number of simulations that have passed through the cell.

visiting nodes in the tree, selecting which actions to take based on a *selection function* and information maintained in the node. Consequently, the algorithm transitions to a successor state. When a node is visited whose immediate children are not all in the tree, the node is expanded by adding a new leaf to the tree. Then, *a rollout policy* (e.g., random action selection) is applied from the new leaf to a terminal state. The outcome of the simulation is then returned as a reward to the new leaf and the information stored in the tree is updated.

Consider again the game depicted in Figure 1. We demonstrate how Monte Carlo Tree Search could progress in this game using the example shown in Figure 6. This game has a root state, two subgames that are simple matrix games, and two arbitrarily large subgames. In the root state, player 1 (Max) has two actions: top (t) and bottom (b), and player 2 also has two actions: left (l) and right (r). The tree is initialized with a single empty state, s. On the first iteration, the first child corresponding to (t, l) is added to the tree, giving a payoff $u_1 = 3$ at the terminal state which is

input : s – current state of the game
1 **if** $s \in \mathcal{Z}$ **then**
2   **return** $u_1(s)$
3 **if** $s \in \mathcal{C}$ *is a chance node* **then**
4   Sample $s' \sim \Delta_\star(s)$
5   **return** SM-MCTS(s')
6 **if** s *is in the MCTS tree* **then**
7     $(a_1, a_2) \leftarrow \textsc{Select}(s)$
8     $s' \leftarrow \mathcal{T}(s, a_1, a_2)$
9     $v_{s'} \leftarrow$ SM-MCTS(s')
10     $\textsc{Update}(s, a_1, a_2, v_{s'})$
11   **return** $v_{s'}$
12 **else**
13   Add s as a new child in the MCTS tree
14     $v_s \leftarrow$ Rollout(s)
15   **return** v_s

Algorithm 5: Simultaneous Move Monte Carlo Tree Search (SM-MCTS)

backpropagated to each state visited on the simulation. Similarly, on the second iteration the second child corresponding to (b, l) is added to the tree, giving a payoff $u_1 = 1$, which is backpropagated up to all of its parents. After four simulations, every cell in the root state has a value estimate.

There are many possible ways to select actions based on the estimates stored in each cell which lead to different variants of the algorithm. We therefore first formally describe a generic template of MCTS algorithms for simultaneous move games (SM-MCTS) and then explain different instantiations derived from this template. Algorithm 5 describes a single iteration of SM-MCTS. The "MCTS tree" is an explicit tree data structure that stores the nodes of the search tree maintained in memory, e.g., the five-node tree shown in Figure 6. Every node s in the tree maintains algorithm-specific statistics about the iterations that previously visited this node. The template can be instantiated by specific implementations of the updates of the statistics on line 10 and the selection based on these statistics on line 7. In the terminal states, the algorithm returns the value of the state for the first player (line 2). At chance nodes, the algorithm samples one of the possible next states based on its distribution (line 4). If the current state has a node in the current MCTS tree, the statistics in the node are used to select an action for each player (line 7). These actions are executed (line 8) and the algorithm is called recursively on the resulting state (line 9). The result of this call is used to update the statistics maintained for state s (line 10). If the current state is not stored in the tree, it is added to the tree (line 13) and its value is estimated using the rollout policy (line 14).

Several different algorithms (e.g., UCB [51], Exp3 [56], and regret matching [73]) can be used as the selection function. We now present the variants of SM-MCTS that were consistently the most successful in the previous works, though more variants can be found in [57, 60, 61].

4.4.1. Decoupled Upper-Confidence Bound Applied to Trees

The most common selection function for SM-MCTS is the decoupled Upper-Confidence Bound applied to Trees (UCT). For the selection and updates, it executes the well-known UCT [46] algorithm independently for each of the players in each nodes. The statistics stored in the tree nodes are independently computed for each action of each player. For player $i \in \mathcal{N}$ and action $a_i \in \mathcal{A}_i(s)$ the reward sums X_{a_i} and the number of times the action was used n_{a_i} are maintained. When a joint action needs to be selected by the SELECT function, an action that maximizes the UCB value over their utility estimates is selected for each player independently (therefore it is called decoupled):

$$a_i = \operatorname*{argmax}_{a_i \in \mathcal{A}_i(s)} \left\{ \bar{X}_{a_i} + C_i \sqrt{\frac{\log n_s}{n_{a_i}}} \right\}, \text{ where } \bar{X}_{a_i} = \frac{X_{a_i}}{n_{a_i}} \text{ and } n_s = \sum_{b_i \in \mathcal{A}_i(s)} n_{b_i}. \tag{6}$$

The UPDATE function increases the visit count and rewards for each player i and its selected action a_i using $X_{a_i} \leftarrow X_{a_i} + u_i$ and $n_{a_i} \leftarrow n_{a_i} + 1$.

Consider again the example shown in Figure 6. Decoupled UCT now groups together all the payoffs obtained for an action. Therefore, at the root Max has $\bar{X}_t = 5/2 = 2.5, \bar{X}_b = 1/2 = 0.5$ and the exploration term for both is $C_i\sqrt{(\log 4)/2}$, and so top action is selected. For Min, $\bar{X}_l = 3/2 = 1.5 = \bar{X}_r$, so both actions have the same value. Therefore, Min must use a tie-breaking rule in this situation to decide which action to take. As we discuss later, the specific tie-breaking rule used here can lead to a significant effect on the quality of the strategy that UCT produces.

After all the simulations are done, there are two options for how to determine the resulting action to play. The more standard option is to choose for each state the action a_i that maximizes n_{a_i} for each player i. This is suitable mainly for games, in which using mixed strategy is not necessary. Alternatively, the action to play in each state can be determined based on the mixed strategy obtained by normalizing the visit counts of each action

$$\sigma_i(a_i) = \frac{n_{a_i}}{\sum_{b_i \in \mathcal{A}_i(s)} n_{b_i}}. \tag{7}$$

Using the first method certainly makes the algorithm not converge to a Nash equilibirum, because the game may require a mixed strategy. Therefore, unless stated otherwise, we only use the mixed form in Equation 7, which was called DUCT(mix) in [11, 61].

Note, that it was shown that this latter variant also might not converge to a Nash equilibrium (a well-known counter-example in Rock, Paper, Scissors with biased payoffs [14]). However, one of the issues when using UCT in game trees is an unspecified behavior in case there are multiple actions with identical value in the maximization described in the UCT formula in Equation 6. This may have a significant impact on the performance of the UCT in simultaneous move games. Consider the matrix game at the right of Figure 2. This game has only one NE: (a, A). However, if UCT selects the first or the last action among the options with the same value, it will always get only the utility 0 and the bias term will cause the players to round-robin over the diagonal indefinitely. This is clearly not optimal, as each player can than improve by playing first action with probability 1. However, if we choose the action to play randomly among the tied actions (where "tied" could be defined as being within a small tolerance gap), UCT will quickly converge to the optimal solution in this game. We experimentally analyze the impact of this randomization on the example used in [14] and show that if a randomized variant of UCT is used, the algorithm still does not converge to a NE but does converge to a strategy that is much closer to a NE than without randomization (see Subsection 6.3). Therefore, unless stated otherwise, we use the randomized variant in our implementation.

Even though UCT is not guaranteed to converge to the optimal solution, it is often very successful in practice. It has been used in general game playing [54], in the card game Urban Rivals [8], and in Tron [57].

4.4.2. Exponential-Weight Algorithm for Exploration and Exploitation

Another common choice of a selection function is to use the Exponential-weight algorithm for Exploration and Exploitation (Exp3) [56] independently for each of the players. Unlike with UCT, two players using Exp3 in a single stage matrix game are guaranteed to converge to a Nash equilibrium [56]; hence, we can expect a good performance of this selection function even in multi-stage games. In Exp3, each player maintains an estimate of the sum of rewards for each action, denoted $\hat{X}_{a_i}$. The joint action produced by SELECT is composed of an action independently selected for each player. An action is selected by sampling from a probability distribution over actions. Define γ to be the probability of exploring, i.e., choosing an action uniformly. The probability of selecting action a_i is proportional to the exponential of the reward estimates:

$$\sigma_i(a_i) = \frac{(1-\gamma)\exp(\eta \hat{X}_{a_i})}{\sum_{b_i \in \mathcal{A}_i(s)} \exp(\eta \hat{X}_{b_i})} + \frac{\gamma}{|\mathcal{A}_i(s)|}, \text{ where } \eta = \frac{\gamma}{|\mathcal{A}_i(s)|}. \tag{8}$$

This standard formulation of Exp3 is suitable for deriving its properties, but a straightforward implementation of this formula leads to problems with a numerical stability. Both the numerator and the denominator of the fraction can quickly become too large. For this reason, other formulations have been suggested, e.g., in [11] and [50] that are more numerically stable. We use the following equivalent formulation from [50]:

$$\sigma_i(a_i) = \frac{(1-\gamma)}{\sum_{b_i \in A_i(s)} \exp(\eta(\hat{X}_{b_i} - \hat{X}_{a_i}))} + \frac{\gamma}{|\mathcal{A}_i(s)|}. \tag{9}$$

The update after selecting actions (a_1, a_2) and obtaining a simulation result v_1 normalizes the result to the unit interval for each player by

$$u_1 \leftarrow \frac{(v_1 - v_{min})}{v_{max} - v_{min}}; \quad u_2 \leftarrow (1 - u_1), \tag{10}$$

and adds to the corresponding reward sum estimates the reward divided by the probability that the action was played by the player using

$$\hat{X}_{a_i} \leftarrow \hat{X}_{a_i} + \frac{u_i}{\sigma_i(a_i)}. \tag{11}$$

Dividing the value by the probability of selecting the corresponding action makes $\hat{X}_{a_i}$ estimate the sum of rewards over all iterations, not only the ones where a_i was selected.

As the final strategy, after all iterations are executed, the algorithm computes the *average strategy* of the Exp3 algorithm over all iterations for each player. Let σ_i^t be the strategy used at time t. After T iterations in a particular node, the average strategy is

$$\bar{\sigma}_i^T(a_i) = \frac{1}{T}\sum_{t=1}^{T}\sigma_i^t(a_i). \tag{12}$$

In our implementation, we maintain the cumulative sum and normalize it to obtain the average strategy.

Previous work [8] suggests removing the samples caused by the exploration first. This modification proved to be useful also in our experiments and it has been shown not to reduce the performance substantially in the worst case [59], so as the resulting final mixed strategy, we use

$$\bar{\sigma}_i(a_i) \leftarrow \max\left(0, \bar{\sigma}_i(a_i) - \frac{\gamma}{|\mathcal{A}_i(s)|}\right), \tag{13}$$

normalized to sum to one.

4.4.3. Regret Matching

The last selection function we propose is inspired by regret matching [73], which forms the bases of the successful algorithms for solving imperfect information games [28]. This variant applies regret matching to the current estimated matrix game at each stage and was first used in [11]. The statistics stored by this algorithm in each node are the visit count of each joint action ($n_{a_1a_2}$) and the sum of rewards for each joint action ($X_{a_1a_2}$).[4] Furthermore, the algorithm for each player i maintains a cumulative regret $r^i_{a_i}$ for having played σ_i^t instead of $a_i \in \mathcal{A}_i(s)$ on iteration t, initially set to 0. The regret values $r^i_{a_i}$ are maintained separately by each player. However, the updates use a value that is a function of the joint action space.

On iteration t, function Select first builds each player's current strategies from the cumulative regrets. Define $x^+ = \max(x, 0)$,

$$\sigma_i(a_i) = \frac{r^{i+}_{a_i}}{R^+_{sum}} \text{ if } R^+_{sum} > 0 \text{ oth. } \frac{1}{|\mathcal{A}_i(s)|}, \text{ where } R^+_{sum} = \sum_{b_i \in \mathcal{A}_i(s)} r^{i+}_{b_i}. \tag{14}$$

The main idea is to adjust the strategy by assigning the probability to actions proportionally to the regret of having not taken them over the long-term. To

[4]Note that $n_{a_1a_2}$ and $X_{a_1a_2}$ correspond to $n_{s'}$ and $X_{s'}$ from Figure 6.

ensure exploration, a sampling procedure similar to Equation 8 is used to select action a_i with probability $\gamma/|\mathcal{A}_i(s)| + (1-\gamma)\sigma_i(a_i)$.

UPDATE adds the regret accumulated at the iteration to the regret tables r^i. Suppose joint action (a_1, a_2) is sampled from the selection policy and utility u_1 is returned from the recursive call on line 9. Label $reward(b_1, b_2) = \frac{X_{b_1 b_2}}{n_{b_1 b_2}}$ if $(b_1, b_2) \neq (a_1, a_2)$, or u_1 otherwise. The updates to the regret are:

$$\forall b_1 \in \mathcal{A}_1(s), \quad r^1_{b_1} \leftarrow r^1_{b_1} + (reward(b_1, a_2) - u_1), \tag{15}$$

$$\forall b_2 \in \mathcal{A}_2(s), \quad r^2_{b_2} \leftarrow r^2_{b_2} - (reward(a_1, b_2) - u_1). \tag{16}$$

After all simulations, the strategy to play in state s is defined by the mean strategy used in the corresponding node (Equation 12).

4.4.4. Theoretical Properties

While the completeness of the exact algorithms is based on the Markov property and backward induction, the concept of the completeness is less clear for the sampling algorithms due to the randomization. Instead, we discuss a form of a probabilistic completeness. Unfortunately, none of the variants of this algorithm introduced above has been proven to eventually converge to a Nash equilibrium. If the algorithm is instantiated by UCT, Shafiei et al. [14] have shown that the algorithm converges to a stable strategy, which is not close to a Nash equilibrium. We replicate the experiment below and note that this is the case only for the deterministic version of UCT. A randomized version of UCT with a well selected exploration parameter empirically converges close to the equilibrium strategy, but then in some games oscillates and does not converge further.

The only known theoretical result about SM-MCTS directly applicable to the algorithms in this paper is negative, and it has been proven in [59].

Theorem 4.5. *There are games, in which SM-MCTS instantiated by any regret minimizing selection function with a constant exploration γ cannot converge to a strategy that would be an ϵ-Nash equilibrium for an $\epsilon < \gamma D$, where D is the depth of the game tree.*

The main idea of the proof is to define a specific class of games (see Example 2 in [59]), in which the exploration in a greater depth of the game tree causes a bias in the values observed in the higher levels of the tree, consequently leading to an incorrect decision in the root.

In order to obtain positive formal results about the convergence of SM-MCTS-like algorithms, the authors in [59] either add an additional averaging step to the algorithm (that makes it significantly slower in practical games used in benchmarks), or assume additional non-trivial technical properties about the selection function, which are not known to hold for any of the selection functions above.

As for computational complexity, the time cost per node is linear in $|\mathcal{A}_i|$ for UCT and RM. The time cost per node is quadratic in the case of Exp3 due to the numerically stable update rule (Equation 9). The memory required per node is linear for UCT and Exp3, and quadratic in $|\mathcal{A}_i|$ for RM due to storing estimates of each child subgame. This can be easily avoided by storing the mean estimates directly in the children.

4.5. Counterfactual Regret Minimization and Outcome Sampling

Finally, we describe algorithms based directly on Counterfactual Regret (CFR, a notion of regret at the information set level), first designed for extensive-form games with imperfect information [28].

Recall from Section 2 the set of histories $\mathcal{H}$. Here we also use $\mathcal{Z}$ defined previously as the set of terminal states, to refer to the set of *terminal histories* since there is a one-to-one correspondence between them. A *history* is a sequence of actions taken by all players (including chance) that starts from the beginning of the game. A history h' is a prefix of another history h, denoted $h' \sqsubset h$, if h contains h' as a prefix sequence of actions. The *counterfactual value* of reaching information set I is the expected payoff given that player i played to reach I, the opponent played σ_{-i} and both players played σ after I was reached:

$$v_i(I, \sigma) = \sum_{(h,z) \in \mathcal{Z}_I} \pi^{\sigma}_{-i}(h) \pi^{\sigma}(h, z) u_i(z), \tag{17}$$

where $\mathcal{Z}_I = \{(h, z) \mid z \in \mathcal{Z}, h \in I, h \sqsubset z\}$, $\pi^{\sigma}_{-i}(h)$ is the product of probabilities to reach h under σ excluding player i's (i.e., including chance) and $\pi^{\sigma}(h, h')$, where $h \sqsubset h'$, is the probability of all actions taken along the path from h to h'. Suppose, at time t, players play with strategy profile σ^t. Define $\sigma^t_{I \to a}$ as identical to σ^t_i except at I action a is taken with probability 1. Player i's counterfactual regret of not taking $a \in \mathcal{A}(I)$ at time t is $r^t_i(I, a) = v_i(I, \sigma^t_{I \to a}) - v_i(I, \sigma^t)$. The CFR algorithm maintains the cumulative regret $R^T_i(I, a) = \sum_{t=1}^{T} r^t_i(I, a)$, for every action at every information set.

Then, the distribution at each information set for the next iteration $\sigma^{T+1}(I)$ is obtained individually using regret-matching [73]. The distribution is proportional to the positive portion of the individual actions' regret:

$$\sigma^{T+1}(I,a) = \begin{cases} R_i^{T,+}(I,a)/R_{i,sum}^{T,+}(I) & \text{if } R_{i,sum}^{T,+}(I) > 0 \\ 1/|\mathcal{A}(I)| & \text{otherwise,} \end{cases}$$

where $x^+ = \max(0,x)$ for any term x, and $R_{i,sum}^{T,+}(I) = \sum_{a' \in A(I)} R_i^{T,+}(I,a')$. Furthermore, the algorithm maintains for each information set the average strategy profile

$$\bar{\sigma}^T(I,a) = \frac{\sum_{t=1}^{T} \pi_i^{\sigma^t}(I)\sigma^t(I,a)}{\sum_{t=1}^{T} \pi_i^{\sigma^t}(I)}, \tag{18}$$

where $\pi_i^{\sigma^t}(I) = \sum_{h \in I} \pi_i^{\sigma^t}(h)$. The combination of the counterfactual regret minimizers in individual information sets also minimizes the overall average regret [28], and hence due to the Folk Theorem the average profile is a 2ϵ-equilibrium, with $\epsilon \to 0$ as $T \to \infty$.

Monte Carlo Counterfactual Regret Minimization (MCCFR) applies CFR to sampled portions of the games [58]. In the *outcome sampling* (OS) variant, a single terminal history $z \in \mathcal{Z}$ is sampled in each iteration. The algorithm updates the regret in the information sets visited along z using the *sampled counterfactual value*,

$$\tilde{v}_i(I,\sigma) = \begin{cases} \frac{1}{q(z)} \pi_{-i}^{\sigma}(h) \pi^{\sigma}(h,z) u_i(z) & \text{if } (h,z) \in \mathcal{Z}_I \\ 0 & \text{otherwise,} \end{cases} \tag{19}$$

where $q(z)$ is the probability of sampling z. As long as every $z \in Z$ has a non-zero probability of being sampled, $\tilde{v}_i(I,\sigma)$ is an unbiased estimate of $v(I,\sigma)$ due to the importance sampling correction $(1/q(z))$. For this reason, applying CFR updates using these sampled counterfactual regrets $\tilde{r}_i^t(I,a) = \tilde{v}_i(I,\sigma^t_{I \to a}) - \tilde{v}_i(I,\sigma^t)$ on the sampled information sets values also eventually converges to the approximate equilibrium of the game with high probability. The required number of iterations for convergence is much larger, but each iteration is much faster.

4.5.1. Online Outcome Sampling

We now present Online Outcome Sampling for simultaneous move games (SM-OOS). Note, importantly, that SM-OOS is different from the general

SM-MCTS algorithms presented in Subsection 4.4. SM-OOS is an adaptation of a more general algorithm which has been proposed for search in imperfect information games [13]. However, since simultaneous move games are decomposable into subgames, the typical problems encountered in the fully imperfect information search setting are not present here. Hence, we present a simpler OOS specifically intended for simultaneous move games.

Online Outcome Sampling resembles MCTS in that it builds its tree incrementally. However, the algorithm is based on MCCFR, from Subsection 4.5, rather than on stochastic and adversarial bandit algorithms, such as UCB and Exp3. A previous version of this algorithm for simultaneous move games was presented by Lanctot et al. [11]. The version presented here is simpler for implementation and it further reduces the variance of the regret estimates, which leads to a faster convergence and better game play. The main novelty in this version is that in any state s, it defines the counterfactual values as if the game actually started in s. This is possible in simultaneous move games, because the optimal strategy in any state depends only on the part of the game below the state.

The pseudo-code is given in Algorithm 6. The game tree is incrementally built, starting only with one node for the root game state. Each node stores for each player: $R_i(s,a)$ the cumulative regret (denoted $R_i^T(I,a)$ above) of player i in state s and action a, and average strategy table $S_i(s)$, which stores the cumulative average strategy contribution for each action. Normalizing S_i gives the resulting strategy of the algorithm for player i.

The algorithm runs iterations from a starting state until it uses the given time limit. A single iteration is depicted in Algorithm 6, which recursively descends down the tree. In the root of the game, the function is run as SM-OOS($root, i$), alternating player $i \in \{1,2\}$ in each iteration. If the function reaches a terminal history of the game (line 1), it returns the utility of the terminal node for player i, and 1 for both the tail and sample probability contribution of i. If it reaches a chance node, it recursively continues after a randomly selected chance outcome (lines 3-4). If none of the first two conditions holds, the algorithm reaches a state where the players make decisions. If this state is already included in the incrementally built tree (line 5), the following state is selected based on the cumulative regrets stored in the tree by regret matching with ϵ-on-policy sampling strategy for player i (lines 6-8) and the exact regret matching strategy for player $-i$ (lines 9-11). The recursive call on line 11 then continues the iteration until the end of the game tree. If the reached node is not in the tree, it is added (line 13)

input : s – current state of the game; i – regret updating player
output: (x_i, q_i, u_i): x_i – i's contribution to tail probability ($\pi^\sigma(h,z)$); q_i – i's contribution to sample probability ($q(z)$); u_i – utility of the sampled leaf

1 **if** $s \in \mathcal{Z}$ **then return** $(1, 1, u_i(s))$
2 **else if** $s \in \mathcal{C}$ *is a chance node* **then**
3   Sample s' from $\Delta_\star(s)$
4   **return** SM-OOS(s', i)
5 **if** s *is already in the OOS tree* **then**
6     $\sigma_i \leftarrow$ RegretMatching$(R_i(s))$
7     $\forall a \in \mathcal{A}_i(s) : \sigma'_i(s,a) \leftarrow (1-\epsilon)\sigma_i(s,a) + \frac{\epsilon}{|\mathcal{A}_i(s)|}$
8   Sample action a_i from σ'_i
9     $\sigma_{-i} \leftarrow$ RegretMatching$(R_{-i}(s))$
10   Sample action a_{-i} from σ_{-i}
11     $(x_i, q_i, u_i) \leftarrow$ SM-OOS$(\mathcal{T}(s, a_i, a_{-i}), i)$
12 **else**
13   Add s to the tree
14     $\forall a \in \mathcal{A}_i(s) : \sigma_i(s,a) \leftarrow \frac{1}{|\mathcal{A}_i(s)|}$
15   Sample action a_i from σ_i
16     $\forall a \in \mathcal{A}_{-i}(s) : \sigma_{-i}(s,a) \leftarrow \frac{1}{|\mathcal{A}_{-i}(s)|}$
17   Sample action a_{-i} from σ_{-i}
18     $(x_i, q_i, u_i) \leftarrow$ OOS-Rollout$(\mathcal{T}(s, a_i, a_{-i}))$
19 $W \leftarrow u_i \cdot x_i / q_i$
20 $R_i(s,a_i) \leftarrow R_i(s,a_i) + \frac{1-\sigma_i(s,a_i)}{\sigma'_i(a_i)} W$
21 $\forall a \in \mathcal{A}_i(s) \setminus \{a_i\} : R_i(s,a) \leftarrow R_i(s,a) - \frac{\sigma_i(s,a_i)}{\sigma'_i(s,a_i)} W$
22 $S_{-i}(s) \leftarrow S_{-i}(s) + \sigma_{-i}$
23 **return** $(x \cdot \sigma_i(s,a_i), q \cdot \sigma'_i(s,a_i), u_i)$

Algorithm 6: Simultaneous Move Online Outcome Sampling (SM-OOS)

and an action for each player is selected based on the uniform distribution (lines 14-16). Afterwards, a random rollout of the game until a terminal node is initiated on line 18. The rollout is similar to the MCTS case, but in addition, it has to compute the tail probability x_i and the sampling probability q_i required to compute the sampled counterfactual value. For example, if in the rollout player i acts n_i times, and each time samples uniformly from exactly b actions, then $x_i = \frac{1}{b^{n_i}}$. Regardless of whether the current node was in the tree, the algorithm updates the regret table of player i based on the simplified definition of sampled counterfactual regret for simultaneous move games (lines 19-21) and the mean strategy of player $-i$ (line 22). Finally, the function returns the updated probabilities to the upper level of the tree.

SM-OOS appears similar to SM-MCTS using the RM selection mechanism (Subsection 4.4.3). However, there are a number of differences: SM-OOS uses importance sampling of a sequence of probabilities to keep its estimate unbiased, but will suffer a higher variance than RM which uses only a one-step correction. RM does not distinguish whether its utility comes from exploration or otherwise, whereas SM-OOS separates the two into the tail probabilities of the strategy for the sequence sampled (x_i) and the sampling probability of the sequence (q_i); when $\sigma_i(s,a) = 0$, due to exploration, then $x_i = 0$ and the value of the update increments are also 0. RM uses the means from the subgames as estimates of utility for those subgames, which could introduce some bias in the estimators. We further discuss the comparison in Subsection 6.6.

4.5.2. Theoretical Properties

SM-OOS, contrary to the MCTS-based algorithms, has finite-time probabilistic convergence guarantees. Since SM-OOS is designed to update each node of the game in the same way as the root of the game, we present the following theorem from the perspective of the root of the entire game. It holds also for starting the algorithm in non-root nodes, but the values of $|S|$ and δ can be adapted to represent the subgame.

Theorem 4.6. *When SM-OOS is run from the root of the game, with probability $(1-p)$ an ϵ-NE is reached after $O(\frac{|\mathcal{A}||\mathcal{S}|^2\Delta_{u,i}^2}{p\delta^2\epsilon^2})$ iterations, where $|\mathcal{A}| = \max_{s\in\mathcal{S}, i\in\{1,2\}} |\mathcal{A}_i(s)|$, $\Delta_{u,i} = \max_{z,z'\in\mathcal{Z}} |u_i(z') - u_i(z)|$, and δ is the smallest probability of sampling any single leaf in the subtree of the root node.*

Proof The proof is composed of two observations. First, the whole game tree is eventually built by the algorithm. A direct consequence of [59, Lemma 40] is that the tree of depth D is built with probability $(1-p_1)$ in less than

$$16D\left(\frac{|\mathcal{A}|}{\gamma}\right)^{2D} \max(D, 4\log p_1^{-1} + 4) \tag{20}$$

iterations by an algorithm with a fixed exploration γ. This is the number of iterations needed for each leaf in the game to be visited at least D times.

Second, during these and the following iterations, the algorithm performs exactly the same updates in the nodes contained in memory, as the Outcome Sampling (OS) MCCFR [58]. If some nodes below a state were not added to the tree yet, a uniform strategy is assumed in these states for the regret updates. Since CFR minimizes the counterfactual regret in an individual information set regardless of the strategies in other information sets, the samples acquired during the tree building cannot have a negative impact on the rate of regret minimization in individual states. Therefore, we can use [74, Theorem 4] that bounds the number of iterations needed for OS as an offline solver with the complete game in the memory, starting after the tree has been built with a high probability. It states that with probability $(1-p_2)$ an ϵ-NE is reached after $O(\frac{|\mathcal{A}||\mathcal{S}|^2\Delta_{u,i}^2}{p_2\delta^2\epsilon^2})$ iterations.

We can see that the OS bound dominates the time required to build the tree. A single explorative action is taken with probability $\gamma/|\mathcal{A}|$, and when sampling a terminal z only due to exploration, $\frac{1}{\delta} = (\frac{|\mathcal{A}|}{\gamma})^{2D}$, and $D^2 < |\mathcal{A}|^{2D} \in O(|\mathcal{S}|)$ for any $\mathcal{A}$, and we can set $p_1 = p_2 = p/2$. Then the probability that both the tree will be built and the convergence will be achieved can be bounded by $(1-p_1)(1-p_2) \geq (1-p)$. □

As for computational complexity, the time cost as well as the memory required per node is linear in $|\mathcal{A}_i|$ in SM-OOS.

5. Online Search

In this section, we describe online adaptations of the algorithms described in the previous section and their application to any-time search given a limited time budget.

5.1. Iterative Deepening Backward Induction Algorithms

Minimax search [5] has been used with much success in sequential perfect information games, leading to super-human chess AI, one of the key advances of artificial intelligence [1]. Minimax search is an online application of backward induction run on a heuristically approximated game. The game is approximated by searching to a fixed depth limit d, treating the states at depth d as terminal states, evaluating their values using a heuristic evaluation function, $eval(s)$. The main focus is to compute an optimal strategy for this heuristic approximation of the original game.

Similarly to the perfect information case, we can modify our algorithms based on backward induction for simultaneous move games. Under the limited time settings, a search algorithm is given a fixed time budget to compute a strategy. We use the classic approach of *iterative deepening* [5] that runs several depth-limited searches, starting at a low depth and iteratively increasing the depth of each successive search. Note that the depth limit of d means that the algorithm evaluates d joint actions (i.e., pairs of simultaneous actions) possibly preceded by a chance outcome if present.

In iterative deepening, the algorithm by default starts at depth $d = 1$ and gradually increases d until there is no more time. In our implementation of iterative deepening we follow a natural observation that saves the computation time *between different searches*: a solution computed in state s by player i to depth d contains an optimal solution on $d - 1$ approximation of subgames starting in possible next states $\mathcal{T}(s, r, c)$, where r is the action selected for the player performing the search and c is the action of the opponent. Therefore, when the iterative deepening algorithm starts a new search in state $s' \in \mathcal{T}(s, r, c)$, it can often begin at depth d. This can require space exponential in the depth d in the worst case, but it is beneficial in practical experiments. When information is missing due to pruning, then a search starts with the lowest possible depth $d = 1$.

5.2. Online Search using Sampling Algorithms

Using sampling algorithms in the online settings is simpler than with the algorithms based on backward induction, since no significant changes are needed and the algorithms do not need an evaluation function. The algorithms are stopped after a given time limit and the move to play or the complete strategy is extracted as described for each sampling algorithm in Section 4.

There are two concepts that have to be discussed. First, the algorithms can re-use all information and statistics gained in the previous iterations; hence, after returning a move and advancing to a succeeding state of the game s', the subtree of the incrementally built tree rooted in s' is preserved and used in the next iterations. Note that reusing the previously gathered statistics in the sub-tree rooted in s' has no potentially negative effect on any variant of the MCTS algorithms since the behavior of the algorithms is exactly the same when the iteration is started in this node, and if this node is reached from its predecessor. This is also true in SM-OOS because of the structure of simultaneous move games; a similar adaptation of the algorithm is not possible in more general imperfect information games [13].

Second, even though the sampling algorithms do not require the use of domain-specific knowledge for online search, they often incorporate this type of knowledge to better guide the sampling and thus to evaluate more relevant parts of the state space [75, 76, 77, 78, 79]. When directly comparing approximative sampling algorithms with the backward induction algorithms using an evaluation function, the outcome of such a comparison strictly depends on the quality of the evaluation function. In a very large game, an accurate evaluation function greatly benefits the backward induction algorithm. Therefore, we also use sampling algorithms combined with an evaluation function. The integration is done via replacing the random rollout by directly using the value of the evaluation function in the current state for MCTS and OOS algorithms; i.e., Rollout(s) in line 14 of Algorithm 5 or line 18 of Algorithm 6 is replaced by $eval(s)$. This has been commonly used in several previous works in Monte Carlo search [76, 78, 79, 80, 81].

Again, such a modification does not generally affect theoretical properties of the algorithms – the proofs of the convergence assume that a whole game tree is eventually built and any statistics in the nodes collected before (either by random rollouts or evaluation functions) can eventually be over-weighted. For MCTS algorithms, there is no reason to believe that a good evaluation function would give a worse estimate of the quality of a sub-tree using random play-outs. The only complication could be with the way the probabilities are computed in OOS. The weight of the sample in Equation 19 is multiplied by the probability of reaching the terminal state z from some history h, $\pi^\sigma(h,z)$. However, the "tail" probability is canceled because the rollout policy is fixed and so its contribution to $q(z)$ is identical to its contribution to $\pi^\sigma(h,z)$.

6. Empirical Evaluation

We now present a thorough experimental evaluation of the described algorithms. We analyze both the offline and the online case on a collection of games inspired by previous work, and randomly generated games. After describing rules and properties of the games, we present the results for the offline strategy computation and we follow with the online game playing.

6.1. Experimental Settings

We start with an experimental evaluation of a well-known example of Biased Rock, Paper, Scissors [14] that often serves as an example that MCTS with UCT selection function does not converge to a Nash equilibrium. We reproduce this experiment and show the differences in performance of the sampling algorithms – primarily the impact of randomization in UCT. Then, we compare the offline performance of the algorithms on other domains. For each domain, we first analyze the exact algorithms and measure the computation time taken to solve a particular instance of the game. Afterward, we analyze the convergence of the approximative algorithms. At a specified time step the algorithm produces strategies (σ_1, σ_2). Using best responses we compute $\mathrm{error}(\sigma_1, \sigma_2) = \max_{\sigma_1' \in \Sigma_1} \mathbb{E}_{z \sim (\sigma_1', \sigma_2)}[u_1(z)] + \max_{\sigma_2' \in \Sigma_2} \mathbb{E}_{z \sim (\sigma_1, \sigma_2')}[u_2(z)]$, which is equal to 0 at a Nash equilibrium. In each offline convergence setting, the reported values are means over at least 20 runs of each sampling algorithm on a single instance of the game. We compared at least 3 different settings for each exploration parameter and present the result only for the best exploration parameter. For OOS, Exp3, and RM the best values for the parameters were almost always 0.6, 0.1, and 0.1, respectively. The only exception was Goofspiel with chance, where both Exp3 and RM converge faster with the parameter set to 0.3. We give the optimal value for UCT constant C in each setting.

Finally, we turn to the comparison of the algorithms in the online setting and we present results from head-to-head tournaments in each game. Here, we use larger instances of each game and compare the algorithms based on actual game play with a limited time for each move. The algorithms based on backward induction need to use a domain-specific evaluation function in the online setting. This may give these algorithms an advantage if the evaluation function is accurate. Therefore, we also run the sampling-based algorithms with an evaluation function for selected domains to compare the algorithms in a fairer setting. Moreover, we have also tuned parameters for the sampling algorithms specifically for each domain. Reported results are means over at least 1000 matches for each pair of algorithms.

Each of the described algorithms was implemented in a generic framework for modeling and solving extensive-form games[5]. We are interested in the

[5]Source code is available at the web page of the authors. We use IBM CPLEX 12.5 to solve the linear programs.

performance of the algorithms and their ability to find or approximate the optimal behavior. Therefore, with the exception of the evaluation function used in selected online experiments, no algorithm uses any domain-specific knowledge.

6.2. Domains

In this subsection, we describe the six domains used in our experiments. The games in our collection differ in characteristics, such as the number of available actions for each player (i.e., the branching factor), the maximal depth, and the number of possible utility values. Moreover, the games also differ in the *randomization factor* – i.e., how often it is necessary to use mixed strategies and whether this randomization occurs at the beginning of the game, near the end of the game, or is spread throughout the whole course of the game.

For each domain we also describe the evaluation function used in the online experiments. Note that we are not seeking the best-performing algorithm for a particular game; hence, we have not aimed for the most accurate evaluation functions for each game. We intentionally use evaluation functions of different quality that allow us to compare the differences between the algorithms from this perspective as well.

Biased Rock, Paper, Scissors. BRPS is a payoff-skewed version of the one-shot game Rock, Paper, Scissors shown in Figure 7. This game was introduced in [14], and it was shown that the visit count distribution of UCT converges to a fixed balanced situation, but not one that corresponds to the optimal mixed strategy of $(\frac{1}{16}, \frac{10}{16}, \frac{5}{16})$.

	r	p	s
R	0	-25	50
P	25	0	-5
S	-50	5	0

Figure 7: Biased Rock, Paper, Scissors matrix game from [14].

Goofspiel. Goofspiel is a card game that appears as a common example of a simultaneous move game (e.g., [35, 37, 38, 11]). There are 3 identical decks of d cards with values $\{0, \ldots, (d-1)\}$, one for chance and one for each player, where d is a parameter of the game. Standard Goofspiel is played with 13 cards. The game is played in rounds: at the beginning of each round, chance reveals one card from its deck and both players bid for the card by simultaneously selecting (and removing) a card from their hands. A player that selects a higher card wins the round and receives a number of points equal to the value of the chance's card. In case both players select the card with the same value, the chance's card is discarded. When there are no more cards to be played, the winner of the game is chosen based on the sum of card values he received during the whole game.

There are two parameters of the game that can be altered to create four different variants of Goofspiel. The first parameter determines whether or not the chance player is included. We can use an assumption made in the previous work that used Goofspiel as a benchmark for evaluation of the exact offline algorithms [38], where the sequence of the cards is randomly chosen at the beginning of the game and it is known to both players. We refer to this setting as the *fixed sequence* of cards. Alternatively, we can treat chance in the standard way, where chance nodes determine the card that gets drawn. We refer to this setting as the *stochastic sequence*. The games are fairly similar in terms of performance of the algorithms, however, the second variant induces a considerably larger game tree. The second parameter relates to the utility functions. Either we treat the game as a win-tie-lose game (i.e., the players receive utility from $\{-1, 0, 1\}$), or the utility values for the players are equal to the points they gain during the game.

Goofspiel forms game trees with interesting properties. First unique feature is that the number of actions for each player is uniformly decreasing by 1 with the depth. Secondly, algorithms must randomize in NE strategies, and this randomization is present throughout the whole course of the game. As an example, the following table depicts the number of states with pure strategies and mixed strategies for each depth in a subgame-perfect NE calculated by backward induction for Goofspiel with 5 cards and a fixed sequence of cards:

Depth	0	1	2	3	4
Pure	0	17	334	3,354	14,400
Mixed	1	8	66	246	0

We can see that the relative number of states with mixed strategies slowly decreases, however, players need to mix throughout the whole game. In the last round, each player has only a single card; hence, there cannot be any mixed strategy.

Our hand-tuned evaluation function used in Goofspiel takes into consideration the remaining cards in the deck weighted by a chance of winning these cards depending on the remaining cards on hand for each player. Moreover, if the position is clearly winning for one of the players (there is not enough cards to change the current score), the evaluation function is set to maximal (or minimal) value. The formal definition follows (c_i is the sum of values of the remaining cards of player i):

$$eval(s) = \begin{cases} u_1(s) & \text{if } c_1 + c_2 = 0 \ ; \\ \tanh\left(\frac{c_1 - c_2}{c_1 + c_2} \cdot \frac{c_\star}{0.5 \cdot d(d+1)}\right) & \text{otherwise.} \end{cases}$$

For the win-tie-lose case we use tanh to scale the evaluation function into the interval $[-1, 1]$; this function is omitted in the exact point case.

Oshi-Zumo. Oshi-Zumo (also called *Alesia* in [22]) is a board game that has been analyzed from the perspective of computational game theory in [36]. There are two players in the game, both starting with N coins, and there is a board represented as a one-dimensional playing field with $2K + 1$ locations (indexed $0, \ldots, 2K$). At the beginning, there is a stone (or a *wrestler*) located in the center of the playing field (i.e., at position K). During each move, both players simultaneously place their bid from the amount of coins they have (but at least M if they still have some coins). Afterward, the bids are revealed, both bids are subtracted from the number of coins of the players, and the highest bidder can push the wrestler one location towards the opponent's side. If the bids are the same, the wrestler does not move. The game proceeds until the money runs out for both players, or the wrestler is pushed out of the field. The winner is determined based on the position of the wrestler – the player in whose half the wrestler is located loses the game. If the final position of the wrestler is the center, the game is a draw. Again, we have examined two different settings of the utility values: they are either restricted to win-tie-lose values $\{-1, 0, 1\}$, or they correspond to the relative position of the wrestler $\{\text{wrestler} - K, K - \text{wrestler}\}$. In the experiments we varied the number of coins and parameter K.

Many instances of the Oshi-Zumo game have a pure Nash equilibrium. With the increasing number of the coins the players need to use mixed strategies, however, mixing is typically required only at the beginning of the game.

As an example, the following table depicts the number of states with pure strategies and mixed strategies in a subgame-perfect NE calculated by backward induction for Oshi-Zumo with $N = 10$ coins, $K = 3$, and minimal bid $M = 1$. The results show that there are very few states where mixed strategies are required, and they are present only at the beginning of the game tree. Also note, that contrary to Goofspiel, not all branches have the same length.

Depth	0	1	2	3	4	5	6	7	8	9
Pure	1	98	2,012	14,767	48,538	79,926	69,938	33,538	8,351	861
Mixed	0	1	4	17	8	0	0	0	0	0

The evaluation function used in Oshi-Zumo takes into consideration two components: (1) the current position of the wrestler and, (2) the remaining coins for each player. Formally:

$$eval(s) = \tanh\left(\frac{b}{2} + \frac{1}{3}\left(\frac{\text{coins}_1 - \text{coins}_2}{M} + \text{wrestler} - K\right)\right),$$

where $b = 1$ if $\text{coins}_1 \geq \text{coins}_2$ and $wrestler \geq K$, and at least one of the inequalities is strict; or $b = -1$ if $\text{coins}_1 \leq \text{coins}_2$ and $wrestler \leq K$, and at least one of the inequalities is strict; $b = 0$ otherwise. Again, we use tanh to scale the value into the interval $[-1, 1]$ only in the win-tie-lose case.

Pursuit-Evasion Games. Another important class of games is pursuit-evasion games (for example, see [82]). There is a single evader and a pursuer that controls 2 pursuing units on a four-connected grid in our pursuit-evasion game. Since all units move simultaneously, the game has larger branching factor than Goofspiel (up to 16 actions for the pursuer). The evader wins if she successfully avoids the units of the pursuer for the whole game. The pursuer wins if her units successfully capture the evader. The evader is captured if either her position is the same as the position of a pursuing unit, or the evader used the same edge as a pursuing unit (in the opposite direction). The game is win-loss and the players receive utility from the set $\{-1, 1\}$. We use 3 different square four-connected grid-graphs (with the size of a side 4, 5, and 10 nodes) for the experiments without any obstacles or holes. In the experiments we varied the maximum length of the game d and we altered the starting positions of the players (the distance between the pursuers and the evader was always at most $\left\lfloor \frac{2}{3}d \right\rfloor$ moves, in order to provide a possibility for the pursuers to capture the evader).

Similarly to Oshi-Zumo, many instances of pursuit-evasion games have a pure Nash equilibrium. However, the randomization can be required towards the actual end of the game in order to capture the evader. Therefore, depending on the length of the game and the distance between the units, there might be many states that do not require mixed strategies (the units of the pursuers are simply going towards the evader). Once the units are close to each other, the game may require mixed strategies for the final coordination. This can be seen on our small example on a graph with 4×4 nodes and depth 5:

Depth	0	1	2	3	4
Pure	1	12	261	7,656	241,986
Mixed	0	0	63	1,008	6,726

The evaluation function used in pursuit-evasion games takes into consideration the distance between the units of the pursuer and the evader (denoted distance_j for the distance in moves of the game between the j^{th} unit of the pursuer and the evader). Formally:

$$eval(s) = \frac{\min(\text{distance}_1, \text{distance}_2) + 0.01 \cdot \max(\text{distance}_1, \text{distance}_2)}{1.01 \cdot (w + l)},$$

where w and l are dimensions of the grid graph.

Random/Synthetic Games. Finally, we also use randomly generated games to be able to experiment with additional parameters of the game, mainly larger utility values and their correlation. In randomly generated games, we fixed the number of actions that the players can play in each stage to 4 and 5 (the results were similar for different branching factors) and we varied the depth of the game tree. We use 2 different methods for randomly assigning the utility values to the terminal states of the game: (1) the utility values are uniformly selected from the interval $[0, 1]$; (2) we randomly assign either -1, 0, or $+1$ value to each joint action (pair of actions) and the utility value in a leaf is a sum of all the values on the edges on the path from the root of the game tree to the leaf. The first method produces extremely difficult games for pruning using either alpha-beta, or the double-oracle algorithm, since there is no correlation between actions and utility values in sibling leaves. The latter method is based on random *P-games* [83] and creates more realistic games using the intuition of good and bad moves.

Randomly generated games represent games that require mixed strategies in most of the states. This holds even for the games of the second type with correlated utility values in the leaves. The following table shows the number of states depending on the depth for a randomly generated game of depth 5 with 4 actions available to both players in each state:

Depth	0	1	2	3	4
Pure	0	2	29	665	20,093
Mixed	1	14	227	3,431	45,443

Only the second type of randomly generated games is used in the online setting. The evaluation function used in this case is computed similarly to the utility value and it is equal to the sum of values on the edges from the root to the current node.

Tron. Tron is a two-player simultaneous move game played on a discrete grid, possibly obstructed by walls [55, 57, 60]. At each step, both players move to adjacent nodes and a wall is placed to the original positions of the players. If a player hits the wall or the opponent, the game ends. The goal of both players is to survive as long as possible. If both players move into a wall, off the board, or into each other on the same turn, the game ends in a draw. The utility is $+1$ for a win, 0 for a draw, and -1 for a loss. In the experiments, we used an empty grid with no obstacles and various sizes of the grid.

Similarly to pursuit-evasion games, there are many instances of Tron that have pure NE. However, even if mixed strategies are required, they appear in the middle of the game once both players reach the center of the board and compete over the advantage of possibly being able to occupy more squares. Once this is determined, the endgame can be solved in pure strategies since it typically consists of filling the available space in an optimal ordering one square at a time. The following table comparing the number of states demonstrates this characteristics of Tron on a 5×6 grid:

Depth	0	1	2	3	4	5	...
Pure	1	4	14	100	565	2,598	
Mixed	0	0	2	0	9	7	

...	6	7	8	9	10	11	12	13
	9,508	25,964	54,304	83,624	87,009	63,642	23,296	3,127
	51	92	106	121	74	0	0	0

The evaluation function is based on how much space is "owned" by each player, which is a more accurate version of the space estimation heuristic [84] that was used in [60]. A cell is owned by player i if it can be reached by player i before the opponent. These values are computed using an efficient flood-fill algorithm whose sources start from the two players' current positions:

$$eval(s) = \tanh\left(\frac{\text{owned}_1 - \text{owned}_2}{5}\right).$$

6.3. Non-Convergence and Random Tie-Breaking in UCT

We first revisit the counter-example given in [14] showing that UCT does not converge to an equilibrium strategy in Biased Rock, Paper, Scissors when using a mixed strategy created by normalizing the visit counts. We expand on this result, showing the effect of the synchronization occurring when the UCT selection mechanism is fully deterministic (see Subsection 4.4.1).

We run SM-MCTS with UCT, Exp3, and Regret Matching selection functions on Biased Rock, Paper, Scissors for 100 million (10^8) iterations, measuring the exploitability of the strategy recommended by each variant at regular intervals. The results are shown in Figures 8 and 9.

The first observation is that deterministic UCT does not seem to converge to a low-exploitability strategy (see Figure 8, top figure). The exploitability of the strategies of Exp3 and RM variants do converge to low-exploitability strategies (see Figure 9), and the resulting approximation depends on the amount of exploration. If less exploration is used, then the resulting strategy is less exploitable, which is natural in the case of a single state. RM does seem to converge slightly faster than Exp3, as we will see in the remaining domains as well.

We then tried adding a stochastic tie-breaking rule to the UCT selection mechanism typically used in MCTS implementations, which chooses an action randomly when the scores of the best values are "tied" (less than 0.01 apart). The bottom figure in Figure 8 shows the convergence. One particularly striking observation is that this simple addition leads to a large drop in the resulting exploitability, where the exploitability ranges from $[0.5, 0.8]$

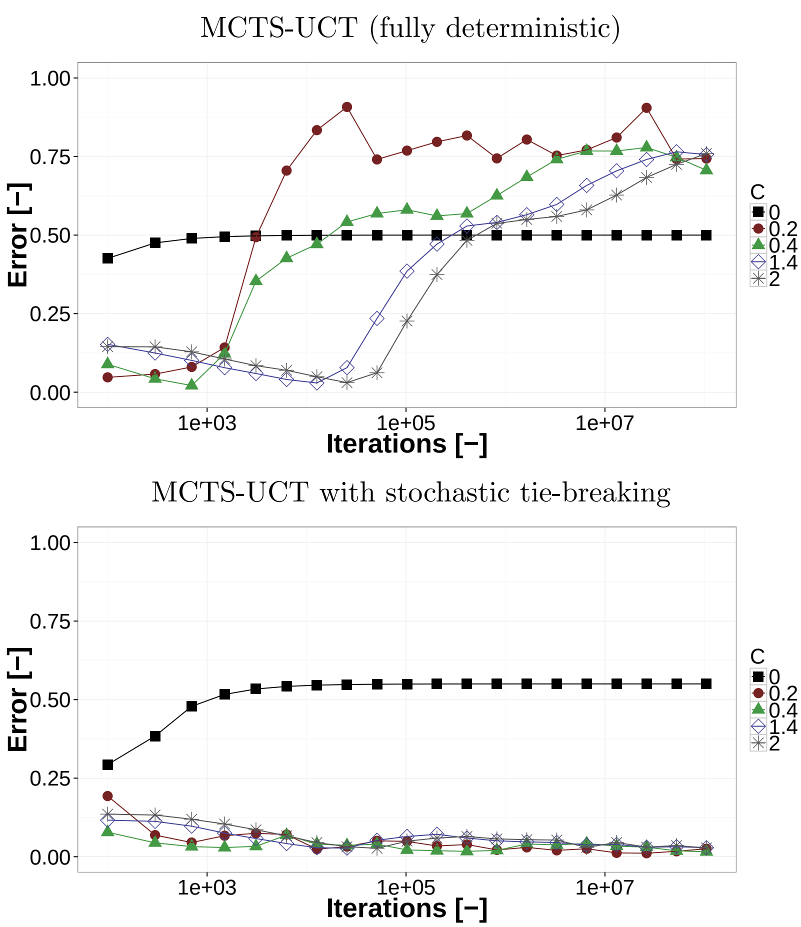


Figure 8: Exploitability of strategies of recommended by MCTS-UCT over time in Biased Rock, Paper, Scissors. Vertical axis represents exploitability.

in the deterministic case, compared to $[0.01, 0.05]$ with the stochastic tie-breaking. Therefore, the stochastic tie-breaking is enabled in all of our experiments.

In summary, with this randomization UCT appears to be converging to an approximate equilibrium in this game but not to an exact equilibrium, which is similar to results of a variant of UCT in Kuhn poker [85].

6.4. Offline Equilibrium Computation

We now compare the offline performance of the algorithm on all the remaining games. We measure the overall computation time for each of the

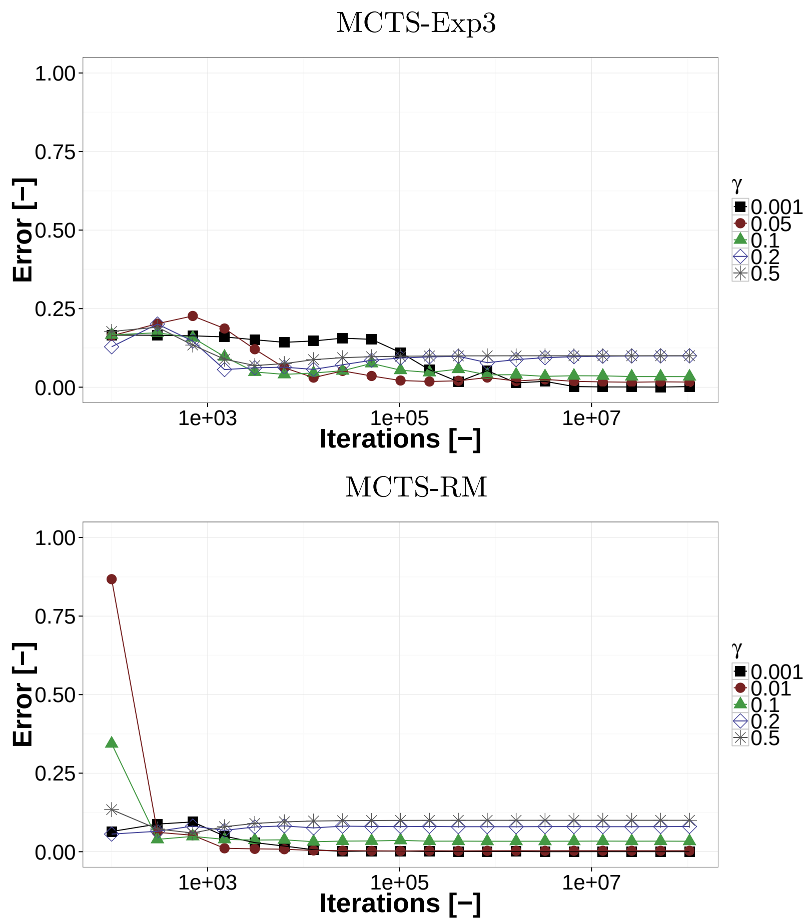
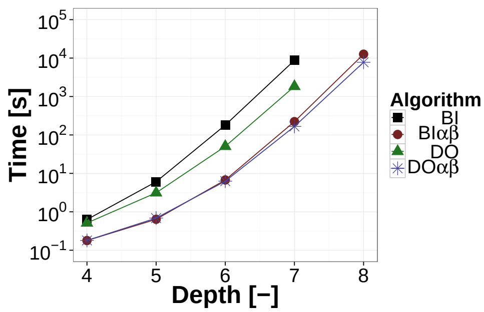


Figure 9: Exploitability of strategies recommended by MCTS-Exp3 and MCTS-RM over time in Biased Rock, Paper, Scissors. Vertical axis represents exploitability.

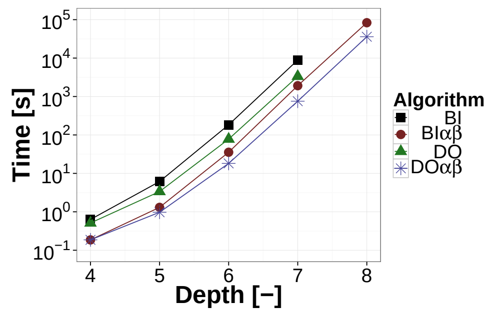
algorithms and the number of evaluated nodes – i.e., the nodes for which the main method of the backward induction algorithm executed (nodes evaluated by serialized alpha-beta algorithms are not included in this count, since they may be evaluated repeatedly). Unless otherwise stated, each data point represents a mean over at least 30 runs.

6.4.1. Goofspiel

We now describe the results for the card game Goofspiel. First, we analyze the games with fixed sequences of the cards.



(a)

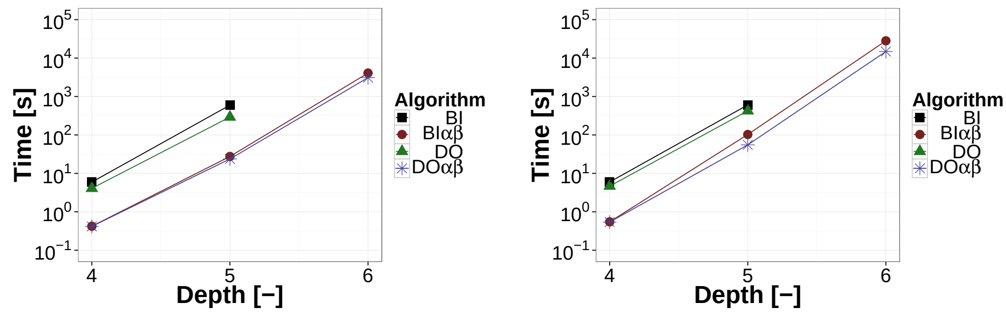


(b)

Figure 10: Running times of the exact algorithms on Goofspiel with fixed sequences of cards for increasing size of the deck; subfigure (a) depicts the results with win-tie-lose utilities, (b) depicts the results with point difference utilities.

Exact algorithms with fixed sequences. The results are depicted in Figure 10 (note the logarithmic vertical scale), where the left subfigure depicts the results for win-tie-lose utilities and the right subfigure depicts the results for point utilities. We present the mean results over 10 different fixed sequences. The comparison on the win-tie-lose variant shows that there is a significant number of subgames with a pure Nash equilibrium that can be computed using the serialized alpha-beta algorithms. Therefore, the performance of BI$\alpha\beta$ and DO$\alpha\beta$ is fairly similar and the gap only slowly increases in favor of DO$\alpha\beta$ with the increasing size of the game. Since serialized alpha-beta is able to solve a large portion of subgames, both of these algorithms significantly reduce the number of the states visited by the backward induction algorithm. While BI evaluates 3.2×10^7 nodes in the setting with 7 cards in more than 2.5 hours, BI$\alpha\beta$ evaluates only $198,986$ nodes in less than 4 minutes. The performance is further improved by DO$\alpha\beta$ that evaluates on average $79,105$ nodes in less than 3 minutes. The overhead is slightly higher in case of DO$\alpha\beta$; hence, the time difference between DO$\alpha\beta$ and BI$\alpha\beta$ is relatively small compared to the difference in evaluated nodes. Finally, the results show that even the DO algorithm without the serialized alpha-beta search can improve the performance of BI. In the setting with 7 cards, DO evaluates more than 6×10^6 nodes which takes on average almost 30 minutes.

The results for the point utilities are the same for BI, while DO is slightly worse. On the other hand, the success of serialized alpha-beta algorithms is significantly lower and it takes both algorithms much more time to solve the



(a) (b)

Figure 11: Running times of exact algorithms on Goofspiel with chance nodes for increasing size of the deck; subfigure (a) depicts the results with win-tie-lose utilities, (b) depicts the results with point utilities.

games of the same size. With 7 cards, BI$\alpha\beta$ evaluates more than 2×10^6 nodes and it takes the algorithm on average 32 minutes to find the solution. DO$\alpha\beta$ is still the fastest and it evaluates more than 3×10^5 nodes in less than 13 minutes on average.

The performance of algorithms BI$\alpha\beta$ and DO$\alpha\beta$ represent a significant improvement over the results of the pruning algorithm SMAB presented in [38]. In their work, the number of evaluated nodes was at best around 29%, and the running time improvement was only marginal.

Exact algorithms with a stochastic sequence. Next we compare the exact algorithms in the variant of Goofspiel with standard chance nodes. Introducing another branching due to moves by chance causes a significant increase in the size of the game tree. For 7 cards, the game tree has more than 10^{11} nodes, which is 4 orders of magnitude more than in the case with fixed sequences of cards. The results depicted in Figure 11 show that the games become quickly too large to solve exactly and the fastest algorithms solved games with at most 6 cards. Relative performance of the algorithms, however, is similar to the case with fixed sequences. With win-tie-lose utilities, serialized alpha-beta is again able to find pure NE in most of the subgames and prunes out a large fraction of the states. For the game with 5 cards, BI evaluates more than 2×10^6 nodes in almost 10 minutes, while BI$\alpha\beta$ evaluates only $17,315$ nodes in 27 seconds and DO$\alpha\beta$ evaluates $6,980$ nodes in 23 seconds. As before, the serialized alpha-beta algorithm is less helpful in the case with point utilities. Again with 5 cards, BI$\alpha\beta$ evaluates $91,419$ nodes in more

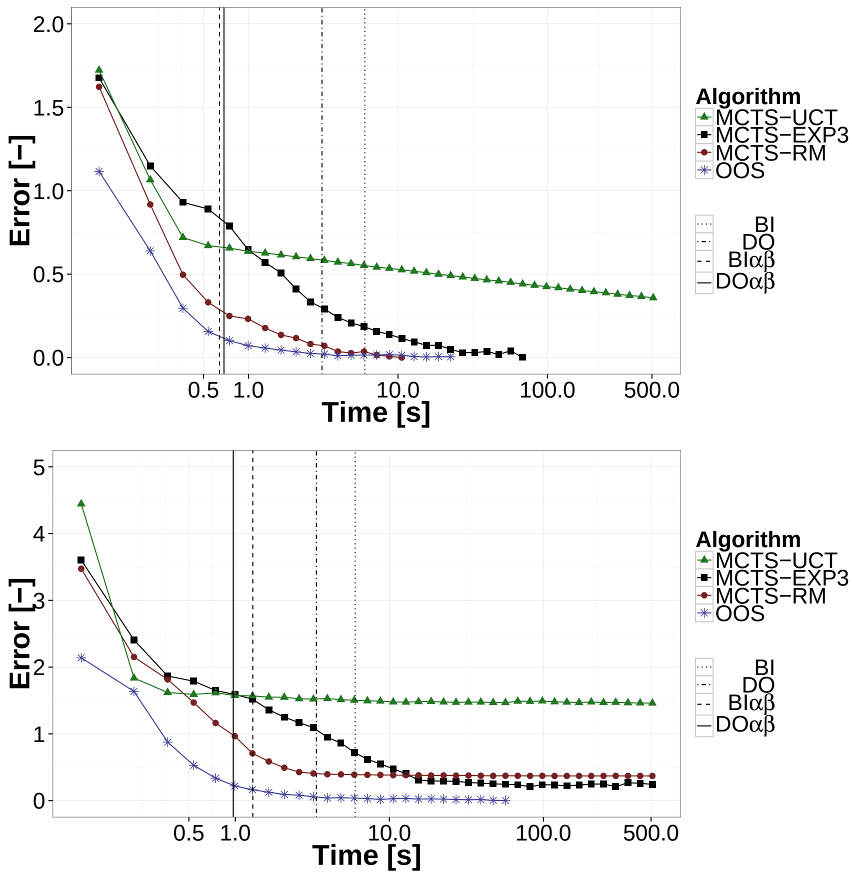


Figure 12: Convergence of the sampling algorithms on Goofspiel with 5 cards and a fixed sequence of cards. The vertical lines correspond to the computation times for the exact algorithms. (Top) Goofspiel with win-tie-lose utility values; (Bottom) Goofspiel with point utilities.

than 100 seconds and DO$\alpha\beta$ evaluates $14,536$ nodes in almost 55 seconds.

Sampling algorithms with fixed sequences. We now turn to the analysis of the convergence of the sampling algorithms – i.e., their ability to approximate Nash equilibrium strategies of the complete game. Figure 12 depicts the results for Goofspiel game with 5 cards with fixed sequence of cards (note the logarithmic horizontal scale). We compare MCTS algorithms with three different selection functions (UCT, Exp3, and RM), and OOS. The results are means over 30 runs of each algorithm. Due to the different selection and update functions, the algorithms differ in the number of iterations per second. RM is the fastest with more than 2.6×10^5 iterations per second, OOS has around 2×10^5 iterations, UCT 1.9×10^5, and Exp3 only 5.4×10^4 iterations.

The results show that OOS converges the fastest out of all sampling algorithms. This is especially noticeable in the point-utility settings, where none of the other sampling algorithms were approaching zero error due to the exploration. MCTS with RM selection function is only slightly slower in the win-tie-lose case, however, the other two selection functions perform worse. While Exp3 eventually converges close to 0 in the win-tie-lose case, the exploitability of UCT decreases rather slowly and it was still over 0.35 at the time limit of 500 seconds. The best C constant for UCT was 5 in the win-tie-lose setting, and 10 in the point utility setting. While setting lower constant typically improves the convergence rate slightly during the first iterations, the final error was always larger. The vertical lines represent the times for the exact algorithms. In the win-tie-lose case, BI$\alpha\beta$ is slightly faster and finishes first in 0.64 seconds, followed by DO$\alpha\beta$ (0.69 seconds), DO (3.1 seconds), and BI (6 seconds). In the point case, DO$\alpha\beta$ is the fastest (0.97 seconds), followed by BI$\alpha\beta$ (1.3 seconds), followed by DO and BI with similar times as in the previous case.

Sampling algorithms with a stochastic sequence. We also performed the experiments in the setting with chance nodes. Due to the size of the game tree, we have reduced the number of cards to 4, since the size of this game tree is comparable to the case with 5 cards and a fixed sequence of cards. The results depicted in Figure 13 show a similar behavior of the sampling algorithms as observed in the previous case. OOS converges the fastest, followed by RM, and Exp3. The main difference is in the convergence of UCT, however, this is mostly due to the fact that a pure NE exists in Goofspiel with 4 cards; hence, UCT can better identify the best action to play and converges faster to a less exploitable strategy than in the case with 5 cards. Surprisingly, the convergence rates of the algorithms do not change that dramatically with the introduction of point utilities as in the previous case. The main reason is that the range of the utility values is smaller compared to the previous case (there is one card less in the present setting and the missing cards has the highest value). For comparison, we again use the vertical lines to denote times of exact algorithms. BI$\alpha\beta$ and DO$\alpha\beta$ are almost equally fast, with DO$\alpha\beta$ being slightly faster, followed by DO and BI.

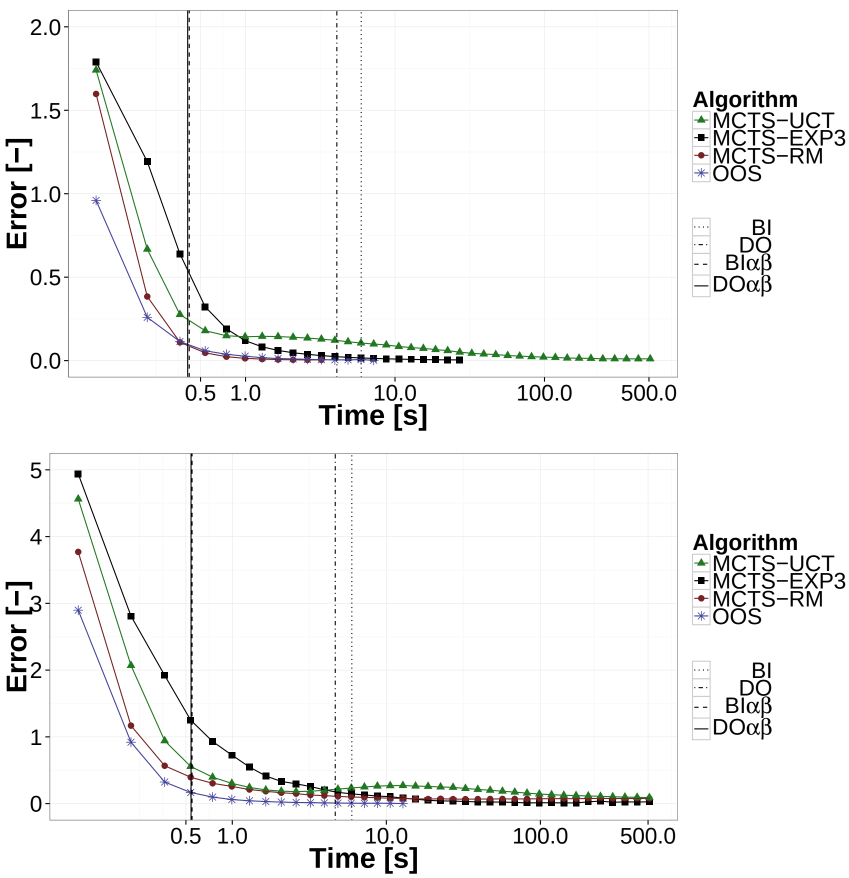
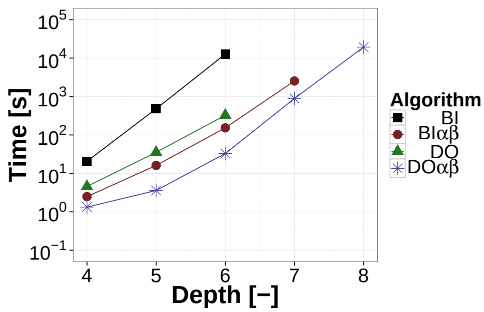


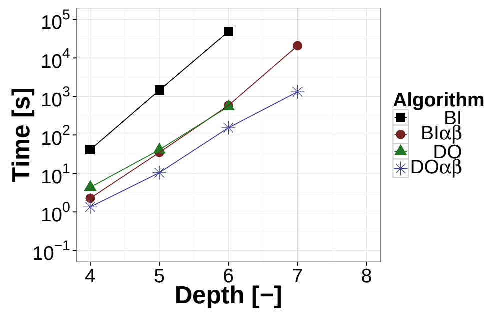
Figure 13: Convergence of the sampling algorithms on Goofspiel with 4 cards and chance nodes. The vertical lines correspond to the computation times for the exact algorithms. (Top) Goofspiel with win-tie-lose utility values; (Bottom) Goofspiel with point utilities.

6.4.2. Pursuit-Evasion Games

The results on pursuit-evasion games show more significant improvement when comparing DO$\alpha\beta$ and BI$\alpha\beta$ (see Figure 14). In all settings, DO$\alpha\beta$ is significantly the fastest. When we compare the performance on a 5×5 graph with depth set to 6, BI evaluates more than 4.9×10^7 nodes taking more than 13 hours. On the other hand, BI$\alpha\beta$ evaluates on average $42,001$ nodes taking almost 10 minutes (584 seconds). Interestingly, the benefits of a pure integration with alpha-beta search is not that helpful in this game. This is apparent from the results of DO algorithm that evaluates less than 2×10^6 nodes but it takes slightly over 9 minutes on average (547 seconds).



(a)



(b)

Figure 14: Running times of exact algorithms on pursuit-evasion games with an increasing number of moves: subfigure (a) depicts the results on 4×4 grid graph, (b) depicts results for 5×5 grid.

Finally, DO$\alpha\beta$ evaluates only $6,692$ nodes and it takes the algorithm less than 3 minutes.

Large parts of these pursuit-evasion games can be solved by the serialized alpha-beta algorithms. These parts typically correspond to clearly winning, or clearly losing positions for a player; hence, the serialized alpha-beta algorithms are able to prune a substantial portion of the space. However, since there are only two pursuit units, it is still necessary to use mixed strategies for a final coordination (capturing the evader close to edge of the graph), and thus mixing strategy occurs near the end of the game tree. Therefore, serialized alpha-beta is not able to solve all subgames, while double-oracle provides additional pruning since many of the actions in the subgames are leading to the same outcome and not all of them required finding equilibrium strategies. This leads to additional reductions in the computation time for DO$\alpha\beta$ compared to BI$\alpha\beta$ and all the other algorithms.

We now turn to the convergence of the sampling algorithms. In terms of the number of iterations per second, again RM was the fastest and OOS the second fastest with similar performance as in Goofspiel. UCT achieved slightly less (1.7×10^5 iterations per second), and Exp3 only 2.6×10^4 iterations. The results are depicted in Figure 15 for the smaller, 4×4 graph and 4 moves for each player (note again the logarithmic horizontal scale). The starting positions were selected such that there does not exist a pure NE strategy in the game. The results again show that OOS is overall the fastest out of all sampling algorithms. During the first iterations, RM preforms sim-

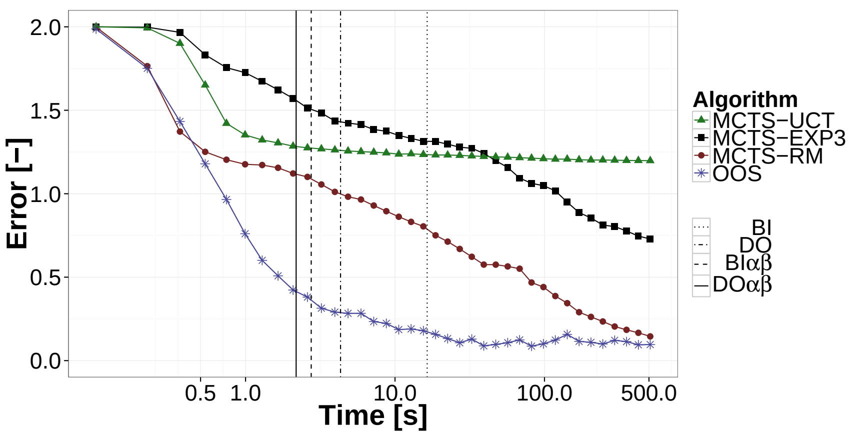


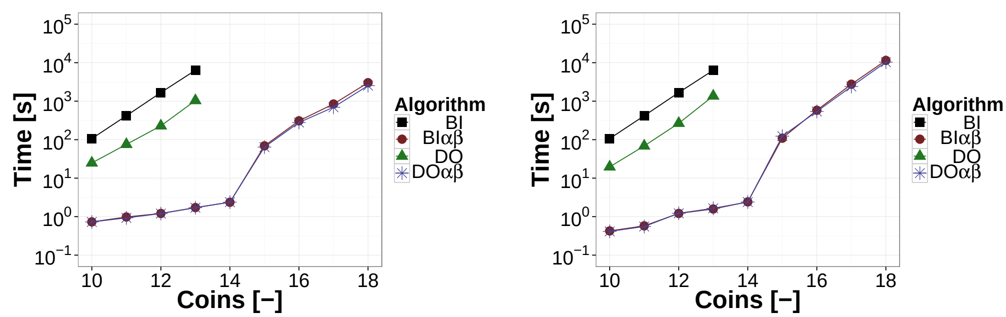
Figure 15: Convergence of the sampling algorithms on a pursuit-evasion game, on a 4×4 graph, with depth set to 4. The vertical lines correspond to the computation times for the exact algorithms.

ilarly, however, OOS is able to maintain its convergence rate, and RM starts converging more slowly. UCT again converges to an exploitable strategy with error 1.16 at best in the time limit of 500 seconds ($C = 2$). Finally, Exp3 is converging even more slowly than in Goofspiel. The main difference between the games is the size of the branching factor for the second player (the pursuer controls two simultaneously moving units), which can cause more difficulties for the sampling algorithms to estimate good strategies.

As before, the vertical lines represent the times for the exact algorithms. In a pursuit-evasion game of this setting, DO$\alpha\beta$ is slightly faster and finishes first in 2.77 seconds, following by BI$\alpha\beta$ (2.89 seconds), DO (5.48 seconds), and BI (12.5 seconds).

6.4.3. Oshi-Zumo

Many instances of the Oshi-Zumo game have Nash equilibria in pure strategies regardless of the type of the utility function. Although this does not hold for all the instances, the sizes of the subgames with pure NE are rather large and cause a dramatic computation speed-up for both algorithms using the serialized alpha-beta search. If the game does not have equilibria in pure strategies, the mixed strategies are still required only near the root node and large end-games are solved using alpha-beta search. Note that this is different than in the pursuit-evasion games, where mixed strategies were necessary close to the end of the game tree. Figure 16 depicts the



(a) (b)

Figure 16: Running times of the exact algorithms on Oshi-Zumo with K set to 4 and an increasing number of coins: subfigure (a) depicts the results for binary utilities, (b) depicts the results with point utilities.

results with the parameter K set to 4 and for two different settings of the utility function[6]; either win-tie-lose utilities (left subfigure) or point difference utilities (right subfigure). In both cases, the graphs show the breaking points when the game stops having an equilibrium in pure strategies ($\geq$ 15 coins for each player). The advantage of BI$\alpha\beta$ and DO$\alpha\beta$ algorithms that exploit the serialized variants of alpha-beta algorithms is dramatic. We can see that both BI and DO scale rather badly. The algorithms were able to scale up to 13 coins in a reasonable time. For setting with $K = 4$ and 13 coins, it takes almost 2 hours for BI to solve the game (the algorithm evaluates 1.5×10^7 nodes) regardless of the utility values. DO improves the performance (the algorithm evaluates 2.8×10^6 nodes in 17 minutes for win-tie-lose utilities; the performance is slightly worse for point utilities: 5×10^6 nodes in 23 minutes). Both BI$\alpha\beta$ and DO$\alpha\beta$, however, solved a single alpha-beta search on each serialization finding a pure NE. Therefore, their performance is identical and it takes around 1.5 seconds to solve the game for both types of utilities. Although with an increasing number of coins the algorithms BI$\alpha\beta$ and DO$\alpha\beta$ need to find a mixed Nash equilibrium, their performance is very similar for both types of utilities. As expected, the case with point utilities is more challenging and the algorithms scale worse – for 18 coins both algorithms solve the game with win-tie-lose utilities in approximately 1 hour (BI$\alpha\beta$ in

[6]We have also performed the same experiments with K set to 3, but the conclusions were the same as in case $K = 4$.

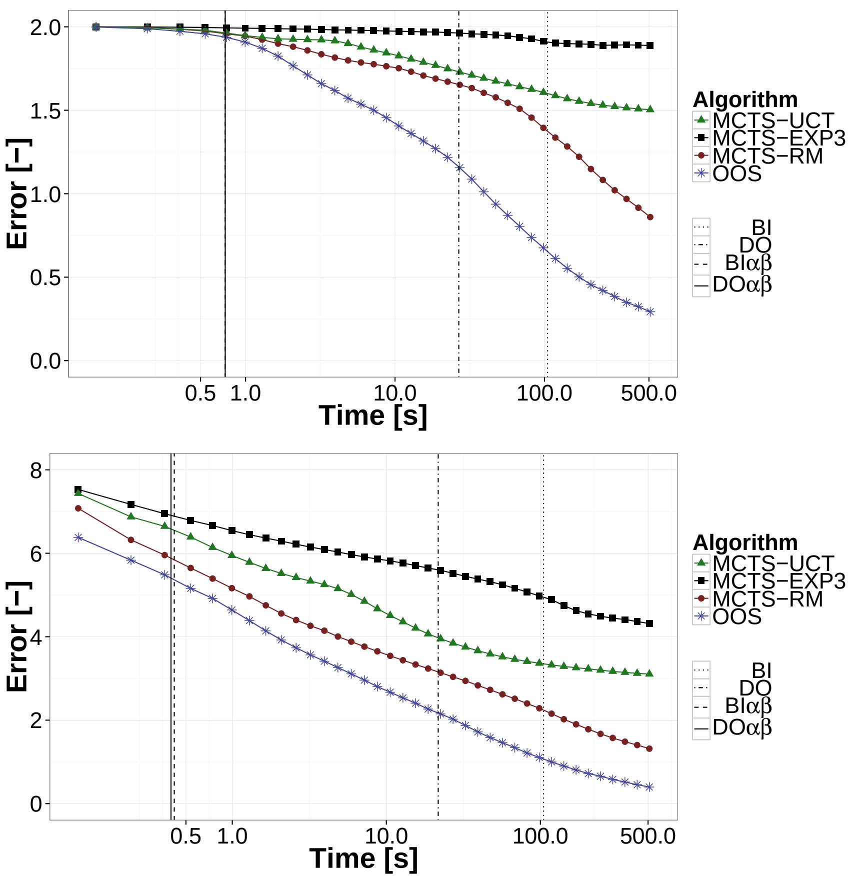
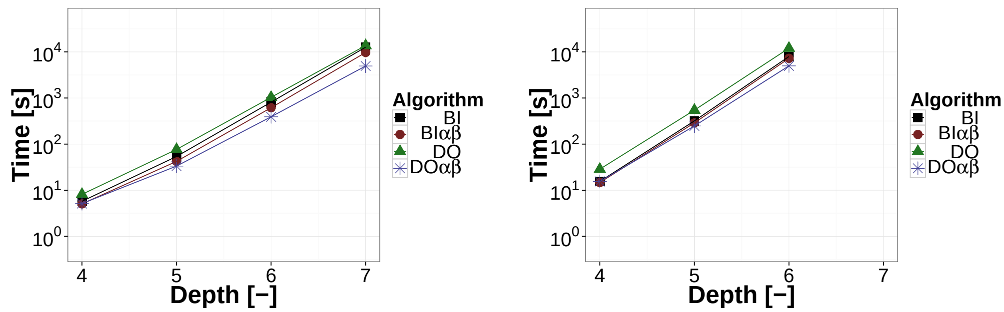


Figure 17: Convergence of the sampling algorithms on Oshi-Zumo game with 10 coins, $K = 3$, and $M = 1$. The vertical lines correspond to the computation times for the exact algorithms. (Top) Oshi-Zumo with win-tie-lose utility values; (Bottom) Oshi-Zumo with point utilities.

50 minutes, DO$\alpha\beta$ in 64). It takes the algorithms around 3 hours to solve the case with point utilities (BI$\alpha\beta$ in 191 minutes, DO$\alpha\beta$ in 172 minutes).

Turning to the sampling algorithms reveals that the game is difficult to approximate even in the win-tie-lose setting. Figure 17 depicts the results for the observed convergence rates of the sampling algorithms for the game with 10 coins, K set to 3 and the minimum bid set to 1. This is an easy game for DO$\alpha\beta$ and BI$\alpha\beta$ with a pure NE and both of these algorithms are able to solve the game in less than a second (0.73). However, due to a large branching factor for both players (10 actions at the root node for each



(a) (b)

Figure 18: Running times of the exact algorithms on randomly generated games with increasing depth: subfigure (a) depicts the results with branching factor set to 4 actions for each player, (b) depicts the results with branching factor 5.

player) all sampling algorithms converge extremely slowly. The performance of the algorithms in terms of iterations per second is similar to the previous games, however, OOS is slightly better in this case with 1.9×10^5 iterations per second compared to the RM with 1.6×10^5 iterations per second.

As before, OOS is the best converging algorithm, however, in a given time limit (500 seconds) the reached error was only slightly below 0.3 (0.29). On the other hand, all of the other sampling algorithms perform significantly worse – RM ends with error slightly over 1, UCT ($C = 2$) with 1.50, and Exp3 with 1.88. This confirms our findings from the previous experiment that increasing the branching factor slows down the convergence rate. Secondly, since there is a pure Nash equilibrium in this particular game configuration, the convergence of the algorithms is also slower since they essentially mix the strategy during the iterations in order to explore the unvisited parts of the game tree. Since none of the sampling algorithms can directly exploit this fact, their performance in offline solving of games like Oshi-Zumo is not compelling. On the other hand, the existence of pure NE explains the better performance of UCT compared to Exp3 that is forced to explore more broadly. Moreover, the convergence takes even more time in the point utility case, since the range of the utility values is larger. OOS is again the fastest and converges to error 0.45 within the time limit, RM to 1.41, UCT ($C = 4$) to 3.1, and Exp3 to 3.7.

6.4.4. Random Games

In the first variant of the randomly generated games we used games with utility values randomly drawn from a uniform distribution on $[0, 1]$. Such games represent an extreme case, where neither alpha-beta nor the double-oracle algorithm can save much computation time, since each action can lead to arbitrarily good or bad terminal states. In these games, BI is typically the fastest. Even though both BI$\alpha\beta$ and DO$\alpha\beta$ evaluate marginally fewer nodes (less than 90%), the overhead of the algorithms (repeated computations of the serialized alpha-beta algorithm, repeatedly solving linear programs, etc.) causes a slower run time performance in this case.

However, completely random games are rarely instances that need to be solved in practice. The situation changes, when we use the intuition of good and bad moves and thus add correlation to the utility values. Figure 18 depicts the results for two different branching factors 4 and 5 for each player and increasing depth. The results show that DO$\alpha\beta$ outperforms all remaining algorithms, although the difference is rather small (still statistically significant). On the other hand, DO without serialized alpha-beta is not able to outperform BI. This is most likely caused by a larger number of undominated actions that forces the double-oracle algorithm to enumerate most of the actions in each state. Moreover, this is also demonstrated by the performance of BI$\alpha\beta$ that is only slightly better compared to BI.

The fact that serialized alpha-beta is less successful in randomly generated games is noticeable also when comparing the number of evaluated nodes. For the case with branching factor set to 4 for both players and depth 7, BI evaluates almost 1.8×10^7 nodes in almost 3.5 hours, while BI$\alpha\beta$ evaluates more than 1×10^7 nodes in almost 3 hours. DO evaluates even more nodes compared to BI$\alpha\beta$ (1.2×10^7) and it is slower compared to both BI and BI$\alpha\beta$. Finally, DO$\alpha\beta$ evaluates 2×10^6 nodes on average and it takes the algorithm slightly over 80 minutes.

Figure 19 depicts the results for convergence of the sampling algorithms for a random game with correlated utility values, branching factor set to 4 and depth 5. The number of iterations per second is similar to the situation in Goofspiel, with Exp3 being the exception able to achieve more than 6.5×10^4 iterations per second, which is still the lowest number of iterations. Interestingly, there is a much less difference between the performance of the sampling algorithms in these games. Since these games are generally more mixed (i.e., NE require to use mixed strategies in many states of the games),

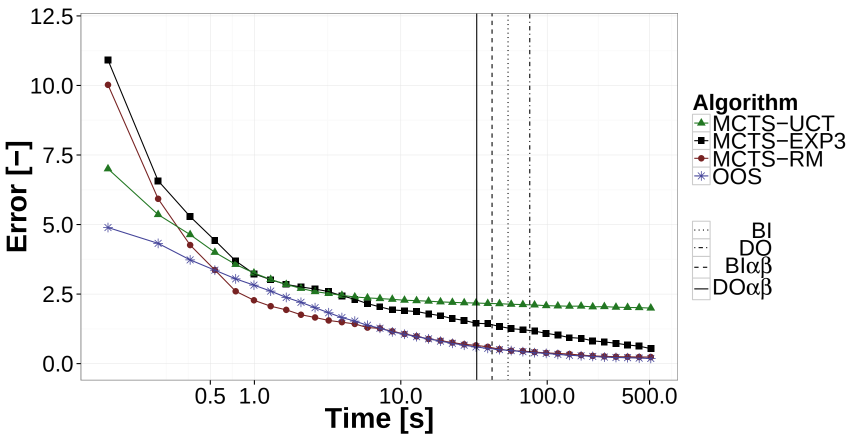
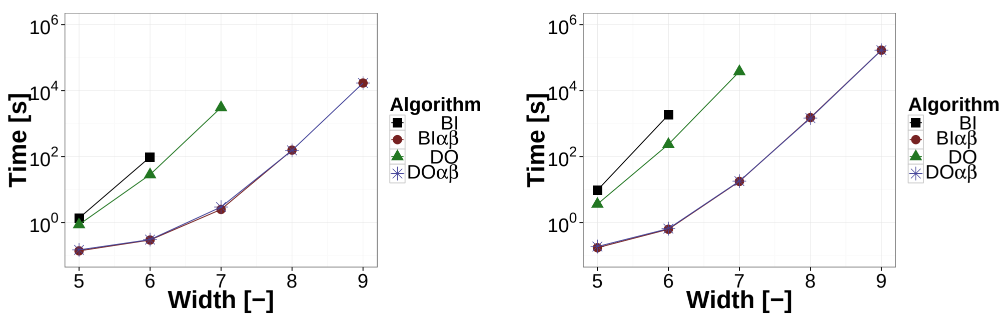


Figure 19: Convergence of the sampling algorithms on a random game with branching factor 4 and depth 5. The vertical lines correspond to the computation times for the exact algorithms.



(a)

(b)

Figure 20: Running times of the exact algorithms on Tron with increasing *width* of the grid graph: subfigure (a) depicts the results with *height* of the graph set to $width - 1$, (b) depicts the results with $height = width$.

they are much more suitable for the sampling algorithms. OOS can be considered the winner in this setting, however, the performance of RM is very similar. Again, since the game is more mixed, Exp3 outperforms UCT in the longer run. The exploration constant for UCT was set to 12 due to a larger utility variance in this setting.

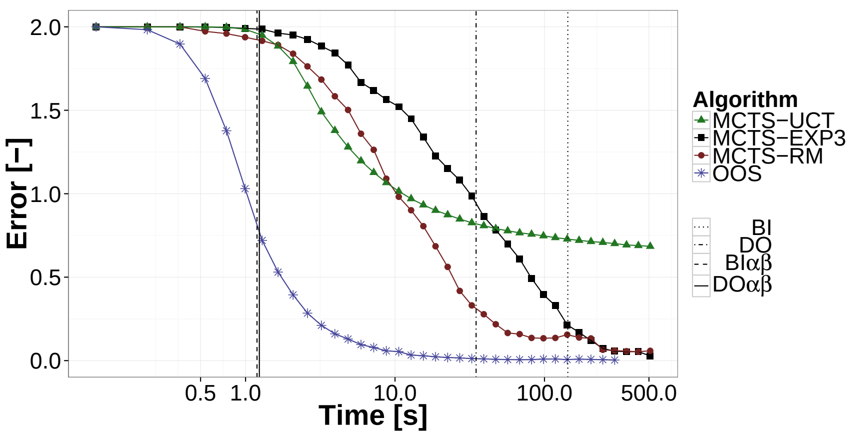


Figure 21: Convergence comparison of different sampling algorithms on Tron on grid 5×6. The vertical lines correspond to the computation times for the exact algorithms.

6.4.5. Tron

Performance of the exact algorithms in Tron is affected by the fact that pure NE exist in all smaller instances (the results are depicted for two different ratios of dimensions of the grid in Figure 20). Therefore, BI$\alpha\beta$ and DO$\alpha\beta$ are essentially the same since serialized alpha-beta is able to solve the game. Moreover, since the size of the game increases dramatically with the increasing size of the grid (the longest branch of the game tree has $(0.5 \cdot w \cdot l - 1)$ joint actions, where w and l are the dimensions of the grid), the performance of standard BI is very poor. While BI is able to solve the grid 5×6 in 96 seconds, it takes around 30 minutes to solve the 6×6 grid. By comparison, DO solves the 6×6 instance in 235 seconds, and both BI$\alpha\beta$ and DO$\alpha\beta$ in 0.6 seconds. BI$\alpha\beta$ and DO$\alpha\beta$ scale much better and the largest graph these algorithms solved had size 9×9 taking almost 2 days to solve.

The size of the game tree in Tron also causes a slow convergence for the sampling algorithms. This is apparent also in the number of iterations that is lower than before. OOS is the fastest performing 1.3×10^5 iterations per second, RM achieves 1.2×10^5, UCT only 8×10^4, and Exp3 is again the slowest with 7.8×10^4 iterations per second. Figure 21 depicts the results for the grid 5×6. Consistently with the previous results, OOS performs the best and it is able to converge very close to an exact solution in 300 seconds. Similarly, both RM and Exp3 are again eventually able to converge to a very small error, however, it takes them more time and in the time limit they achieve error 0.05, or 0.02 respectively. Finally, UCT ($C = 5$) performs reasonably well during the first 10 seconds, where the exploitability is better than both RM and Exp3. This is most likely due to the existence of pure NE, however, the length of the game tree prohibits UCT from converging and the best error the algorithm was able to achieve in the time limit was equal to 0.68.

6.4.6. Summary of the Offline Equilibrium Computation Experiments

The offline comparison of the algorithms offer several conclusions. Among the exact algorithms, DO$\alpha\beta$ is clearly the best algorithm, since it typically outperforms all other algorithms (especially in pursuit-evasion games and random games). Although for smaller games (e.g., Goofspiel with 5 cards) BI$\alpha\beta$ can be slightly faster, this difference is not significant and DO$\alpha\beta$ is never significantly slower compared to BI$\alpha\beta$.

Among the sampling algorithms, OOS is the clear winner since it is often able to quickly converge to a very small error and significantly outperforms all variants of MCTS. On the other hand, comparing OOS and DO$\alpha\beta$, the exact DO$\alpha\beta$ algorithm is always faster and it is able to find an exact solution much faster compared to OOS. Moreover, DO$\alpha\beta$ has significantly lower memory requirements since it is a depth-first search algorithm and does not use any form of global cache, while OOS iteratively constructs the game tree in memory.

6.5. Online Search

We now compare the performance of the algorithms in head-to-head matches in the same games as in the offline equilibrium computation experiments, but we use much larger instances of these games. Each algorithm has a strictly limited computation time per move set to 1 second. After this time, the algorithm outputs an action to be played in the current game state, receives information about the action selected by its opponent, and the game proceeds to the next state. As described in Section 5, each algorithm keeps results of previous computations and does not start from scratch in the next state. We have also performed a large set of experiments with 5 seconds of computation time per move, however, the results are very similar to the results with 1 second per move. Therefore, we presents the results with 1 second in detail and only comment on the 5-second results where the additional time leads to an interesting difference.

We compare all of the approximative sampling algorithms and DO$\alpha\beta$ as a representative of backward induction algorithms, because it was clearly the fastest algorithm in all of the considered games. Finally, we also include a random player (denoted RAND) into the tournament to confirm that the algorithms choose much better strategies than the simple random game play. We report expected rewards and win rates of the algorithms, in which a tie counts as half of a win. The parameters of the algorithms are tuned for each domain separately. We first present the comparison of different algorithms and we discuss the influence of the parameters in Subsection 6.5.6.

In this subsection, we show cross tables of each algorithm (in each row) matched up against each competitor algorithm (in each column). Each entry represents a mean of at least 1000 matches with the half of the width of the 95% confidence interval show in parentheses, e.g., 52.9(0.3) refers to 52.9% $\pm$ 0.3%. The result shown is the win rate for the row player, so as an example in the standard game of Goofspiel (top of Table 1) DO$\alpha\beta$ wins 67.2%$\pm$1.4% of games against the random player. All evaluated games except the pursuit-evasion game are symmetric from the perspective of the first and the second player. We made even the random games symmetric by always playing matches on the same game instance in pairs with alternating players' positions. However, for easier comparison of the algorithms, we mirror the same results to both fields corresponding to a pair of players in the cross tables.

6.5.1. Goofspiel

In the head-to-head comparisons, our focus is primarily on the standard Goofspiel with 13 cards and chance nodes. Additionally, for the sake of consistency with the offline results, we also evaluate the variant with a fixed known sequence of cards. The full game has more than 2.4×10^{29} terminal states and the variant with a fixed sequence has still more than 3.8×10^{19} terminal states. The results are presented in Table 1, where the top table shows the win rates of the algorithms in the full game and the other two tables show the win rates and the expected number of points gained by the algorithms in the game with a fixed point card sequence. The results for the fixed card sequence are means over 10 fixed random sequences. For each table, the algorithms were set up to optimize the presented measure (i.e., win rate or points) and the exploration parameters were tuned to the values presented in the header of the table.

First, we can see that finding a good strategy in Goofspiel is difficult for

Goofspiel: 13 cards, stochastic sequence of cards, win rate

	DO$\alpha\beta$	OOS(0.2)	UCT(0.6)	EXP3(0.3)	RM(0.1)	RAND
DO$\alpha\beta$	•	26.6(2.7)	36.0(2.9)	26.1(2.7)	25.9(2.7)	67.2(1.4)
OOS	73.4(2.7)	•	**51.2(2.1)**	52.5(2.2)	47.5(3.0)	81.4(1.7)
UCT	64.0(2.9)	48.8(2.1)	•	55.6(2.1)	**49.7(3.0)**	77.3(1.8)
EXP3	73.8(2.7)	47.5(2.2)	44.4(2.1)	•	41.1(3.0)	**86.1(1.5)**
RM	**74.1(2.7)**	**52.5(3.0)**	50.3(3.0)	**58.9(3.0)**	•	85.2(2.2)
RAND	32.8(1.4)	18.6(1.7)	22.7(1.8)	13.9(1.5)	14.8(2.2)	•

Goofspiel: 13 cards, known sequence of cards, win rate

	DO$\alpha\beta$	OOS(0.3)	UCT(0.8)	EXP3(0.2)	RM(0.1)	RAND
DO$\alpha\beta$	•	28.2(2.8)	35.0(2.9)	30.1(2.8)	31.5(2.8)	67.2(2.9)
OOS	**71.8(2.8)**	•	46.2(3.0)	51.8(3.0)	**49.6(3.0)**	83.8(2.3)
UCT	65.0(2.9)	**53.8(3.0)**	•	**57.1(2.9)**	48.6(2.9)	79.5(2.5)
EXP3	70.0(2.8)	48.2(3.0)	42.9(2.9)	•	46.5(3.0)	**85.8(2.1)**
RM	68.5(2.8)	50.4(3.0)	**51.4(2.9)**	53.5(3.0)	•	84.2(2.2)
RAND	32.8(2.9)	16.2(2.3)	20.5(2.5)	14.2(2.1)	15.8(2.2)	•

Goofspiel: 13 cards, known sequence of cards, point utilities

	DO$\alpha\beta$	OOS(0.3)	UCT(0.8)	EXP3(0.2)	RM(0.1)	RAND
DO$\alpha\beta$	•	-7.74(0.94)	-8.89(0.91)	-6.45(0.94)	-7.88(0.96)	6.67(0.99)
OOS	7.74(0.94)	•	1.19(0.78)	3.27(0.82)	**0.35(0.76)**	14.42(0.96)
UCT	**8.89(0.91)**	-1.19(0.78)	•	1.72(0.80)	-1.94(0.73)	13.30(1.00)
EXP3	6.45(0.94)	-3.27(0.82)	-1.72(0.80)	•	-5.02(0.79)	**14.79(0.97)**
RM	7.88(0.96)	**-0.35(0.76)**	**1.94(0.73)**	**5.02(0.79)**	•	14.20(0.98)
RAND	-6.67(0.99)	-14.42(0.96)	-13.30(1.00)	-14.79(0.97)	-14.20(0.98)	•

Table 1: Results of head-to-head matches in Goofspiel variants with exploration parameter settings indicated in the table headers.

all the algorithms. This is noticeable from the results of the RAND player, that performs reasonably well (RAND typically loses almost every match in all the remaining game domains). Next, we analyze the results of the DO$\alpha\beta$ algorithm compared to the sampling algorithms. The results show that even though DO$\alpha\beta$ uses a domain-specific heuristic evaluation function, it does not win significantly against any of the sampling algorithms that do not use any domain knowledge. The difference is always statistically significant with a large margin. When optimizing win percentage, DO$\alpha\beta$ loses the least against UCT while in optimizing the expected reward, UCT performs significantly best. The performance of the other sampling algorithms is very similar against DO$\alpha\beta$, with Exp3 winning the least in the reward optimization.

We compare the sampling algorithms in the game variants in the order of the presented tables. The differences in the performance of the sampling algorithms are relatively small between each other. They are more noticeable mainly against the weaker players, which are outperformed by all sampling algorithms. In the game with stochastic point card sequence, OOS, UCT and RM make approximately 10×10^3 iterations in the 1 second time limit in the root of the game. Exp3 is slightly slower with 8×10^3 iterations. The best algorithm in this game variant is RM, which wins against all other sampling algorithms and wins most often against DO$\alpha\beta$ and Exp3. The second best algorithm is OOS, which loses only against RM and Exp3 is the weakest algorithm losing against all other sampling algorithms.

The sampling algorithms in the second game variant (without chance) perform the same number of samples as in the first variant, with the exception of UCT, which performs 12×10^3 iterations per second. However, they each build a considerably deeper search tree, since the game tree is less wide. The exploration parameters were tuned to slightly larger numbers, which indicate that more exploration is beneficial in smaller games. The results are similar to the previous game variant. RM is still winning against all opponents, but it is not able to win more often against weaker players, which is consistent with playing close to a Nash equilibrium. UCT loses only against RM in this variant and it significantly outperforms OOS and Exp3. This indicates that UCT was able to better focus on the relevant part of the smaller game, which is supported also by a larger number of simulations, which can be caused by shorter random simulations after leaving the part of the search tree stored in memory.

When the players optimize the expected point difference, the differences between the algorithms are larger. We can see that RM and OOS perform significantly better than UCT and Exp3. OOS wins against all opponents and RM loses only against OOS. An important reason behind the decrease of the performance of Exp3 is that after normalizing the reward to unit interval, the important differences in values for reasonably good strategies become much smaller, which slows down the learning of the algorithm. UCT compensates the range of the rewards by the choice of the exploration parameter, but different nodes would benefit from different exploration parameters, which causes more inefficiencies with more variable rewards. An important advantage of OOS and RM is that their behavior is practically independent of the utility range.

In summary, RM is the only algorithm that did not lose significantly against any other sampling algorithm in any of the game variants and it often wins significantly. Exp3 is overall the weakest algorithm, losing to all other algorithms in all Goofspiel variants. Interestingly, Exp3 always performs the best against the random player, which indicates a slower convergence against more sophisticated strategies.

Oshi-Zumo: 50 coins, $K = 3$, win rate

	DO$\alpha\beta$	OOS(0.2)	UCT(0.4)	EXP3(0.8)	RM(0.1)	RAND
DO$\alpha\beta$	•	**79.2(2.5)**	**77.6(2.6)**	**84.8(2.2)**	**84.0(2.3)**	98.8(0.5)
OOS	20.9(2.5)	•	27.7(2.6)	57.1(2.1)	51.2(2.1)	98.9(0.4)
UCT	**22.4(2.6)**	72.3(2.6)	•	83.0(2.0)	70.3(2.6)	**99.9(0.2)**
EXP3	15.2(2.2)	42.9(2.1)	17.0(2.0)	•	44.5(2.8)	98.5(0.5)
RM	16.0(2.3)	48.8(2.1)	29.6(2.6)	55.5(2.8)	•	99.0(0.4)
RAND	1.2(0.5)	1.1(0.4)	0.1(0.2)	1.5(0.5)	1.0(0.4)	•

Oshi-Zumo: 50 coins, $K = 3$, point utilities

	DO$\alpha\beta$	OOS(0.2)	UCT(0.4)	EXP3(0.8)	RM(0.1)	RAND
DO$\alpha\beta$	•	**2.33(0.19)**	**2.27(0.20)**	3.62(0.10)	**2.85(0.17)**	3.65(0.09)
OOS	-2.33(0.19)	•	-0.53(0.19)	3.46(0.10)	0.25(0.20)	3.87(0.05)
UCT	**-2.27(0.20)**	0.53(0.19)	•	**3.68(0.07)**	0.58(0.17)	**3.93(0.02)**
EXP3	-3.62(0.10)	-3.46(0.10)	-3.68(0.07)	•	-3.53(0.09)	1.31(0.17)
RM	-2.85(0.17)	-0.25(0.20)	-0.58(0.17)	3.53(0.09)	•	3.87(0.04)
RAND	-3.65(0.09)	-3.87(0.05)	-3.93(0.02)	-1.31(0.17)	-3.87(0.04)	•

Oshi-Zumo: 50 coins, $K = 3$, win rate, evaluation function

	DO$\alpha\beta$	OOS(0.3)	UCT(0.8)	EXP3(0.8)	RM(0.1)	RAND
DO$\alpha\beta$	•	63.0(2.1)	11.8(1.3)	52.9(2.2)	61.7(2.1)	98.6(0.5)
OOS	37.0(2.1)	•	24.8(1.9)	33.4(2.0)	43.6(2.1)	99.6(0.3)
UCT	**88.2(1.3)**	**75.2(1.9)**	•	**80.5(1.7)**	**71.1(2.0)**	**99.8(0.2)**
EXP3	47.1(2.2)	66.6(2.0)	19.5(1.7)	•	58.7(2.1)	98.7(0.5)
RM	38.3(2.1)	56.4(2.1)	**28.9(2.0)**	41.3(2.1)	•	99.6(0.3)
RAND	1.4(0.5)	0.4(0.3)	0.2(0.2)	1.3(0.5)	0.4(0.3)	•

Table 2: Results of head-to-head matches in Oshi-Zumo variants with exploration parameter settings indicated in the header. In the first two tables only DO$\alpha\beta$ uses a heuristic evaluation function and in the third table all algorithms use the evaluation function.

6.5.2. Oshi-Zumo

In Oshi-Zumo, we use the setting with 50 coins, $2 \cdot 3 + 1 = 7$ fields of the board (i.e., $K = 3$), and the minimal bet of 1. The size of the game is large with strictly more than 10^{15} terminal states and 50 actions for each player in the root.

The results are depicted in Table 2. As in the case of Goofspiel, we show the results for both the win rate as well as the point utilities. Moreover, our evaluation function in Oshi-Zumo is much more accurate than the one in Goofspiel and DO$\alpha\beta$ is clearly outperforming all sampling algorithms when they do not use any domain specific knowledge. Therefore we also run experiments where the sampling algorithms also use an evaluation function instead of random rollout simulations.

In the offline experiment (Figure 17), none of the sampling algorithms were able to converge anywhere close to the equilibrium in a short time. Moreover, the game used in the offline experiments was orders of magnitude smaller (there were only 10 coins for each player). In spite of the negative results in the offline experiments, all sampling algorithms are able to find a

	DO$\alpha\beta$	OOS(0.1)	UCT(1.5)	EXP3(0.6)	RM(0.3)	RAND
DO$\alpha\beta$	•	57.4(2.9)	**49.6(2.8)**	53.4(2.8)	51.3(2.8)	88.8(1.8)
OOS	42.6(2.9)	•	33.5(2.5)	43.5(2.7)	42.5(2.8)	85.0(2.4)
UCT	**50.4(2.8)**	**66.5(2.5)**	•	**67.4(2.5)**	**55.7(2.6)**	95.9(1.2)
EXP3	46.6(2.8)	56.5(2.7)	32.6(2.5)	•	42.9(2.7)	**96.0(1.1)**
RM	48.7(2.8)	57.5(2.8)	44.3(2.6)	57.1(2.7)	•	93.1(1.5)
RAND	11.2(1.8)	15.0(2.4)	4.1(1.2)	4.0(1.1)	6.9(1.5)	•

Table 3: Win-rate in head-to-head matches of Random games with 5 actions for each player in each move and depth of 15 moves.

reasonably good strategy. UCT is clearly the strongest sampling algorithm in all variants. In the win rate setting, the strongest opponent of UCT among the sampling algorithms is RM (UCT wins 70.3% of games), followed by OOS performing only slightly worse (UCT wins 72.3% of games). Finally, Exp3 is clearly the weakest of all sampling algorithms. A possible reason may be that Exp3 manages to perform around 2.5×10^3 iterations per second in the root, while the other algorithms perform ten times more. This is caused by the quadratic dependence of its computational complexity on the number of actions, which is relatively high in this game. The situation remains similar when the algorithms optimize the point utilities.

We now turn to the experiments with the evaluation function, the results of which are presented in the third table of Table 2. The results show that the quality of play of all sampling algorithms is significantly improved. With this modification, UCT already significantly outperforms all algorithms including DO$\alpha\beta$. DO$\alpha\beta$ is the second best and still winning over the remaining sampling algorithms. Exp3 benefits from the evaluation function more than OOS and RM, which are relatively weaker with the evaluation function.

The reason why UCT performs well in this game is that the game mostly requires pure strategies, rather than precise mixing between multiple strategies (see Subsection 6.2). UCT is able to quickly disregard other actions, if a single action is optimal. So, the evaluation function generally helps every algorithm, but can make significant changes in ranking of the algorithms.

6.5.3. Random Games

The next set of matches was played on 10 different random games with each player having 5 actions in each stage and depth 15. Hence, the game has more than 9.3×10^{20} terminal states. In order to compute the win-rates as in the other games, we use the sign of the utility value defined in Subsection 6.2. The results are presented in Table 3.

Tron: 13x13 grid, win rate

	DO$\alpha\beta$	OOS(0.1)	UCT(0.6)	EXP3(0.5)	RM(0.1)	RAND
DO$\alpha\beta$	•	**78.2(2.0)**	**53.8(2.0)**	**66.6(2.3)**	**65.0(2.2)**	**98.6(0.5)**
OOS	21.8(2.0)	•	29.4(2.2)	46.1(1.8)	38.0(2.2)	97.2(0.5)
UCT	**46.2(2.0)**	70.6(2.2)	•	64.8(2.2)	57.0(2.1)	98.0(0.6)
EXP3	33.4(2.3)	53.9(1.8)	35.1(2.2)	•	44.3(2.3)	97.7(0.5)
RM	35.0(2.2)	62.0(2.2)	43.0(2.1)	55.7(2.3)	•	97.6(0.9)
RAND	1.4(0.5)	2.9(0.5)	2.0(0.6)	2.3(0.5)	2.4(0.9)	•

Tron: 13x13 grid, win rate, evaluation function

	DO$\alpha\beta$	OOS(0.1)	UCT(2)	EXP3(0.1)	RM(0.2)	RAND
DO$\alpha\beta$	•	**50.2(1.3)**	42.7(1.5)	53.1(1.6)	46.3(1.6)	98.8(0.4)
OOS	49.8(1.3)	•	53.0(0.9)	**54.7(0.8)**	**52.2(0.8)**	**97.9(0.4)**
UCT	**57.3(1.5)**	47.0(0.9)	•	49.7(0.5)	46.7(0.6)	98.8(0.3)
EXP3	46.9(1.6)	45.3(0.8)	50.3(0.5)	•	45.8(0.6)	98.2(0.4)
RM	53.7(1.6)	47.8(0.8)	**53.3(0.6)**	54.2(0.6)	•	98.5(0.4)
RAND	1.2(0.4)	2.1(0.4)	1.2(0.3)	1.8(0.4)	1.5(0.4)	•

Table 4: Win-rate in head-to-head matches of Tron with random simulations (top) and evaluation function in the sampling algorithms (bottom).

The clearly best performing algorithm in this domain is UCT that significantly outperforms the other sampling algorithms, and ties with DO$\alpha\beta$ that uses a rather strong evaluation function. This is true even though UCT performs around 11×10^3 iterations per second, which is the least form all sampling algorithms. DO$\alpha\beta$ wins over all other sampling algorithms. OOS has the weakest performance in spite of good convergence results in the offline settings (see Figure 19). The reason is the quickly growing variance and decreasing number of samples in longer games, which we discuss in more details in Subsection 6.6. OOS performs 20×10^3 iterations per second and only around 3×10^3 of them actually update the regrets in the root. All the other iterations return with zero tail probability (x_i) in the root, which leads to no change in the regret values.

6.5.4. Tron

The large variant of Tron in our evaluation was played on an empty 13×13 board. The branching factor of this game is up to 4 for each player and its depth is up to 83 moves. This variant of Tron has more than 10^{21} terminal states[7]. The results are shown in Table 4.

The evaluation function in Tron approximates the situation in the game fairly well; hence, DO$\alpha\beta$ strongly outperforms all other algorithms when

[7]The number only estimates the number of possible paths when both players stay on their half of the board.

they do not use the evaluation function (top). Its win-rates are even higher with more time per move. UCT is the strongest opponent for DO$\alpha\beta$ – UCT loses 53.8% of matches and wins over all other sampling algorithms in mutual matches. This is again because of the low need for mixed strategies in this game (see Subsection 6.2). Again, OOS performs the worst despite its clearly fastest convergence on the smaller game variant in the offline setting due to the great depth of the game tree in this setting. It won only 21.8% matches against DO$\alpha\beta$ and 29.4% matches against UCT. In this game, the variance of the regret updates is likely not the key factor, since it is between 20 and 40. However, only 1×10^3 out of 12×10^3 iterations per second update regrets in the root.

The good performance of DO$\alpha\beta$ is consistent with the previous analysis in Tron where the best-performing algorithms, including the winner of the 2011 Google AI Challenge, were based on depth-limited minimax searches [84, 57].

As in the case of Oshi-Zumo, we also run the matches with the evaluation function in place of the random rollout simulation in the sampling algorithms. We present the results in the second table of Table 4. Using the evaluation function improves the performance of all sampling algorithms against DO$\alpha\beta$ and it decreases the differences in performance between each algorithm. The difference is most notable for OOS, since using the evaluation function strongly reduces the length of the game. In this setting, both RM and UCT outperform DO$\alpha\beta$. Interestingly, while UCT performs quite well against DO$\alpha\beta$ and wins 57.3% of matches, it is not winning against any other sampling algorithm. Even Exp3 which loses against all other algorithms manages to slightly outperform it. OOS practically ties with DO$\alpha\beta$, but it wins significantly against all sampling algorithms. RM loses to OOS, but wins significantly against all other algorithms.

6.5.5. Pursuit-Evasion Game

Finally, we compare the algorithms on the pursuit-evasion game on an empty 10×10 grid with 15 moves time limit and 10 different randomly selected initial positions of the units. The branching factor is at most 12, causing the number of terminal states to be less than 10^{16}.

The results in Table 5 show that the game is strongly biased towards the first player, which is the evader. The self-play results on the diagonal show that DO$\alpha\beta$ won over 81.5% matches against itself as the evader. Adding more computational time typically improves the play of the pursuer in self-play. This is caused by a more complex optimal strategy of the pursuer. This

	DO$\alpha\beta$	OOS(0.3)	UCT(0.8)	EXP3(0.5)	RM(0.1)	RAND
DO$\alpha\beta$	81.5(2.4)	89.1(1.9)	**61.8(3.0)**	**91.2(1.8)**	77.2(2.6)	**100.0(0.0)**
OOS(0.2)	77.5(2.6)	91.2(1.8)	57.8(3.1)	85.8(2.2)	79.3(2.5)	99.8(0.3)
UCT(1.5)	77.1(2.6)	**94.2(1.4)**	57.6(3.1)	88.9(1.9)	**82.2(2.4)**	**100.0(0.0)**
EXP3(0.2)	65.1(3.0)	92.1(1.7)	53.1(3.1)	83.9(2.3)	75.1(2.7)	99.8(0.3)
RM(0.1)	**81.8(2.4)**	92.7(1.6)	58.5(3.1)	86.7(2.1)	78.6(2.5)	99.8(0.3)
RAND	5.1(1.4)	28.8(2.8)	5.8(1.4)	1.7(0.8)	3.1(1.1)	71.1(2.8)

Table 5: Win-rate in head-to-head matches of pursuit-evasion games with time limit of 15 moves and 10×10 grid board. The evader is the row player and pursuer is the column player.

optimal strategy is more difficult to find due to a larger branching factor (recall that the pursuer controls two units) and the requirement for a more precise execution (a single move played incorrectly can cause an escape of the evader and can result in losing the game due to the time limit).

We first look at the differences in the performance of the algorithms on the side of the pursuer, which are more consistent. We need to compare the different columns, in which the pursuer tries to minimize the values. The clear winner is UCT that generally captures the evaders in approximately 40% of the matches. The second best pursuer is DO$\alpha\beta$ and the weakest is OOS that captures the non-random opponents in less than 10% of the cases.

The situation is less clear for the evader. Different algorithms performed best against different opponents. UCT was the best against OOS and RM, but DO$\alpha\beta$ was the best against UCT and Exp3. Exp3 is the weakest evader.

6.5.6. Parameter Tuning

The exploration parameters can have a significant influence on the performance of the algorithm. We choose the parameters individually for each domain by running mutual matches with a pre-selected fixed pool of opponents. This pool includes DO$\alpha\beta$ and each of the sampling algorithms with one setting of the parameter selected based on the results of the offline experiments. These values are 0.6 for OOS, 2 for UCT, 0.2 for Exp3 and 0.1 for RM. For each domain, we created a table such as the two examples in Table 6. We then picked the parameter for the final cross tables presented above as the parameter with the best mean performance against all the fixed opponents.

In the presented variant of Goofspiel, the choice of the exploration parameter has a rather large influence on the performance against DO$\alpha\beta$. This is often the case for weaker players. The selection of the exploration parameter for OOS has little effect on the mean performance, with a noticeable

Goofspiel: 13 cards, stochastic point card sequence

		DO$\alpha\beta$	OOS(0.6)	UCT(2)	EXP3(0.2)	RM(0.1)	Mean
OOS	0.5	73.8(2.7)	50.2(3.0)	54.4(4.2)	54.9(3.0)	49.4(3.0)	56.54
OOS	0.4	72.0(2.8)	50.5(3.0)	56.4(4.2)	54.1(3.0)	47.5(3.0)	56.1
OOS	0.3	73.0(2.7)	47.6(3.0)	58.4(4.2)	54.3(3.0)	48.0(3.1)	56.26
OOS	**0.2**	73.5(2.7)	50.2(3.0)	58.7(4.2)	54.3(3.0)	47.9(3.0)	**56.92**
OOS	0.1	70.2(2.8)	47.4(3.1)	53.4(4.3)	48.6(3.0)	43.9(3.0)	52.7
UCT	1.5	52.2(3.1)	45.4(3.0)	52.4(3.2)	53.9(3.9)	39.4(4.6)	48.66
UCT	1	52.5(3.0)	49.9(3.0)	58.3(3.2)	56.1(3.8)	43.1(4.6)	51.98
UCT	0.8	52.5(3.0)	51.1(3.0)	60.8(3.2)	59.7(3.8)	46.8(4.7)	54.18
UCT	**0.6**	54.2(3.0)	53.9(3.0)	61.2(3.1)	62.3(3.8)	46.6(4.7)	**55.64**
UCT	0.4	58.6(3.0)	54.9(3.0)	61.6(3.1)	58.6(3.8)	49.5(4.8)	55.04
EXP3	0.5	77.1(2.6)	42.6(3.0)	44.4(3.0)	47.4(3.0)	40.1(3.0)	50.32
EXP3	0.4	76.2(2.6)	44.8(3.0)	48.4(3.0)	49.5(3.0)	39.5(3.0)	51.68
EXP3	**0.3**	73.2(2.7)	44.5(3.0)	51.8(3.0)	51.1(3.0)	41.0(3.0)	**52.32**
EXP3	0.2	73.5(2.7)	47.2(3.0)	47.6(3.0)	50.0(3.0)	41.3(3.0)	51.92
EXP3	0.1	71.2(2.8)	44.9(3.0)	48.9(3.0)	51.2(3.0)	40.9(3.0)	51.42
RM	0.5	77.7(2.5)	44.9(3.0)	43.9(3.0)	46.9(3.0)	42.4(3.0)	51.16
RM	0.3	73.2(2.7)	49.3(3.0)	57.9(2.9)	53.9(3.0)	48.5(3.0)	56.56
RM	0.2	70.8(2.8)	50.7(3.0)	63.8(2.9)	57.8(3.0)	48.2(3.0)	58.26
RM	**0.1**	74.0(2.7)	54.1(3.0)	61.2(2.9)	58.1(3.0)	51.2(3.0)	**59.72**
RM	0.05	74.5(2.7)	51.6(3.0)	60.1(2.9)	59.0(3.0)	49.0(3.1)	58.84

Oshi-Zumo: 50 coins, $K = 3$, win rate, evaluation function

		DO$\alpha\beta$	OOS(0.6)	UCT(2)	EXP3(0.2)	RM(0.1)	Mean
OOS	0.5	35.3(2.9)	50.9(3.6)	28.5(3.3)	54.9(3.6)	43.7(3.5)	42.66
OOS	0.4	35.0(2.9)	56.0(3.6)	26.6(3.2)	56.1(3.6)	42.6(3.6)	43.26
OOS	**0.3**	36.5(3.0)	57.8(3.5)	27.7(3.2)	55.7(3.6)	44.8(3.6)	**44.5**
OOS	0.2	35.0(2.9)	53.1(3.6)	26.8(3.2)	54.1(3.6)	41.4(3.5)	42.08
OOS	0.1	34.6(2.9)	55.6(3.6)	24.1(3.1)	56.2(3.6)	43.0(3.6)	42.7
UCT	1.5	83.2(2.2)	74.0(3.8)	79.1(2.9)	87.4(2.9)	70.6(3.9)	78.86
UCT	1	83.8(2.1)	74.8(3.7)	81.4(2.7)	89.8(2.6)	68.8(4.0)	79.72
UCT	**0.8**	86.5(2.0)	77.9(3.6)	77.1(3.0)	89.2(2.7)	74.1(3.8)	**80.96**
UCT	0.6	89.4(1.8)	75.7(3.7)	54.9(3.9)	90.0(2.6)	74.1(3.7)	76.82
UCT	0.4	75.8(2.6)	75.0(3.7)	31.4(3.7)	89.8(2.6)	70.6(3.9)	68.52
EXP3	0.9	47.8(3.1)	68.2(2.8)	23.1(2.4)	67.2(2.8)	55.2(2.8)	52.3
EXP3	**0.8**	46.9(3.1)	68.4(3.6)	23.0(3.1)	74.2(3.4)	61.5(3.7)	**54.8**
EXP3	0.6	42.5(3.1)	67.6(3.7)	20.4(3.1)	65.4(3.7)	59.4(3.8)	51.06
EXP3	0.5	38.7(3.0)	60.9(3.8)	15.1(2.7)	64.7(3.7)	52.9(3.9)	46.46
EXP3	0.4	35.9(3.0)	57.5(3.9)	17.5(3.0)	64.1(3.8)	54.9(3.9)	45.98
RM	0.5	44.5(3.0)	41.1(3.5)	31.7(3.3)	49.4(3.6)	34.3(3.3)	40.2
RM	0.3	42.8(3.0)	52.1(3.5)	33.8(3.4)	61.2(3.5)	43.7(3.5)	46.72
RM	0.2	41.8(3.0)	55.7(3.6)	30.7(3.3)	59.2(3.5)	46.4(3.6)	46.76
RM	**0.1**	37.0(2.9)	58.1(3.5)	34.9(3.4)	57.6(3.6)	54.1(3.6)	**48.34**
RM	0.05	36.4(3.0)	59.6(3.5)	29.7(3.3)	59.3(3.5)	51.1(3.6)	47.22

Table 6: Sample parameter tuning tables for Goofspiel with stochastic point cards sequence and Oshi-Zumo.

drop in performance for 0.1. In UCT, less exploration is generally better, but the sudden drop of performance against Exp3 causes the optimum to be at 0.6. In Exp3, the optimal exploration parameter against DO$\alpha\beta$ would be even greater than 0.5, while the optimum against OOS would be 0.2. These kinds of inconsistencies are common with the Exp3 algorithm. In the mean over all opponents, the optimum is 0.3. With RM, the optimal exploration parameter against individual opponents stays around 0.1 and it is clearly the best value in the mean.

Parameter selection is generally more important when facing weaker players. The differences are more noticeable in matches against other algorithms, but since the optimal parameters vary depending on the different opponents, the mean performance presented in the last column does not vary much. OOS is consistently the least sensitive to different parameter settings, while the performance differences in the other algorithms from changing exploration strongly depends on the specific domains.

The differences between various parameter settings are larger in smaller games and mainly if an evaluation function is used. Consider the results for Oshi-Zumo in Table 6. For OOS, the exploration parameter of 0.3 is consistently the best against all opponents, with the exception of Exp3, which loses slightly more to OOS with exploration 0.4. However, the difference is far from significant even after 1000 matches. The differences in performance of UCT with different parameters are more often statistically significant. Overall, the best parameter is 0.8, even though the performance is significantly better against DO$\alpha\beta$ with smaller exploration and against UCT(2) with higher exploration. The best performance for Exp3 was surprisingly achieved with a very high exploration. The best of the tested values was 0.8, which means that 80% of the time, the next action to sample is selected randomly regardless of the collected statistics about move qualities. The higher values were consistently better for all opponents. RM seems to be quite sensitive to the parameter choice in this domain and the results for specific opponents are more inconclusive than for the other algorithms. When playing DO$\alpha\beta$, RM wins 7% more matches with parameter 0.3 than with the overall optimal 0.1. On the other hand, when playing OOS, an even smaller parameter value would be preferable.

The presented parameter tuning tables are representative of the behavior of the algorithms with different parameters. The choices of the optimal parameters generally depend much more on the domain than the selected opponent, but in some cases the optimal choice for one opponent is far from the optimum for another opponent. Especially with Exp3 and UCT, very different parameters are optimal for different domains. While in the presented results in Oshi-Zumo with evaluation function, 0.8 is best for Exp3, in Tron with evaluation function, the optimal parameter for Exp3 is 0.1. The range of optimal parameters is much smaller for OOS and RM, which were always between 0.1 and 0.3. This can be a notable advantage for playing previously unknown games without a sufficient time to tune the parameters for the specific domain.

6.5.7. Summary of the Online Search Experiments

Several conclusions can be made from the head-to-head comparisons of the algorithms in larger games. First, the fast convergence and low exploitability of OOS in the smaller variants of the games is not a very good predictor of its performance in the online setting. OOS was often not the best algorithm in the online setting. In random games and Tron without the evaluation function, it was the worst performing algorithm. We discuss the possible reasons in detail in Subsection 6.6.

Second, DO$\alpha\beta$ with a good evaluation function often wins over the sampling algorithms without a domain specific knowledge. This is not the case with a weaker evaluation function, as we can see in Goofspiel. Moreover, when the sampling algorithms are allowed to use the evaluation functions, DO$\alpha\beta$ is outperformed by UCT in both domains tested with evaluation function and also by RM in Tron. Using a good evaluation function instead of random simulations helps all sampling algorithms, but the amount of improvement is different for individual algorithms in different domains.

Third, the novel RM and OOS algorithms have proven efficient in a wider range of domains. Besides Goofspiel used for evaluating earlier versions of the algorithms in [11], RM showed strong performance in random games and both RM and OOS were the best performing algorithms in Tron with the evaluation function. These algorithms did not exploit the weaker opponents the most but often won against all other competitors. A notable advantage of these algorithms is a lower sensitivity for the parameter tuning, since they perform well in a wide range of domains with similar exploration parameters.

Fourth, when the algorithms have five times more time for finding a move to play, the differences between win rates of the sampling algorithms get smaller. Longer thinking time also has the same effect on parameter tuning and it also significantly improves the performance of the sampling algorithms against backward induction. This is expected, since the difference is too small for the DO$\alpha\beta$ algorithm to reach a greater depth, while it is sufficient for the sampling algorithms to execute five times more iterations improving their strategy.

Finally, the performance of Exp3 is the weakest in general. Its main problems are its larger computational complexity and problematic normalization for wider ranges of payoffs. Exp3 was significantly worse than other algorithms in both domains where we evaluated the point difference optimization and it performs an order of magnitude fewer iterations in Oshi-Zumo, compared to all other sampling algorithms.

6.6. Online Outcome Sampling versus Regret Matching

Given the similar nature of OOS and RM, one might wonder why RM typically performs better than OOS in online search, despite OOS being the only algorithm with provable convergence properties and the fastest converging algorithm in the offline setting. In this subsection, we investigate this phenomenon and present the results of additional experiments.

We need to look at the convergence properties of OOS, which is essentially an application of outcome sampling MCCFR. From the convergence bound of outcome sampling MCCFR presented in [86], after T iterations the strategy produced by the algorithm is an ϵ-Nash equilibrium with probability $1-p$ and

$$\epsilon \leq O\left(\frac{\tilde{\Delta}_i|\mathcal{S}|}{\sqrt{T}} + \sqrt{\frac{\mathbf{Var}}{pT} + \frac{\mathbf{Cov}}{p}}\right),$$

where $\tilde{\Delta}_i$ is determined by the structure of the game, and **Var** and **Cov** are the maximal variance and covariance of the differences between the exact value of a regret of an action and its estimate computed based on the selected sample $(r^t(s,a) - \tilde{r}^t(s,a))$ over all states, actions, and time steps. Computing these quantities exactly is prohibitively expensive, and since the scale of the exact regrets is bounded by a relatively small range of utilities, we can estimate the variance of the difference by the variance of the sampled regrets, which has often a very large range due to the importance sampling correction (see Section 4.5). We measure the estimate $\widehat{\mathbf{Var}} = \mathrm{Var}[\max_{a\in\mathcal{A}(s)} \tilde{r}^t(s,a)]$ in the root of the games, since they have the largest range of possible values of $\tilde{r}^t(s,a)$. Regret matching also computes a quantity similar to $\tilde{r}^t(s,a)$. The only difference is that they are not counterfactual, i.e., they take into account only the value of the current sample and not the expected value of the strategy used throughout the entire game. We show these variances for Goofspiel(13), OZ(50,3,1), and Tron(13, 13) in Table 7.

The results show that the variance of OOS is significantly higher than in case of RM. As such, even though RM may be introducing some kind of bias by bootstrapping value estimates from its own subgame, when there are so few samples this trade-off may be worthwhile to avoid the uncertainty

Game	Run	T_{OOS}	$\widehat{\mathbf{Var}}_{\text{OOS}}$	T_{RM}	$\widehat{\mathbf{Var}}_{\text{RM}}$
Goofspiel(13)	1	12,582	32,939.94	11,939	283.03
Goofspiel(13)	2	13,888	26,737.95	7,160	359.96
Goofspiel(13)	3	13,906	27,283.47	7,897	552.24
OZ(50,3,1)	1	34,900	1,010.73	25,654	9.19
OZ(50,3,1)	2	40,876	1,225.89	26,719	7.93
OZ(50,3,1)	3	40,306	1,016.42	26,121	7.99
Tron(13,13)	1	11,222	40.23	11,634	0.84
Tron(13,13)	2	12,526	35.91	11,134	0.83
Tron(13,13)	3	13,000	22.23	10,075	0.75

Table 7: Measure of variances of estimated regret quantities in OOS and Regret Matching at the root of each game. T is the number of iterations each algorithm runs for, and Run marks the run number (instance).

introduced by the variance. This problem is not apparent in the smaller games, because the higher probability of sampling individual terminal histories causes smaller variance and OOS performs enough samples to make the regret estimates sufficiently close to the true values. For example, in Goofspiel(5) used for offline convergence experiments, OOS performs approximately 2×10^5 iterations per second and the variance is only around 350.

7. Conclusion and Future Research

In this paper, we provide an extensive analysis of algorithms for solving and playing zero-sum extensive-form games with perfect information and simultaneous moves. We describe a collection of exact algorithms based on backward induction as well as a collection of Monte Carlo tree search algorithms including our novel algorithms DO$\alpha\beta$, BI$\alpha\beta$, SM-OOS and SM-MCTS with regret matching selection function.

We empirically compare the performance of these algorithms on six substantially different games in two different settings. In the offline equilibrium computation setting, we show that our novel algorithm based on backward induction, DO$\alpha\beta$, is able to prune large parts of the search space. In most games, DO$\alpha\beta$ is several orders of magnitude faster than the classical backward induction and it is never significantly outperformed by any of its competitors. The only benefit of the sampling algorithms in the offline setting is a to get a rough approximation of the equilibrium solution in a short time. Their results are often inconsistent with short computation times. Given enough time, the results clearly show that SM-OOS achieves the fastest convergence to a Nash equilibrium. Finally, our offline experiments also explained different behavior reported in variants of SM-MCTS with UCT selection. We have shown that adding randomization to tie-breaking rules can significantly improve the performance of this algorithm.

The success in the offline equilibrium computation is, however, not a very good indicator of the game playing performance in the online setting of head-to-head matches. First of all, the sizes of the games used for online experiments are too large for exact algorithms to be applicable without a domain-specific evaluation function. Performance of the representative of the exact algorithms, DO$\alpha\beta$, depends heavily on the accuracy of the used evaluation function. Secondly, in spite of the fastest convergence of SM-OOS among the sampling algorithms, SM-OOS does not always perform well in the online game playing. This is mainly due to the large variance of the regret updates that increases significantly in these large games. Among the remaining sampling algorithms, SM-MCTS based on regret matching is often very good, but sometimes it was outperformed by SM-MCTS with UCT selection, especially in games that require less randomized strategies.

Our work opens several interesting directions for future research. After introducing a strong pruning algorithm, it is of interest to formally study the limitations of pruning for this class of games, similarly to the theory developed for games with sequential moves. Future work could show if these pruning techniques can be substantially improved or if they are in some sense optimal. The main prerequisite is, however, estimating the expected number of iterations of the double-oracle algorithms for single step matrix games, which still remains an open problem. Furthermore, running large head-to-head tournaments for evaluating the game playing performance is time consuming, sensitive to setting correct parameters, and provides only limited insights into the performance of the algorithms. Proximity to the Nash equilibrium is not always a good indicator of game playing performance; hence, it is interesting to study alternative measures of quality of the algorithms that would better predict their game-playing performance in large games.

Acknowledgements. This work is funded by the Czech Science Foundation (grant no. P202/12/2054 and 15-23235S) and the Netherlands Organisation for Scientific Research (NWO) in the framework of the project Go4Nature, grant number 612.000.938. Branislav Bošanský also acknowledges support from the Danish National Research Foundation and The National Science

Foundation of China (under the grant 61361136003) for the Sino-Danish Center for the Theory of Interactive Computation, and the support from the Center for Research in Foundations of Electronic Markets (CFEM), supported by the Danish Strategic Research Council. The access to computing and storage facilities owned by parties and projects contributing to the National Grid Infrastructure MetaCentrum, provided under the programme "Projects of Large Infrastructure for Research, Development, and Innovations" (LM2010005) is highly appreciated.

References

[1] M. Campbell, A. J. Hoane, F. Hsu, Deep Blue, Artificial Intelligence 134 (1-2) (2002) 57–83.

[2] J. Schaeffer, R. Lake, P. Lu, M. Bryant, Chinook: The world man-machine checkers champion, AI Magazine 17 (1) (1996) 21–29.

[3] G. Tesauro, Temporal difference learning and TD-Gammon, Communications of the ACM 38 (3) (1995) 58–68.

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, D. Hassabis, Human-level control through deep reinforcement learning, Nature 518 (7540) (2015) 529–533.

[5] S. Russell, P. Norvig, Artificial Intelligence: A Modern Approach, 3rd Edition, Prentice Hall, 2009.

[6] A. Keuter, L. Nett, Ermes-auction in germany. first simultaneous multiple-round auction in the european telecommunications market, Telecommunications Policy 21 (4) (1997) 297 – 307.

[7] D. Beard, P. Hingston, M. Masek, Using Monte Carlo tree search for replanning in a multistage simultaneous game, in: Proceedings of the IEEE Congress on Evolutionary Computation (CEC), 2012.

[8] O. Teytaud, S. Flory, Upper confidence trees with short term partial information, in: Applications of Evolutionary Computation (EvoGames 2011), Part I, Vol. 6624 of LNCS, 2011, pp. 153–162.

[9] H. Gintis, Game Theory Evolving, 2nd Edition, Princeton University Press, 2009.

[10] B. Bošanský, V. Lisý, J. Čermák, R. Vítek, M. Pěchouček, Using Double-oracle Method and Serialized Alpha-Beta Search for Pruning in Simultaneous Moves Games, in: Proceedings of the 23rd International Joint Conference on Artificial Intelligence (IJCAI), 2013, pp. 48–54.

[11] M. Lanctot, V. Lisý, M. H. M. Winands, Monte Carlo tree search in simultaneous move games with applications to Goofspiel, in: Computer Games Workshop at IJCAI 2013, Vol. 408 of Communications in Computer and Information Science (CCIS), 2014, pp. 28–43.

[12] V. Lisý, V. Kovařík, M. Lanctot, B. Bošanský, Convergence of Monte Carlo tree search in simultaneous move games, in: Advances in Neural Information Processing Systems (NIPS), Vol. 26, 2013, pp. 2112–2120.

[13] V. Lisý, M. Lanctot, M. Bowling, Online Monte Carlo counterfactual regret minimization for search in imperfect information games, in: Proceedings of the 14th International Conference on Autonomous Agents and Multi-Agent Systems (AAMAS), 2015, pp. 27–36.

[14] M. Shafiei, N. Sturtevant, J. Schaeffer, Comparing UCT versus CFR in simultaneous games, in: Proceeding of the IJCAI Workshop on General Game-Playing (GIGA), 2009, pp. 75–82.

[15] A. Saffidine, Solving games and all that, Ph.D. thesis, Université Paris-Dauphine, Paris, France (2013).

[16] J. V. Neumann, Zur theorie der gesellschaftsspiele, Math Annalen 100 (1928) 295–320.

[17] M. L. Littman, Markov games as a framework for multi-agent reinforcement learning, in: In Proceedings of the 11th International Conference on Machine Learning (ICML), 1994, pp. 157–163.

[18] M. L. Littman, Value-function reinforcement learning in Markov games, Journal of Cognitive Systems Research 2 (1) (2001) 55–66.

[19] M. L. Littman, C. Szepesvári, A generalized reinforcement-learning model: Convergence and applications, in: Proceedings of the 13th International Conference on Machine Learning (ICML), 1996, pp. 310–318.

[20] M. G. Lagoudakis, R. Parr, Value function approximation in zero-sum markov games, in: Proceedings of the 18th Conference on Uncertainty in Artificial Intelligence (UAI), 2002, pp. 283–292.

[21] U. Savagaonkar, R. Givan, E. K. P. Chong, Sampling techniques for zero-sum, discounted Markov games, in: Proceedings of the 40th Annual Allerton Conference on Communication, Control and Computing, 2002, pp. 285–294.

[22] J. Perolat, B. Scherrer, B. Piot, O. Pietquin, Approximate dynamic programming for two-player zero-sum Markov games, in: Proceedings of the 32nd International Conference on Machine Learning (ICML), 2015.

[23] S. Singh, M. Kearns, Y. Mansour, Nash convergence of gradient dynamics in general-sum games, in: Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence (UAI), 2000, pp. 541–548.

[24] M. Bowling, M. Veloso, Convergence of gradient dynamics with a variable learning rate, in: Proceedings of the 18th International Conference on Machine Learning (ICML), 2001, pp. 27–34.

[25] M. Zinkevich, Online convex programming and generalized infinitesimal gradient ascent, in: Proceedings of 20th International Conference on Machine Learning (ICML), 2003, pp. 928–936.

[26] M. Bowling, Convergence and no-regret in multiagent learning, in: Advances in Neural Information Processing Systems 17 (NIPS), 2005, pp. 209–216.

[27] G. Gordon, No-regret algorithms for online convex programs, in: Proceedings of the 20th Annual Conference on Neural Information Processing Systems (NIPS), 2006, pp. 489–496.

[28] M. Zinkevich, M. Johanson, M. Bowling, C. Piccione, Regret minimization in games with incomplete information, in: Advances in Neural Information Processing Systems (NIPS), 2008, pp. 1729–1736.

[29] M. Bowling, N. Burch, M. Johanson, O. Tammelin, Heads-up limit hold'em poker is solved, Science 347 (6218) (2015) 145–149.

[30] A. Nowé, P. Vrancx, Y.-M. D. Hauwere, Game theory and multi-agent reinforcement learning, in: Reinforcement Learning: State-of-the-Art, 2012, Ch. 12, pp. 441–470.

[31] L. Buşoniu, R. Babuška, B. D. Schutter, A comprehensive survey of multi-agent reinforcement learning, IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews 38 (2) (2008) 156–172.

[32] D. Bloembergen, K. Tuyls, D. Hennes, M. Kaisers, Evolutionary dynamics of multi-agent learning: A survey", Journal of Artificial Intelligence Research (JAIR) 53 (2015) 659–697.

[33] Y. Shoham, K. Leyton-Brown, Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations, Cambridge University Press, 2009.

[34] R. Bellman, Dynamic Programming, Princeton University Press, 1957.

[35] S. M. Ross, Goofspiel — the game of pure strategy, Journal of Applied Probability 8 (3) (1971) 621–625.

[36] M. Buro, Solving the Oshi-Zumo game, in: Advances in Computer Games: Many Games, Many Challenges, Vol. 135 of IFIP—The International Federation for Information Processing, 2003, pp. 361–366.

[37] G. C. Rhoads, L. Bartholdi, Computer solution to the game of pure strategy, Games 3 (4) (2012) 150–156.

[38] A. Saffidine, H. Finnsson, M. Buro, Alpha-beta pruning for games with simultaneous moves, in: Proceedings of the 32nd Conference on Artificial Intelligence (AAAI), 2012, pp. 556–562.

[39] H. McMahan, G. Gordon, A. Blum, Planning in the presence of cost functions controlled by an adversary, in: Proceedings of the 20th International Conference on Machine Learning (ICML), 2003, pp. 536–543.

[40] M. Zinkevich, M. Bowling, N. Burch, A New Algorithm for Generating Equilibria in Massive Zero-Sum Games, in: Proceedings of the 27th Conference on Artificial Intelligence (AAAI), 2007, pp. 788–793.

[41] T. D. Hansen, P. B. Miltersen, T. B. Sørensen, On range of skill, in: Proceedings of the 28th Conference on Artificial Intelligence (AAAI), 2008, pp. 277–282.

[42] D. Koller, N. Megiddo, B. von Stengel, Efficient computation of equilibria for extensive two-person games, Games and Economic Behavior 14 (2) (1996) 247–259.

[43] T. Sandholm, The state of solving large incomplete-information games, and application to poker, AI Magazine 31 (4) (2010) 13–32.

[44] B. Bošanský, C. Kiekintveld, V. Lisý, M. Pěchouček, An Exact Double-Oracle Algorithm for Zero-Sum Extensive-Form Games with Imperfect Information, Journal of Artificial Intelligence Research 51 (2014) 829–866.

[45] R. Coulom, Efficient selectivity and backup operators in Monte-Carlo tree search, in: Proceedings of the 5th International Conference on Computers and Games (CG), Vol. 4630 of LNCS, 2006, pp. 72–83.

[46] L. Kocsis, C. Szepesvári, Bandit-based Monte Carlo planning, in: 15th European Conference on Machine Learning, Vol. 4212 of LNCS, 2006, pp. 282–293.

[47] S. Gelly, D. Silver, Monte-Carlo tree search and rapid action value estimation in computer Go, Artificial Intelligence 175 (11) (2011) 1856–1875.

[48] S. Gelly, L. Kocsis, M. Schoenauer, M. Sebag, D. Silver, C. Szepesvári, O. Teytaud, The grand challenge of computer Go: Monte Carlo tree search and extensions, Communications of the ACM 55 (3) (2012) 106–113.

[49] P. Ciancarini, G. Favini, Monte Carlo tree search in Kriegspiel, Artificial Intelligence 174 (11) (2010) 670–684.

[50] P. I. Cowling, E. J. Powley, D. Whitehouse, Information set Monte Carlo tree search, IEEE Transactions on Computational Intelligence and AI in Games 4 (2) (2012) 120–143.

[51] P. Auer, N. Cesa-Bianchi, P. Fischer, Finite-time analysis of the multi-armed bandit problem, Machine Learning 47 (2-3) (2002) 235–256.

[52] M. Genesereth, N. Love, B. Pell, General game-playing: Overview of the AAAI competition, AI Magazine 26 (2005) 73–84.

[53] H. Finnsson, Cadia-player: A general game playing agent, Master's thesis, Reykjavík University (2007).

[54] H. Finnsson, Simulation-based general game playing, Ph.D. thesis, Reykjavík University (2012).

[55] S. Samothrakis, D. Robles, S. M. Lucas, A UCT agent for Tron: Initial investigations, in: Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games (CIG), 2010, pp. 365–371.

[56] P. Auer, N. Cesa-Bianchi, Y. Freund, R. E. Schapire, The nonstochastic multiarmed bandit problem, SIAM Journal of Computing 32 (1) (2003) 48–77.

[57] P. Perick, D. L. St-Pierre, F. Maes, D. Ernst, Comparison of different selection strategies in Monte-Carlo tree search for the game of Tron, in: Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG), 2012, pp. 242–249.

[58] M. Lanctot, K. Waugh, M. Bowling, M. Zinkevich, Sampling for Regret Minimization in Extensive Games, in: Advances in Neural Information Processing Systems (NIPS), 2009, pp. 1078–1086.

[59] V. Kovařík, V. Lisý, Analysis of Hannan consistent selection for Monte Carlo tree search in simultaneous move games, CoRR abs/1509.00149.

[60] M. Lanctot, C. Wittlinger, M. H. M. Winands, N. G. P. Den Teuling, Monte Carlo tree search for simultaneous move games: A case study in the game of Tron, in: Proceedings of the 25th Benelux Conference on Artificial Intelligence (BNAIC), 2013, pp. 104–111.

[61] M. J. W. Tak, M. H. M. Winands, M. Lanctot, Monte Carlo tree search variants for simultaneous move games., in: Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG), 2014, pp. 232–239.

[62] T. Pepels, M. H. M. Winands, M. Lanctot, Real-time Monte Carlo tree search for Ms Pac-Man, IEEE Transactions on Computational Intelligence and AI in Games 6 (3) (2014) 245–257.

[63] D. Perez, E. J. Powley, D. Whitehouse, P. Rohlfshagen, S. Samothrakis, P. I. Cowling, S. M. Lucas, Solving the physical traveling salesman problem: Tree search and macro actions, IEEE Transanctions on Computational Intelligence and AI in Games 6 (1) (2014) 31–45.

[64] R.-K. Balla, A. Fern, UCT for tactical assault planning in real-time strategy games, in: Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI), 2009, pp. 40–45.

[65] P. I. Cowling, M. Buro, M. Bida, A. Botea, B. Bouzy, M. V. Butz, P. Hingston, H. Muñoz-Avila, D. Nau, M. Sipper, Search in real-time video games, in: Artificial and Computational Intelligence in Games, Vol. 6 of Dagstuhl Follow-Ups, 2013, pp. 1–19.

[66] M. G. Bellemare, Y. Naddaf, J. Veness, M. Bowling, The arcade learning environment: An evaluation platform for general agents, Journal of Artificial Intelligence Research (JAIR) 47 (2013) 253–279.

[67] S. Ontañón, G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill, M. Preuss, A survey of real-time strategy game AI research and competition in StarCraft, IEEE Transactions on Computational Intelligence and AI in Games 5 (4) (2013) 293–311.

[68] A. Kovarsky, M. Buro, Heuristic search applied to abstract combat games, Advances in Artificial Intelligence (2005) 55–77.

[69] F. Sailer, M. Buro, M. Lanctot, Adversarial planning through strategy simulation, in: IEEE Symposium on Computational Intelligence and Games (CIG), 2007, pp. 37–45.

[70] V. Lisý, B. Bošanský, M. Jakob, M. Pěchouček, Adversarial search with procedural knowledge heuristic, in: Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS), 2009, pp. 899–906.

[71] D. Churchill, A. Saffidine, M. Buro, Fast heuristic search for RTS game combat scenarios, in: 8th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE), 2012, pp. 112–117.

[72] A. Reinefeld, An improvement to the scout tree-search algorithm, ICCA Journal 6 (4) (1983) 4–14.

[73] S. Hart, A. Mas-Colell, A simple adaptive procedure leading to correlated equilibrium, Econometrica 68 (5) (2000) 1127–1150.

[74] M. Lanctot, Monte Carlo sampling and regret minimization for equilibrium computation and decision-making in large extensive form games, Ph.D. thesis, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada (June 2013).

[75] S. Gelly, D. Silver, Combining online and offline learning in UCT, in: Proceedings of the 24th International Conference on Machine Learning, 2007, pp. 273–280.

[76] R. Lorentz, Amazons discover Monte-Carlo, in: Proceedings of the 6th International Conference on Computers and Games (CG), Vol. 5131 of LNCS, 2008, pp. 13–24.

[77] M. H. M. Winands, Y. Björnsson, J.-T. Saito, Monte Carlo tree search in Lines of Action, IEEE Transactions on Computational Intelligence and AI in Games 2 (4) (2010) 239–250.

[78] R. Lorentz, T. Horey, Programming Breakthrough, in: Proceedings of the 8th International Conference on Computers and Games (CG), Vol. 8427 of LNCS, 2013, pp. 49–59.

[79] M. Lanctot, M. H. M. Winands, T. Pepels, N. R. Sturtevant, Monte Carlo tree search with heuristic evaluations using implicit minimax backups, in: Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG), 2014, pp. 341–348.

[80] R. Ramanujan, B. Selman, Trade-offs in sampling-based adversarial planning, in: Proceedings of the 21st International Conference on Automated Planning and Scheduling (ICAPS), 2011, pp. 202–209.

[81] M. Lanctot, A. Saffidine, J. Veness, C. Archibald, M. H. M. Winands, Monte Carlo *-minimax search, in: Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence (IJCAI), 2013, pp. 580–586.

[82] K. Q. Nguyen, R. Thawonmas, Monte Carlo tree search for collaboration control of ghosts in Ms. Pac-Man, IEEE Transactions on Computational Intelligence and AI in Games 5 (1) (2013) 57–68.

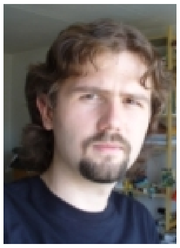
[83] S. Smith, D. Nau, An analysis of forward pruning, in: Proceedings of the National Conference on Artificial Intelligence, 1995, pp. 1386–1386.

[84] N. G. P. Den Teuling, M. H. M. Winands, Monte-Carlo Tree Search for the simultaneous move game Tron, in: Proceedings of Computer Games Workshop (ECAI), 2012, pp. 126–141.

[85] M. Ponsen, S. de Jong, M. Lanctot, Computing approximate Nash equilibria and robust best-responses using sampling, Journal of Artificial Intelligence Research (JAIR) 42 (2011) 575–605.

[86] R. Gibson, M. Lanctot, N. Burch, D. Szafron, M. Bowling, Generalized sampling and variance in counterfactual regret minimization, in: Proceedings of the 26th Conference on Artificial Intelligence (AAAI), 2012, pp. 1355–1361.

Vitae



Branislav Bošanský is an Assistant Professor at the Department of Computer Science, at the Faculty of Electrical Engineering at the Czech Technical University in Prague. He received his Ph.D. degree in Artificial Intelligence from the Department of Cybernetics at the same faculty and was a post-doctoral research fellow at Aarhus University in Denmark. His research focuses on algorithmic and computational game theory, computing strategies in sequential games, and applications to security.

Viliam Lisý reveived his Ph.D. degree in Artificial Intelligence from Czech Technical University in Prague. He holds a master's degree in Technical Artificial Intelligence from VU University Amsterdam and a master's degree in Theoretical Computer Science from the Faculty of Mathematics and Physics at Charles University in Prague. Currently, he is a postdoctoral fellow at Department of Computing Science, University of Alberta.

Marc Lanctot is a research scientist at Google DeepMind in London, United Kingdom. He received his Ph.D. degree in Artificial Intelligence from the Department of Computer Science, University of Alberta, Edmonton, Canada, in 2012. Previously, he was a post-doctoral research fellow at the Department of Data Science and Knowledge Engineering, Maastricht University. His work has focused on learning and search in sequential games.

Jiří Čermák is a PhD student at Agent Technology Center, Department of Computer Science, Faculty of Electrical Engineering at Czech Technical University in Prague. His research focuses on computational game theory and sequential games.

Mark H.M. Winands received the Ph.D. degree in Artificial Intelligence from the Department of Computer Science, Maastricht University, Maastricht, The Netherlands, in 2004. Currently, he is an Assistant Professor at the Department of Data Science and Knowledge Engineering, Maastricht University. His research interests include heuristic search, machine learning and games. Dr. Winands serves as a section editor of the ICGA Journal and as an associate editor of IEEE Transactions on Computational Intelligence and AI in Games.