Journal of Artificial Intelligence Research 51 (2014) 829-866 Submitted 06/14; published 12/14

An Exact Double-Oracle Algorithm for Zero-Sum Extensive-Form Games with Imperfect Information

Branislav Bošanský BRANISLAV.BOSANSKY@AGENTS.FEL.CVUT.CZ
Agent Technology Center
Department of Computer Science
Faculty of Electrical Engineering
Czech Technical University in Prague

Christopher Kiekintveld CDKIEKINTVELD@UTEP.EDU
Computer Science Department
University of Texas at El Paso, USA

Viliam Lisý VILIAM.LISY@AGENTS.FEL.CVUT.CZ
Michal Pěchouček MICHAL.PECHOUCEK@AGENTS.FEL.CVUT.CZ
Agent Technology Center
Department of Computer Science
Faculty of Electrical Engineering
Czech Technical University in Prague

Abstract

Developing scalable solution algorithms is one of the central problems in computational game theory. We present an iterative algorithm for computing an exact Nash equilibrium for two-player zero-sum extensive-form games with imperfect information. Our approach combines two key elements: (1) the compact sequence-form representation of extensive-form games and (2) the algorithmic framework of double-oracle methods. The main idea of our algorithm is to restrict the game by allowing the players to play only selected sequences of available actions. After solving the restricted game, new sequences are added by finding best responses to the current solution using fast algorithms.

We experimentally evaluate our algorithm on a set of games inspired by patrolling scenarios, board, and card games. The results show significant runtime improvements in games admitting an equilibrium with small support, and substantial improvement in memory use even on games with large support. The improvement in memory use is particularly important because it allows our algorithm to solve much larger game instances than existing linear programming methods.

Our main contributions include (1) a generic sequence-form double-oracle algorithm for solving zero-sum extensive-form games; (2) fast methods for maintaining a valid restricted game model when adding new sequences; (3) a search algorithm and pruning methods for computing best-response sequences; (4) theoretical guarantees about the convergence of the algorithm to a Nash equilibrium; (5) experimental analysis of our algorithm on several games, including an approximate version of the algorithm.

1. Introduction

Game theory is a widely used methodology for analyzing multi-agent systems by applying formal mathematical models and solution concepts. One focus of computational game theory is the development of scalable algorithms for reasoning about very large games. The need for continued algorithmic advances is driven by a growing number of applications of game theory that require solving very large game instances. For example, several decision support systems have recently been deployed in homeland security domains to recommend policies based on game-theoretic models for placing checkpoints at airports (Pita, Jain, Western, Portway, Tambe, Ordonez, Kraus, & Parachuri, 2008), scheduling Federal Air Marshals (Tsai, Rathi, Kiekintveld, Ordóñez, & Tambe, 2009), and patrolling ports (Shieh, An, Yang, Tambe, Baldwin, Direnzo, Meyer, Baldwin, Maule, & Meyer, 2012). The capabilities of these systems are based on a large amount of research in fast algorithms for security games (Tambe, 2011). Another notable example is the algorithmic progress that has led to game-theoretic Poker agents that are competitive with highly skilled human opponents (e.g., see Zinkevich, Bowling, & Burch, 2007; Sandholm, 2010).

We focus on developing new algorithms for an important general class of games that includes security games and Poker, as well as many other familiar games. More precisely, we study two-player zero-sum extensive-form games (EFGs) with imperfect information. This class of games captures sequential interactions between two strictly competitive players in situations where they make decisions under uncertainty. Uncertainty can be caused either by having a stochastic environment or by having opponent actions that are not directly observable. We consider general models for both sequential interactions and uncertainty, while many of the fast algorithms that have been developed for Poker and security domains rely on more specific game structure.

We propose a new class of algorithms for finding exact (or approximate) Nash equilibrium solutions for the class of EFGs with imperfect information. The leading exact algorithm in the literature uses the compact *sequence-form* representation and linear programming optimization techniques to solve games of this type (Koller, Megiddo, & von Stengel, 1996; von Stengel, 1996). Our approach exploits the same compact representation, but we improve the solution methods by adopting the algorithmic framework based on decompositions known in the computational game theory literature as *oracle algorithms* (McMahan, Gordon, & Blum, 2003). Oracle algorithms are related to the methods of constraint/column generation used for solving large-scale optimization problems (Dantzig & Wolfe, 1960; Barnhart, Johnson, Nemhauser, Savelsbergh, & Vance, 1998) and exploit two characteristics commonly found in games. First, in many cases finding a solution to a game only requires using a small fraction of the possible strategies, so it is not necessary to enumerate all of the strategies to find a solution (Wilson, 1972; Koller & Megiddo, 1996). Second, finding a best response to a specific opponent strategy in a game is computationally much less expensive than solving for an equilibrium. In addition, best response algorithms can often make use of domain-specific knowledge or heuristics to speed up the calculations even further.

Our sequence-form double-oracle algorithm integrates the decomposition ideas of oracle algorithms with the compact sequence-form representation for EFGs with imperfect information. This results in an iterative algorithm that does not always need to generate the complete linear program for the game to find a Nash equilibrium solution. The main idea of the algorithm is to create a restricted game in which the players choose from a limited space of possible strategies (represented as sequences of actions). The algorithm solves the restricted game and then uses a fast best-response algorithm to find strategies in the original unrestricted game that perform well against the current solution of the restricted game. These strategies are added to the restricted game and the process iterates until no best response can be found to improve the solution. In this case, the current solution is an equilibrium of the original game. Typically, a solution can be found by adding only a small fraction of the strategies to the restricted game.

We begin by presenting related work, technical background, and our notation. We then describe our main algorithm in three parts: (1) methods for creating, solving, and expanding a valid restricted game, (2) the algorithm for finding the best-response strategies to be added to the restricted game, and (3) variants of the main loop controlling the iterative process of solving restricted games and adding new strategies. We present a formal analysis and prove that our algorithm converges to a Nash equilibrium of the original game. Finally, we provide an experimental evaluation of the runtime performance and convergence behavior of our algorithm on several realistic games with different characteristics including a border patrolling scenario, Phantom Tic-Tac-Toe, and a simplified variant of Poker. We compare our results with state-of-the-art algorithms for finding both exact and approximate solutions: linear programming using the sequence form, and Counterfactual Regret Minimization (CFR, Zinkevich, Johanson, Bowling, & Piccione, 2008; Lanctot, 2013).

The experimental results confirm that our algorithm requires only a fraction of all possible sequences to solve a game in practice and significantly reduces memory requirements when solving large games. This advances the state of the art and allows us to exactly solve much larger games compared to the existing algorithms. Moreover, in games admitting an equilibrium with small support (i.e., only a few sequences have non-zero probability in an equilibrium), our algorithm also achieves significant improvements in computation time and finds an equilibrium after only few iterations. These result hold without using any domain-specific knowledge, but we also show that incorporating domain-specific heuristics and bounds into the algorithm in a straightforward way can lead to even more significant performance improvements. Analysis of the convergence rate shows that the approximative bounds on the value of the game are either similar or a bit worse during the early stages compared to CFR. However, the convergence behavior of CFR algorithm has a very long tail and our algorithm always finds an exact solution much faster than CFR.

2. Related Work

Solving imperfect-information EFGs is a computationally challenging task, primarily due to uncertainty about the actions of the opponent and/or a stochastic environment. The leading exact algorithm (Koller et al., 1996; von Stengel, 1996) is based on formulating the problem of finding an optimal strategy to play as a linear program. This algorithm exploits a compact representation of strategies as sequences of individual actions (called *the sequence form*) and results in a linear program of linear size in the size of the game tree. However, this approach has limited applicability since the game tree grows exponentially with the number of sequential actions in the game. A common practice for overcoming the limited scalability of sequence-form linear programming is to use an approximation method. The best known approximative algorithms include counterfactual regret minimization (CFR, Zinkevich et al., 2008), improved versions of CFR with sampling methods (Lanctot, Waugh, Zinkevich, & Bowling, 2009; Gibson, Lanctot, Burch, Szafron, & Bowling, 2012); Nesterov's Excessive Gap Technique (EGT, Hoda, Gilpin, Peña, & Sandholm, 2010); and variants of

Monte Carlo Tree Search (MCTS) algorithms applied to imperfect-information games (e.g., see Ponsen, de Jong, & Lanctot, 2011).

The family of counterfactual regret minimization algorithms is based on learning methods that can be informally described as follows. The algorithm repeatedly traverses the game tree and learns a strategy to play by applying a no-regret learning rule that minimizes a specific variant of regret (counterfactual regret) in each information set. The no-regret learning converges to an optimal strategy in each information set. The overall regret is bounded by the sum of the regret in each information set; hence, the strategy as a whole converges to a Nash equilibrium. The main benefits of this approach include simplicity and robustness, as it can be adapted for more generic games (e.g., see Lanctot, Gibson, Burch, Zinkevich, & Bowling, 2012, where CFR is applied on games with imperfect recall). However, the algorithm operates on the complete game tree and therefore requires convergence in all information sets, which can be very slow for large games when one desires a solution with small error.

Another popular method is Excessive Gap Technique that exploits the convex properties of the sequence-form representation and uses recent mathematical results on finding extreme points of smooth functions (see Hoda et al., 2010, for the details). The main idea is to approximate the problem of finding a pair of equilibrium strategies by two smoothed functions and guiding them to find an approximate solution. Although this approach achieves faster convergence in comparison with CFR, the algorithm is less robust (it is not known whether a similar approach can be used for more general classes of games) and less used in practice. Like CFR, EGT also operates in the complete strategy space of all sequences.

Monte Carlo Tree Search (MCTS) is another family of methods that has shown promise for solving very large games, in particular perfect information board games such as Go (e.g., Lee et al., 2009). While the CFR and EGT algorithms are guaranteed to find an ε-Nash equilibrium, convergence to an equilibrium solution has not been formally shown for any of the variants of MCTS in imperfect-information games. On the contrary, the most common version of MCTS based on the Upper Confidence Bounds (UCB) selection function can converge to incorrect solutions even in simultaneous-move games (Shafiei, Sturtevant, & Schaeffer, 2009) that are the simplest class of imperfect-information EFGs. MCTS algorithms therefore do not (in general) guarantee finding an (approximate) optimal solution in imperfect-information games. One exception is the recent proof of convergence of MCTS with certain selection methods for simultaneous-move games (Lisy, Kovarik, Lanctot, & Bosansky, 2013). Still, using MCTS is sometimes a reasonable choice since it can produce good strategies in practice (Ponsen et al., 2011).

Contrary to the existing approximative approaches, our algorithm aims to find an exact solution without explicitly considering the strategy in the complete game tree. Our work combines the compact sequence-form representation and the double-oracle algorithmic framework. Previous work on the double-oracle framework has focused primarily on applications in normal-form games, where the restricted game was expanded by adding pure best-response strategies in each iteration. One of the first examples of solving games using the double-oracle principle was by McMahan et al. (2003). They introduced the double-oracle algorithm, proved the convergence to a Nash equilibrium, and experimentally verified that the algorithm achieves computation time improvements on a search game where an evader was trying to cross an environment without being detected by sensors placed by the opponent. The double-oracle algorithm reduced the computation time from several hours to tens of seconds and allowed to solve much larger instances of this game. Similar success with the domain-specific double-oracle methods has been demonstrated on a variety of different domains inspired by pursuit-evasion games (Halvorson, Conitzer, & Parr, 2009) and security games played on a graph (Jain, Korzhyk, Vanek, Conitzer, Tambe, & Pechoucek, 2011; Letchford & Vorobeychik, 2013; Jain, Conitzer, & Tambe, 2013).

Only a few works have tried to apply the iterative framework of oracle algorithms to EFGs, primarily using pure and mixed strategies in EFGs. The first work that exploited this iterative principle is the predecessor of the sequence-form linear-program formulation (Koller & Megiddo, 1992). In this algorithm, the authors use a representation similar to the sequence form only for a single player, while the strategies for the opponent are iteratively added as constraints into the linear program (there is an exponential number of constraints in their formulation). This approach can be seen as a specific variant of the oracle algorithms, where the strategy space is expanded gradually for a single player. Our algorithm is a generalization of this work, since our algorithm uses the sequence-form representation for both players and it also incrementally expands the strategy space for both players.

More recent work has been done by McMahan in his thesis (McMahan, 2006) and follow-up work (McMahan & Gordon, 2007). In these works the authors investigated an extension of the double-oracle algorithm for normal-form games to the extensive-form case. Their double-oracle algorithm for EFGs operates very similarly to the normal-form variant and uses pure and mixed strategies defined for EFGs. The main disadvantage of this approach is that in the basic version it still requires a large amount of memory since a pure strategy for an EFG is large (one action needs to be specified for each information set), and there is an exponential number of possible pure strategies. To overcome this disadvantage, the authors propose a modification of the double-oracle algorithm that keeps the number of the strategies in the restricted game bounded. The algorithm removes from the restricted game those strategies that are the least used in the current solution of the restricted game. In order to guarantee the convergence, the algorithm adds in each iteration into the restricted game a mixed strategy representing the mean of all removed strategies; convergence is then guaranteed similarly to fictitious play (see McMahan & Gordon, 2007, for the details). Bounding the size of the restricted game results in low memory requirements. However, the algorithm converges extremely slowly and it can take a very long time (several hours for a small game) for the algorithm to achieve a small error (see the experimental evaluation in McMahan, 2006; McMahan & Gordon, 2007).

A similar concept for using pure strategies in EFGs is used in an iterative algorithm designed for Poker in the work of Zinkevich et al. (2007). The algorithm in this work expands the restricted game with strategies found by *a generalized best response* instead of using pure best response strategies. Generalized best response is a Nash equilibrium in a partially restricted game – the player computing the best response can use any of the pure strategies in the original unrestricted game, while the opponent is restricted to use only the strategies from the restricted game. However, the main disadvantages of using pure and mixed strategies in EFGs are still present and result in large memory requirements and an exponential number of iterations.

In contrast, our algorithm directly uses the compact sequence-form representation of EFGs and uses the sequences as the building blocks (i.e., the restricted game is expanded by allowing new sequences to be played in the next iteration). Using sequences and the sequence form for solving the restricted game reduces the size of the restricted game and the number of iterations, however, it also introduces new challenges when constructing and maintaining the restricted game, and ensuring the convergence to a Nash equilibrium, which we must solve for our algorithm to converge to a correct solution.

3. Technical Background

We begin by presenting the standard game-theoretic model of extensive-form games, followed by a discussion of the most common solution concepts and the algorithms for computing these solutions. Then we present the sequence-form representation and the state-of-the-art linear program for computing solutions using this representation. Finally, we describe oracle algorithms as they are used for solving normal-form games. A summary of the most common notation is provided in Table 1 for quick reference.

3.1 Extensive-Form Games

Extensive-form games (EFGs) model sequential interactions between players in a game. Games in the extensive form are visually represented as game trees (e.g., see Figure 2). Nodes in the game tree represent states of the game; each state of the game corresponds to a sequence of moves executed by all players in the game. Each node is assigned to a player that acts in the game state associated with this node. An edge in the game tree from a node corresponds to an action that can be performed by the player who acts in this node. Extensive-form games model limited observations of the players by grouping the nodes into *information sets*, so that a given player cannot distinguish between nodes that belong to the same information set when the player is choosing an action. The model also represents uncertainty about the environment and stochastic events by using a special *Nature player*.

Formally, a two-player EFG is defined as a tuple $G = (N, H, Z, A, p, u, \mathcal{C}, \mathcal{I})$: N is a set of two players $N = \{1, 2\}$. We use i to refer to one of the two players (either 1 or 2), and $-i$ to refer to the opponent of i. H denotes a finite set of *nodes* in the game tree. Each node corresponds to a unique *history* of actions taken by all players and Nature from the root of the game; hence, we use the terms history and node interchangeably. We denote by $Z \subseteq H$ the set of all *terminal nodes* of the game. A denotes the set of all actions and we overload the notation and use $A(h) \subseteq A$ to represent the set of actions available to the player acting in node $h \in H$. We specify $ha = h' \in H$ to be node h' reached from node h by executing action $a \in A(h)$. We say that h is a *prefix* of h' and denote it by $h \sqsubseteq h'$. For each terminal node $z \in Z$ we define a *utility function* for each player i ($u_i : Z \to \mathbb{R}$). We study zero-sum games, so $u_i(z) = -u_{-i}(z)$ holds for all $z \in Z$.

The function $p : H \to N \cup \{c\}$ assigns each node to a player who takes an action in the node, where c means that the Nature player selects an action in the node based on a fixed probability distribution known to all players. We use function $\mathcal{C} : H \to [0, 1]$ to denote the probability of reaching node h due to Nature (i.e., assuming that both players play all required actions to reach node h). The value of $\mathcal{C}(h)$ is the product of the probabilities assigned to all actions taken by the Nature player in history h. Imperfect observation of player i is modeled via *information sets* $\mathcal{I}_i$ that form a partition over the nodes assigned to player i $\{h \in H : p(h) = i\}$. Every information set contains at least one node and each node belongs to exactly one information set. Nodes in an information set of a player are indistinguishable to the player. All nodes h in a single information set $I_i \in \mathcal{I}_i$ have the same set of possible actions $A(h)$. Action a from $A(h)$ uniquely identifies information set I_i and there cannot exist any other node $h' \in H$ that does not belong to information set I_i and for which a is allowed to be played (i.e., $a \in A(h')$). Therefore we overload notation and use $A(I_i)$ to denote the set of actions defined for each node h in this information set. We assume *perfect recall*, which means that players perfectly remember their own actions and all information gained during the course of the game. As a result, all nodes in any information set I_i have the same history of actions for player i.

3.2 Nash Equilibrium in Extensive-Form Games

Solving a game requires finding a strategy profile (i.e., one strategy for each player) that satisfies conditions defined by a specific solution concept. *Nash equilibrium* (NE) is the best known solution concept in game theory and it describes the behavior of players under certain assumptions about their rationality. In a Nash equilibrium, every player plays a best response to the strategies of the other players. Let Π_i be the set of *pure strategies* for player i. In EFGs, a pure strategy is an assignment of exactly one action to be played in each information set. A mixed strategy is a probability distribution over the set of all pure strategies of a player. We denote by Δ_i the set of all mixed strategies of player i. For any pair of strategies $\delta \in \Delta = (\Delta_1, \Delta_2)$ we use $u_i(\delta) = u_i(\delta_i, \delta_{-i})$ for the expected outcome of the game for player i when players follow strategies δ. A *best response* of player i to the opponent's strategy δ_{-i} is a strategy δ_i^{BR}, for which $u_i(\delta_i^{BR}, \delta_{-i}) \geq u_i(\delta_i', \delta_{-i})$ for all strategies $\delta_i' \in \Delta_i$. A strategy profile $\delta = (\delta_1, \delta_2)$ is a NE if and only if for each player i it holds that δ_i is a best response to δ_{-i}. A game can have multiple NEs; in the zero-sum setting, all of these equilibria have the same value (i.e., the expected utility for every player is the same). This is called the *value of the game*, denoted V^*. The problem of finding a NE in a zero-sum game has a polynomial computational complexity in the size of the game.

The NE solution concept is somewhat weak for extensive-form games. Nash equilibrium requires that both players act rationally. However, there can be irrational strategies selected for the parts of the game tree that are not reachable when both players follow the NE strategies (these parts are said to be *off the equilibrium path*). The reason is that NE does not expect this part of the game to be played and therefore does not sufficiently restrict strategies in these information sets. To overcome these drawbacks, a number of refinements of NE have been introduced imposing further restrictions with the intention of describing more sensible strategies. Examples include *subgame-perfect equilibrium* (Selten, 1965) used in perfect-information EFGs. The subgame-perfect equilibrium forces the strategy profile to be a Nash equilibrium in each sub-game (i.e., in each sub-tree rooted in some node h) of the original game. Unfortunately, sub-games are not particularly useful in imperfect-information EFGs; hence, here the refinements include *strategic-from perfect equilibrium* (Selten, 1975), *sequential equilibrium* (Kreps & Wilson, 1982), or *quasi-perfect equilibrium* (van Damme, 1984; Miltersen & Sørensen, 2010). The first refinement avoids using weakly dominated strategies in equilibrium strategies for two-player games (van Damme, 1991, p. 29) and it is also known as the *undominated equilibrium*. Sequential equilibrium tries to exploit the mistakes of the opponent by using the notion of beliefs consistent with the strategy of the opponent even in information sets off the equilibrium path. The main intuitions behind the first two refinements are combined in quasi-perfect equilibrium.

Even though the solution described by NE does not always prescribe rational strategies off the equilibrium path, it is still valuable to compute exact NE of large extensive-form games for several reasons. We focus on zero-sum games, so the NE strategy guarantees the value of the game even off the equilibrium path. In other words, the strategy off the equilibrium path does not optimally exploit the mistakes of the opponent, but it still guarantees an outcome of at least value gained by following the equilibrium path. Moreover, a refined equilibrium is still a NE and calculating the value of the game is often a starting point for many of the algorithms that compute these refinements – for example it is used for computing undominated equilibrium (e.g., see Ganzfried & Sandholm, 2013; Cermak, Bosansky, & Lisy, 2014) and *normal-form proper equilibrium* (Miltersen & Sørensen, 2008).

3.3 Sequence-Form Linear Program

Extensive-form games with perfect recall can be compactly represented using the *sequence form* (Koller et al., 1996; von Stengel, 1996). A *sequence* σ_i is an ordered list of actions taken by a single player i in a history h. The number of actions (i.e., the length of sequence σ_i) is denoted by $|\sigma_i|$ and the empty sequence (i.e., sequence with no actions) is denoted by $\emptyset$. The set of all possible sequences for player i is denoted by Σ_i and the set of sequences for all players is $\Sigma = \Sigma_1 \times \Sigma_2$. A sequence $\sigma_i \in \Sigma_i$ can be extended by a single action a taken by player i, denoted by $\sigma_i a = \sigma_i'$ (we use $\sigma_i \sqsubseteq \sigma_i'$ to denote that σ_i is a prefix of σ_i'). In games with perfect recall, all nodes in an information set I_i share the same sequence of actions for player i and we use $\mathsf{seq}_i(I_i)$ to denote this sequence. We overload the notation and use $\mathsf{seq}_i(h)$ to denote the sequence of actions of player i leading to node h, and $\mathsf{seq}_i(H') \subseteq \Sigma_i$, where $\mathsf{seq}_i(H') = \bigcup_{h' \in H'} \mathsf{seq}_i(h')$ for some $H' \subseteq H$. Since action a uniquely identifies information set I_i and all nodes in an information set share the same history of actions of player i, each sequence uniquely identifies an information set. We use the function $\mathsf{inf}_i(\sigma_i')$ to denote the information set in which the last action of the sequence σ_i' is taken. For an empty sequence, function $\mathsf{inf}_i(\emptyset)$ is the information set of the root node.

Finally, we define the auxiliary payoff function $g_i : \Sigma \to \mathbb{R}$ that extends the utility function to all nodes in the game tree. The payoff function g_i represents the expected utility of all nodes reachable by sequentially executing the actions specified in a pair of sequences σ:

$$g_i(\sigma_i, \sigma_{-i}) = \sum_{h \in Z \,:\, \forall j \in N \; \sigma_j = \mathsf{seq}_j(h)} u_i(h) \cdot \mathcal{C}(h) \tag{1}$$

The value of the payoff function is defined to be 0 if no leaf is reachable by sequentially executing all of the actions in the sequences σ – either all actions from the pair of sequences σ are executed and an inner node ($h \in H \setminus Z$) is reached, or during the sequential execution of the actions node h is reached, for which the current action a to be executed from sequence $\sigma_{\rho(h)}$ is not defined (i.e., $a \notin A(h)$). Formally we define a pair of sequences σ to be *compatible* if there exists node $h \in H$ such that sequence σ_i of every player i equals to $\mathsf{seq}_i(h)$.

We can compute a Nash equilibrium of a two-player zero-sum extensive-form game using a linear program (LP) of a polynomial size in the size of the game tree using the sequence form (Koller et al., 1996; von Stengel, 1996). The LP uses an equivalent compact representation of mixed strategies of players in a form of *realization plans.* A realization plan for a sequence σ_i is the probability that player i will play this sequence of actions under the assumption that the opponent will choose compatible sequences of actions that reach the information sets for which the actions specified in the sequence σ_i are defined. We denote the realization plan for player i by $r_i : \Sigma_i \to \mathbb{R}$. The equilibrium realization plans can be computed using the following LP (e.g., see Shoham & Leyton-Brown, 2009, p. 135):

$$\max_{r,v} v_{\mathsf{inf}_{-i}(\emptyset)}$$

$$v_{\mathsf{inf}_{-i}(\sigma_{-i})} - \sum_{I'_{-i} \in \mathcal{I}_{-i} : \mathsf{seq}_{-i}(I'_{-i}) = \sigma_{-i}} v_{I'_{-i}} \leq \sum_{\sigma_i \in \Sigma_i} g_i(\sigma_{-i}, \sigma_i) \cdot r_i(\sigma_i) \qquad \forall \sigma_{-i} \in \Sigma_{-i} \tag{2}$$

$$r_i(\emptyset) = 1 \tag{3}$$

$$\sum_{\forall a \in A(I_i)} r_i(\sigma_i a) = r_i(\sigma_i) \qquad \forall I_i \in \mathcal{I}_i,\ \sigma_i = \mathsf{seq}_i(I_i) \tag{4}$$

$$r_i(\sigma_i) \geq 0 \qquad \forall \sigma_i \in \Sigma_i \tag{5}$$

Solving the LP yields a realization plan for player i using variables r_i, and expected values for the information sets of player $-i$ (variables $v_{I_{-i}}$). The LP works as follows: player i maximizes the expected utility value by selecting the values for the variables of realization plan that is constrained by Equations (3–5). The probability of playing the empty sequence is defined to be 1 (Equation 3), and the probability of playing a sequence σ_i is equal to the sum of the probabilities of playing sequences extended by exactly one action (Equation 4). Finding such a realization plan is also constrained by the best responding opponent, player $-i$. This is ensured by Equation (2), where player $-i$ selects in each information set I_{-i} such action that minimizes the expected utility value $v_{I_{-i}}$ in this information set. There is one constraint defined for each sequence σ_{-i}, where the last action of this sequence determines the best action to be played in information set $\mathsf{inf}_{-i}(\sigma_{-i}) = I_{-i}$. The expected utility is composed of the expected utilities of the information sets reachable after playing sequence σ_{-i} (sum of v variables on the left side) and of the expected utilities of leafs to which this sequence leads (sum of g values on the right side of the constraint).

3.4 Double-Oracle Algorithm for Normal-Form Games

We now describe the concept of column/constraint generation techniques applied previously in normal-form games and known as the double-oracle algorithm (McMahan et al., 2003). *Normal-form games* are represented using game matrices; rows of the matrix correspond to pure strategies of one player, columns correspond to pure strategies of the opponent, and values in the matrix cells represent the expected outcome of the game when players play corresponding pure strategies. Zero-sum normal-form games can be solved by linear programming in polynomial time in the size of the matrix (e.g., see Shoham & Leyton-Brown, 2009, p. 89).

Figure 1 shows the visualization of the main structure of the double-oracle algorithm for normal-form games. The algorithm consists of the following three steps that repeat until convergence:

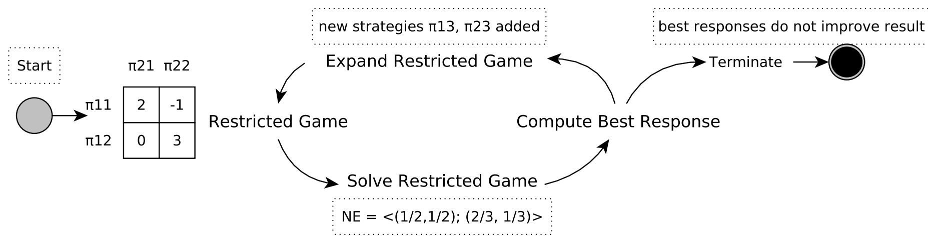


Figure 1: Schematic of the double-oracle algorithm for a normal-form game.

1. create a restricted game by limiting the set of pure strategies that each player is allowed to play
2. compute a pair of Nash equilibrium strategies in this restricted game using the LP for solving normal-form games
3. for each player, compute a pure best response strategy against the equilibrium strategy of the opponent found in the previous step; the best response may be *any* pure strategy in 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 continues 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. In this case both players play a best response to the strategy of the opponent in the original unrestricted game. The algorithm maintains the values of the expected utilities of the best-response strategies throughout the iterations of the algorithm. These values provide bounds on the value of the original unrestricted game V^* – from the perspective of player i, the minimal value of all of her past best-response calculations represents an upper bound of the value of the original game, V_i^{UB}, and the maximal value of all of past best-response calculations of the opponent represents the lower bound on the value of the original game, V_i^{LB}. Note that for the bounds it holds that the lower bound for player i is equal to the negative of the value of the upper bound for the opponent:

$$V_i^{LB} = -V_{-i}^{UB}$$

In general, computing best responses is computationally less demanding than solving the game, since the problem is reduced to a single-player optimization. Due to the fact that best-response algorithms can operate very quickly (e.g., also by exploiting additional domain-specific knowledge), they are called *oracles* in this context. If the algorithm incrementally adds strategies only for one player, the algorithm is called a *single-oracle algorithm*, if the algorithm incrementally adds the strategies for both players, the algorithm is called a *double-oracle algorithm*. Double-oracle algorithms are typically initialized by an arbitrary pair of strategies (one pure strategy for each player). However, we can also use a larger set of initial strategies selected based on a domain-specific knowledge.

The double-oracle algorithm for zero-sum normal-form games runs in a polynomial time in the size of the game matrix. Since each iteration adds at least one pure strategy to

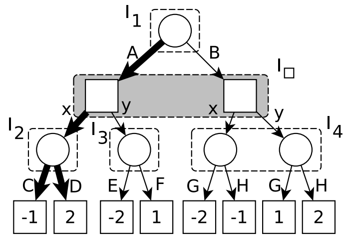


Figure 2: Example of a two-player extensive-form game visualized as a game tree. Circle player aims to maximize the utility value, box aims to minimize the utility value. The bold edges represent the sequences of actions added to the restricted game.

the restricted game and there are finite pure strategies, the algorithm stops after at most $|\Pi_i| + |\Pi_{-i}|$ iterations. Each iteration is also polynomial, since it consists of solving the linear program and computing best responses. The relative performance of the double-oracle algorithm compared to solving the linear program for the original unrestricted game closely depends on the number of iterations required for convergence. In the worst case, the algorithm adds all pure strategies and solves the original game, although this is rarely the case in practice. Estimating the expected number of iterations needed for the double-oracle algorithm to converge, however, remains an open problem.

3.4.1 Towards Extensive-Form Games

The straightforward method of applying the double-oracle algorithm for EFGs is to use pure strategies defined in EFGs (i.e., assignments of action for each information set, or realization plans) and apply exactly the algorithm described in this section – i.e., iteratively add pure strategies from the unrestricted extensive-form game into the restricted game matrix. However, this can result in an exponential number of iterations and an exponentially large restricted game in the worst case. Our algorithm differs significantly from this idea since it directly operates on (more compact) sequences instead of full strategies.

4. Sequence-Form Double-Oracle Algorithm for Extensive-Form Games

We now describe our sequence-form double-oracle algorithm for solving extensive-form games with imperfect information. First, we give an informal overview of our algorithm. We use an example game depicted in Figure 2 to illustrate some of the key concepts. Afterwards, we formally define the restricted game and describe the key components of the algorithm, following by a full example run of our algorithm.

The overall scheme of our algorithm is based on the double-oracle framework described in the previous section. The main difference is that our algorithm uses the sequences to define the restrictions in the game tree. The restricted game in our model is defined by allowing players to use (i.e., to play with non-zero probability) only a subset of the sequences from the original unrestricted game. This restricted subset of sequences defines the subsets of reachable actions, nodes, and information sets from the original game tree. Consider our example in Figure 2. A restricted game can be defined by sequences $\emptyset, A, AC, AD$ for the circle player, and $\emptyset, x$ for the box player. These sequences represent actions allowed in the game, they define reachable nodes (using history we can reference them as $\emptyset, A, Ax, AxC, AxD$), and reachable information sets (I_1, I_2 for the circle player and the only information set $I_\square$ for the box player).

The algorithm iteratively adds new sequences of allowed actions into the restricted game, similarly to the double-oracle algorithm for normal-form games. The restricted game is solved as a standard zero-sum extensive-form game using the sequence-form linear program. Then a best response algorithm searches the original unrestricted game to find new sequences to add to the restricted game. When the sequences are added, the restricted game tree is expanded by adding all new actions, nodes, and information sets that are now reachable based on the new sets of allowed sequences. The process of solving the restricted game and adding new sequences iterates until no new sequences that improve the solution can be added.

There are two primary complications that arise when we use sequences instead of full strategies in the double-oracle algorithm, both due to the fact that sequences do not necessarily define actions in all information sets: (1) a strategy computed in the restricted game may not be a complete strategy in the original game, because it does not define behavior for information sets that are not in the restricted game, and (2) it may not be possible to play every action from a sequence that is allowed in the restricted game, because playing a sequence can depend on having a compatible sequence of actions for the opponent. In our example game tree in Figure 2, no strategy of the circle player in the restricted game specifies what to play in information sets I_3 and I_4. The consequence of the second issue is that some inner nodes of the original unrestricted game can (temporarily) become leafs in the restricted game. For example, the box player can add sequence y into the restricted game making node Ay a leaf in the restricted game, since there are no other actions of the circle player in the restricted game applicable in this node.

Our algorithm solves these complications using two novel ideas. The first idea is the concept of a *default pure strategy* (denoted $\pi_i^{\text{DEF}} \in \Pi_i$). Informally speaking, the algorithm assumes that each player has a fixed implicit behavior that defines what the player does by default in any information set that is not part of the restricted game. This is described by the default strategy π_i^{DEF}, which specifies an action for every information set. Note that this default strategy does not need to be represented explicitly (which could use a large amount of memory). Instead, it can be defined implicitly using rules, such as selecting the first action from a deterministic method for generating the ordered set of actions $A(h)$ in node h. We use the default pure strategies to map every strategy from the restricted game into a valid strategy in the full game. Specifically, the strategy in the original unrestricted game selects actions according to the probabilities specified by a strategy for the restricted game in every information set that is part of the restricted game, and for all other information sets it plays according to the default pure strategy. Recall our example in Figure 2, where the pure default strategy for the circle player can be $\langle A, C, E, G \rangle$ (i.e., selecting the leftmost action in each information set). Hence, a strategy in the original unrestricted game can use a strategy from the restricted game in information sets I_1 and I_2, and select pure actions in E, G in information sets I_3 and I_4 respectively.

The second key idea is to use *temporary utility values* for cases where there are no allowed actions that can be played in some node in the restricted game that is an inner node in the original game (so called *temporary leaf*). To ensure the correct convergence of

H	game-tree nodes / histories
$Z \subseteq H$	leafs / terminal states
π_i^{DEF}	implicit default pure strategy for player i
$r_i : \Sigma_i \mapsto \mathbb{R}$	realization plan of player i for a sequence
$\mathcal{C} : H \mapsto \mathbb{R}$	probability of reaching a node due to Nature play
$g_i : H \mapsto \mathbb{R}$	extension of the utility function to all nodes; $g_i(h) = u_i(h) \cdot \mathcal{C}(h)$ if $h \in Z$ and $g_i(h) = 0$ if h is not a terminal node ($h \notin Z$)
seq_i	sequence(s) of actions of player i leading to a node / a set of nodes / / an information set
$\mathsf{inf}_i : \Sigma_i \mapsto I_i$	an information set in which the last action of the sequence was executed

Table 1: An outline of the main symbols used in the paper.

the algorithm these temporary utilities must be assigned so that they provide a bound on the expected value gained by continuing the play from the given node. Our algorithm uses a value that corresponds to the expected outcome of continuing the game play, assuming the player making the choice in the temporary leaf uses the default strategy, while the opponent plays a best response. Assume we add sequence y for the box player into the restricted game in our example tree in Figure 2. The temporary utility value for node Ay would correspond to value -2, since the default strategy in information set I_3 is to play E for the circle player. In the next section we formally describe this method and prove the correctness of the algorithm given these temporary values.

We now describe in detail the key parts of our method. We first formally define the restricted game and methods for expanding the restricted game, including the details of both of the key ideas introduced above. Then we describe the algorithm for selecting the new sequences that are allowed in the next iteration. The decision of which sequences to add is based on calculating a best response in the original unrestricted game using game-tree search improved with additional pruning techniques. Finally, we discuss different variations of the main logic of the double-oracle algorithm that determines for which player(s) the algorithm adds new best-response sequences in the current iteration.

4.1 Restricted Game

This section formally defines the restricted game as a subset of the original unrestricted game. A restricted game can be fully specified by the set of allowed sequences. We define the sets of nodes, actions, and information sets as subsets of the original unrestricted sets based on the allowed sequences. We denote the original unrestricted game by a tuple $G = (N, H, Z, A, p, u, \mathcal{C}, \mathcal{I})$ and the restricted game by $G' = (N, H', Z', A', p, u', \mathcal{C}, \mathcal{I}')$. All sets and functions associated with the restricted game use prime in the notation; the set of players, and the functions p and $\mathcal{C}$ remain the same.

The restricted game is defined by a set of *allowed sequences* (denoted by $\Phi' \subseteq \Sigma$) that are returned by the best response algorithms. As indicated above, even an allowed sequence $\sigma_i \in \Phi'$ might not be playable to the full length due to missing compatible sequences of the opponent. Therefore, the restricted game is defined using the maximal compatible set of sequences $\Sigma' \subseteq \Phi'$ for a given set of allowed sequences Φ'. We define Σ' as the *maximal* *subset* of the sequences from Φ' such that:

$$\Sigma'_i \leftarrow \{\sigma_i \in \Phi'_i \ : \ \exists \sigma_{-i} \in \Phi'_{-i} \exists h \in H \ \forall j \in N \ \mathsf{seq}_j(h) = \sigma_j\} \quad \forall i \in N \tag{6}$$

Equation (6) means that for each player i and every sequence σ_i in Σ'_i, there exists a compatible sequence of the opponent σ_{-i} that allows the sequence σ_i to be executed in full (i.e., by sequentially executing of all the actions in these sequences σ some node h can be reached such that $\mathsf{seq}_j(h) = \sigma_j$ for all players $j \in N$).

The set of sequences Σ' fully defines the restricted game, because all other sets in the tuple G' can be derived from Σ'. A node h is in the restricted game if and only if the sequences that must be played to reach h are in the set Σ' for *both* players:

$$H' \leftarrow \{h \in H \ : \ \forall i \in N \ \mathsf{seq}_i(h) \in \Sigma'\} \tag{7}$$

If a pair of sequences is in Σ', then *all* nodes reachable by executing this pair of sequences are included in H'. Actions defined for a node h are in the restricted game if and only if playing the action in this node leads to a node that is in the restricted game:

$$A'(h) \leftarrow \{a \in A(h) \ : \ ha \in H'\} \quad \forall h \in H' \tag{8}$$

Nodes from the restricted game corresponding to inner nodes in the original unrestricted game may not be inner nodes in the restricted game. Therefore, the set of leaves in the restricted game is a union of leaf nodes of the original game and inner nodes from the original game that currently do not have a valid continuation in the restricted game, based on the allowed sequences:

$$Z' \leftarrow \left(Z \cap H'\right) \cup \{h \in H' \setminus Z \ : \ A'(h) = \emptyset\} \tag{9}$$

We explicitly differentiate between leaves in the restricted game that correspond to leaves in the original unrestricted game (i.e., $Z' \cap Z$) and leaves in the restricted game that correspond to inner nodes in the original unrestricted game (i.e., $Z' \setminus Z$), since the algorithm assigns temporary utility values to nodes in the latter case.

The information sets in the restricted game correspond to information sets in the original unrestricted game. If some node h belongs to an information set $I_{p(h)}$ in the original game, then the same holds in the restricted game. We define an information set to be a part of the restricted game if and only if at least one inner node that belongs to this information set is included in the restricted game:

$$\mathcal{I}'_i \leftarrow \{I_i \in \mathcal{I}_i \ : \ \exists h \in I_i \ h \in H' \setminus Z'\} \tag{10}$$

An information set in the restricted game $I_i \in \mathcal{I}'_i$ consists only of nodes that are in the restricted game – i.e., $\forall h \in I_i \ : \ h \in H'$.

Finally, we define the modified utility function u' for the restricted game. The primary reason for the modified utility function is to define the temporary utility values for leaves in the set $Z' \setminus Z$. Consider $h \in Z' \setminus Z$ to be a temporary leaf and player i to be the player acting in this node ($i = p(h)$). Moreover, let $u^*_i(h)$ be the expected outcome of the game starting from this node assuming both players are playing NE strategies in the original unrestricted game. The modified utility function u'_i for this leaf must return a value that is a lower bound on value $u_i^*(h)$. Due to the zero-sum assumption, this value represents an upper bound on value for the opponent $-i$. Setting the value this way ensures two things: (1) player $-i$ is likely to use sequences leading to node h in optimal strategies in the restricted game (since the modified utility value is an upper bound of an actual value), and (2) player i adds new sequences using best-response algorithms that prolong sequence $\mathsf{seq}_i(h)$ leading to node h if there are sequences that would yield better expected value than u'_i. Later we show a counterexample where setting the value otherwise can cause the algorithm to converge to an incorrect solution. We calculate the lower bound by setting the utility value so that it corresponds to the outcome in the original game if the player i continues by playing the default strategy π_i^{Def} and the opponent plays a best response δ_{-i}^{BR} to this default strategy. This is a valid lower bound since we consider only a single strategy for the player acting in node h, which correspond to the default strategy; considering other strategies could allow this player to improve the value of continuing from the node h. For all other leaf nodes $h \in Z' \cap Z$ we set $u'_i(h) \leftarrow u_i(h)$.

4.1.1 Solving the Restricted Game

The restricted game defined in this section is a valid zero-sum extensive-form game and it can be solved using the sequence-form linear programming described in Section 3. The algorithm computes a NE of the restricted game by solving a pair of linear programs using the restricted sets Σ', H', Z', I', and the modified utility function u'.

Each strategy from the restricted game can be translated to the original game by using the pure default strategy to extend the restricted strategy where it is not defined. Formally, if r'_i is a mixed strategy represented as a realization plan of player i in the restricted game, then we define the *extended strategy* $\overline{r}'_i$ to be a strategy identical to the strategy in the restricted game for sequences included in the restricted game, and correspond to the default strategy π_i^{Def} if a sequence is not included in the restricted game:

$$\overline{r}'_i(\sigma_i) \leftarrow \begin{cases} r'_i(\sigma_i) & \sigma_i \in \Sigma'_i \\ r'_i(\sigma'_i) \cdot \pi_i^{\text{Def}}(\sigma_i \setminus \sigma'_i) & \sigma_i \notin \Sigma'_i;\ \sigma'_i = \arg\max_{\sigma''_i \in \Sigma'_i;\ \sigma''_i \sqsubseteq \sigma_i} |\sigma''_i| \end{cases} \tag{11}$$

The realization plan of a sequence σ_i not allowed in the restricted game (i.e., $\sigma_i \notin \Sigma'_i$) is equal to the realization probability of the longest prefix of the sequence allowed in the restricted game (denoted by σ'_i), and setting the remaining part of the sequence (i.e., $\sigma_i \setminus \sigma'_i$) to correspond to the default strategy of player i. This computation is expressed as a multiplication of two probabilities, where we overload the notation and use $\pi_i^{\text{Def}}(\sigma_i \setminus \sigma'_i)$ to be 1 if the remaining part of the sequence σ_i corresponds to the default strategy of player i, and 0 otherwise.

In each iteration of the double-oracle algorithm one sequence-form LP is solved for each player to compute a pair of NE strategies in the restricted game. We denote these strategies as (r_i^*, r_{-i}^*) and $(\overline{r}_i^*, \overline{r}_{-i}^*)$ when they are extended to the original unrestricted game using the default strategies.

4.1.2 Expanding the Restricted Game

The restricted game is expanded by adding new sequences to the set Φ' and updating the remaining sets according to their definition. After adding new sequences, the algorithm calculates and stores the temporary utility values for leaves in $Z' \setminus Z$ so they can be used in the sequence-form LP.

After updating the restricted game, the linear programs are modified so that they correspond to the new restricted game. For all newly added information sets and sequences, new variables are created in the linear programs and the constraints corresponding to these information sets/sequences are created (Equations 2 and 4). Moreover, some of the constraints already existing in the linear program need to be updated. If a sequence σ_i is added to the set Σ'_i and the immediate prefix sequence (i.e., sequence $\sigma'_i \sqsubseteq \sigma_i$ such that $|\sigma'_i| + 1 = |\sigma_i|$) was already a part of the restricted game, then we need to update the constraint for information sets I_i for which $\sigma'_i = \mathsf{seq}_i(I_i)$ to ensure the consistency of the strategies (Equation 4), and the constraint corresponding to sequence σ'_i (Equation 2). In addition, the algorithm updates Equations (2) assigned to sequences of the opponent σ_{-i} for which $g(\sigma_i, \sigma_{-i}) \neq 0$. Finally, the algorithm updates all constraints that previously used utilities for temporary leaf nodes that are no longer leaf nodes in the restricted game after adding the new sequences.

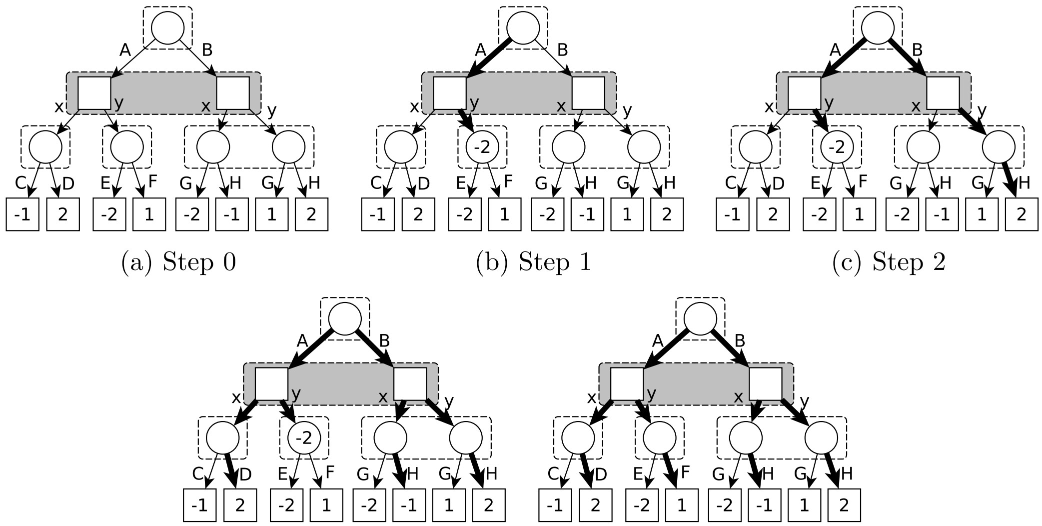
New sequences for each player are found using the best response sequence (BRS) algorithms described in Section 4.2. From the perspective of the sequence-form double-oracle algorithm, the BRS algorithm calculates a pure best response for player i against a fixed strategy of the opponent in the original unrestricted game. This pure best response specifies an action to play in each information set that is currently reachable given the opponent's extended strategy $\overline{r}^*_{-i}$. The best response can be formally defined as a pure realization plan r_i^{BR} that assigns only integer values 0 or 1 to the sequences. This realization plan is not necessarily a pure strategy in the original unrestricted game because there may not be an action specified for every information set. Specifically, there is no action specified for information sets that are not reachable (1) due to choices of player i, and (2) due to zero probability in the realization plan of the opponent $\overline{r}^*_{-i}$. Omitting these actions does not affect the value of the best response because these information sets are never reached; hence, for r_i^{BR} it holds that $\forall \overline{r}'_i \in \Delta_i \; u_i(r_i^{\mathrm{BR}}, \overline{r}^*_{-i}) \geq u_i(\overline{r}'_i, \overline{r}^*_{-i})$ and there exists a pure best response strategy $\pi_i^{\mathrm{BR}} \in \Pi_i$ such that $u_i(r_i^{\mathrm{BR}}, \overline{r}^*_{-i}) = u_i(\pi_i^{\mathrm{BR}}, \overline{r}^*_{-i})$. The sequences that are used in the best-response pure realization plan with probability 1 are returned by BRS algorithm and we call these the *best-response sequences*:

$$\{\sigma_i \in \Sigma_i \;:\; r_i^{\mathrm{BR}}(\sigma_i) = 1\} \tag{12}$$

4.1.3 Example Run of the Algorithm

We now demonstrate the sequence-form double-oracle algorithm on an example game depicted in Figure 3a. In our example, there are two players: circle and box. Circle aims to maximize the utility value in the leafs, box aims to minimize the utility value. We assume that choosing the leftmost action in each information set is the default strategy for both players in this game.

The algorithm starts with an empty set of allowed sequences in the restricted game $\Phi' \leftarrow \emptyset$; hence, the algorithm sets the current pair of $(\overline{r}^*_i, \overline{r}^*_{-i})$ strategies to be equivalent to $(\pi_i^{\mathrm{Def}}, \pi_{-i}^{\mathrm{Def}})$. Next, the algorithm adds new sequences that correspond to the best response to the default strategy of the opponent; in our example the best response sequences for the circle player are $\{\emptyset, A, AD\}$, and $\{\emptyset, y\}$ for the box player. These sequences are added



(a) Step 0 (b) Step 1 (c) Step 2

(d) Step 3 (e) Step 4

Figure 3: Example of the steps of the sequence-form double-oracle algorithm in a two-player zero-sum game, where circle player aims to maximize the utility value, box aims to minimize the utility value. Bold edges correspond to the sequences of actions added into the restricted game. The dashed boxes indicate the information sets.

to the set of allowed sequences Φ'. Next, the set of sequences of the restricted game Σ' is updated. The maximal compatible set of sequences from set Φ' cannot contain sequence AD because the compatible sequence of the box player (i.e., x in this case) is not allowed in the restricted game yet and sequence AD cannot be fully executed. Moreover, by adding sequences A and y, the restricted game will contain node Ay for which actions E and F are defined in the original unrestricted game. However, there is no continuation in the current restricted game yet; hence, this node is a temporary leaf, belongs to $Z' \setminus Z$, and the algorithm needs to define a new value for a modified utility function u' for this node. The value $u'(Ay)$ is equal to -2 and corresponds to the outcome of the game if the circle player continues by playing the default strategy and the box player plays the best response. To complete the first step of the algorithm we summarize the nodes and information sets included in the restricted game; H' contains 3 nodes (the root, the node after playing an action A and the node Ay), and two information sets (the information set for node Ay is not added into the restricted game, because this node is now a leaf in the restricted game). Playing the sequences A and y with probability 1 is the Nash equilibrium of the restricted game. The situation is depicted in Figure 3b, the sequences in Σ' are shown as bold edges.

The algorithm proceeds further and the complete list of steps of the algorithm is summarized in Table 2. In the second iteration, new sequences B and BH are added into the restricted game. The box player does not add new sequences in this iteration because y is the best response to the extended equilibrium strategy of the circle player – i.e., playing sequences A, AC, AE with probability 1. NE in the updated restricted game changes to playing sequences B, BH and sequence y, all with probability 1. In the third iteration the situation changes and the box player adds sequence x, while there are no new sequences added for the circle player. After adding sequence x, sequence AD also becomes a part of the set $\Sigma'_{\bigcirc}$ as it can now be fully executed due to adding the compatible sequence x. NE in the restricted game is now fully mixed, the sequences starting with A and with B are played in a ratio of 3 : 4, x and y in a ratio of 4 : 3. In the fourth iteration, the algorithm adds sequence AF to the restricted game (the best response for the circle player), which removes the assigned value $u'(Ay)$ since the node no longer belongs to set Z'. The algorithm stops after four iterations. No other sequences are added into the restricted game, the solution of the restricted game (r_i^*, r_{-i}^*) can be translated to the solution in the original unrestricted game, and $(\overline{r}_i^*, \overline{r}_{-i}^*)$ is Nash equilibrium of the original game.

Iteration	$r_{\bigcirc}^{\text{BR}}$	$r_{\Box}^{\text{BR}}$	$\Sigma'_{\bigcirc}$	$\Sigma'_{\Box}$
1.	$\emptyset, A, AD$	$\emptyset, y$	$\emptyset, A$	$\emptyset, y$
2.	$\emptyset, B, BH$	$\emptyset, y$	$\emptyset, A, B, BH$	$\emptyset, y$
3.	$\emptyset, B, BH$	$\emptyset, x$	$\emptyset, A, AD, B, BH$	$\emptyset, y, x$
4.	$\emptyset, A, AF$	$\emptyset, y$	$\emptyset, A, AD, AF, B, BH$	$\emptyset, y, x$

Table 2: Steps of the sequence-form double-oracle algorithm applied to the example.

Consider now a small modification of the example game where there is a utility value of -3 in the leaf following action F (i.e., node AyF). In this case, the algorithm does not need to add sequence AF (nor AE) to the restricted game because it does not improve the value of the restricted game. Note that this modified example game shows why the algorithm needs to set the utility values for nodes in $Z' \setminus Z$. If the algorithm simply uses the unmodified utility function, then the node Ay will be treated as if it had zero utility value. This value overestimates the outcome of any actual continuation following this node in the original game for the circle player and since sequences AE or AF will never be a part of the best response for the circle player, the algorithm can converge to an incorrect solution.

4.2 Best-Response Sequence Algorithm

The purpose of the best-response sequence (BRS) algorithm is to generate new sequences that will be added to the restricted game in the next iteration, or to prove that there is no best response with better expected value that uses sequences currently not allowed in the restricted game. Throughout this section we use the term *searching player* to represent the player for whom the algorithm computes the best response sequences. We refer to this player as i.

The BRS algorithm calculates the expected value of a pure best response to the opponent's strategy $\overline{r}_{-i}^*$. The algorithm returns both the set of best-response sequences as well as the expected value of the strategy against the extended strategy of the opponent.

The algorithm is based on a depth-first search that traverses the original unrestricted game tree. The behavior of the opponent $-i$ is fixed to the strategy given by the extended realization plan $\overline{r}_{-i}^*$. To save computation time, the best-response algorithms use branch and bound during the search for best-response sequences. The algorithm uses a bound on the expected value for each inner node, denoted by λ. This bound represents the minimal utility value that the node currently being evaluated needs to gain in order to be a part

Require: i - searching player, h - current node, I_i^k - current information set, $\overline{r}'_{-i}$ - opponent's strategy, Min/MaxUtility - bounds on utility values, λ - lower bound for a node h

1: $w \leftarrow \overline{r}_{-i}(\mathsf{seq}_{-i}(h)) \cdot \mathcal{C}(h)$
2: **if** $h \in Z$ **then**
3:    **return** $u_i(h) \cdot w$
4: **else if** $h \in Z' \setminus Z$ **then**
5:    **return** $u'_i(h) \cdot w$
6: **end if**
7: sort $a \in A(h)$ based on probability $w_a \leftarrow \overline{r}'_{-i}\left(\mathsf{seq}_{-i}(ha)\right) \cdot \mathcal{C}(ha)$
8: $v^h \leftarrow 0$
9: **for** $a \in A(h),\ w_a > 0$ **do**
10:    $\lambda' \leftarrow \lambda - \left[v^h + (w - w_a) \cdot \text{MaxUtility}\right]$
11:    **if** $\lambda' \leq w_a \cdot \text{MaxUtility}$ **then**
12:       $v' \leftarrow \text{BRS}_i(ha, \lambda')$
13:       **if** $v' = -\infty$ **then**
14:          **return** $-\infty$
15:       **end if**
16:       $v^h \leftarrow v^h + v'$
17:       $w \leftarrow w - w_a$
18:    **else**
19:       **return** $-\infty$
20:    **end if**
21: **end for**
22: **return** v^h

Figure 4: BRS_i in the nodes of other players.

of a best-response sequence. Using this bound during the search, the algorithm is able to prune branches that will certainly not be part of any best-response sequence. The bound λ is set to *MinUtility* for the root node.

We distinguish 2 cases in the search algorithm: either the algorithm is evaluating an information set (or more specifically a node h) assigned to the searching player i, or the node is assigned to one of the other players (either to the opponent, player $-i$, or it is a chance node). The pseudocode for these two cases is depicted in Figures 4 and 5.

4.2.1 Nodes of the Opponent

We first describe the case used when the algorithm evaluates node h assigned to either the opponent of the searching player or to Nature (see Figure 4). The main idea is to calculate the expected utility for this node according to the (fixed) strategy of the player. The strategy is known because it is either given by the extended realization plan $\overline{r}^*_{-i}$, or by the stochastic environment ($\mathcal{C}$). Throughout the algorithm, the variable w represents the probability of this node based on the realization probability of the opponent and stochastic environment (line 1). This value is iteratively decreased by values w_a that represent realization probabilities of the currently evaluated action $a \in A(h)$. Finally, v_h is the expected utility value for this node.

The algorithm evaluates actions in the descending order according to the probability of being played (based on $\overline{r}'_{-i}$ and $\mathcal{C}$; lines 9–21). First, we calculate a new lower bound

λ' for the successor ha (line 10). The new lower bound is the minimal value that must be returned from the recursive call $\text{BRS}_i(ha)$ under the optimistic assumption that all the remaining actions will yield the maximum possible utility. If the lower bound does not exceed the maximum possible utility in the game, the algorithm is executed recursively on the successors (line 12). Note that the algorithm does not evaluate branches with zero realization probability (line 9).

There are 3 possibilities for pruning in this part of the search algorithm. The first pruning is possible if the currently evaluated node is a leaf in the restricted game, but this node is an inner node in the original node (i.e., $h \in Z' \setminus Z$; line 5). The algorithm can directly use the value from the modified utility function u' in this case, since it is calculated as a best response of the searching player against the default strategy of the opponent that will be applied in the successors of node h since $h \in Z'$. Secondly, a cut-off also occurs if the new lower bound for a successor is larger than the maximum possible utility in the game, since this value can never be obtained in the successor (line 19). Finally, a cut-off occurs if there was a cut-off in one of the successors (line 14).

4.2.2 Nodes of the Searching Player

In nodes assigned to the searching player, the algorithm evaluates every action in each state that belongs to the current information set. The algorithm traverses the states in the descending order according to the probability of occurrence given the strategies of the opponent and Nature (line 8). Similar to the previous case, in each iteration the algorithm calculates a new lower bound for the successor node (line 17). The new lower bound λ' is the minimal value that must be returned from the recursive call $\text{BRS}_i(h'a)$ in order for the action a to be selected as the best action for this information set under the optimistic assumption that this action yields the maximum possible utility value after applying it in each of the remaining states in this information set. The algorithm performs a recursive call (line 20) only for an action that still could be the best in this information set (i.e., the lower bound does not exceed the maximal possible utility in the game). Note that if a cut-off occurs in one of the successors, the currently evaluated action a can no longer be the best action in this information set. Hence, v_a is set to $-\infty$ and action a will not be evaluated for any of the remaining nodes. When the algorithm determines which action will be selected as the best one in an information set, it evaluates only this action for all remaining nodes in the information set. Finally, the algorithm stores the values for the best action for all nodes in this information set (line 30). These are reused if the same information set is visited again (i.e., the algorithm reaches a different node h' from the same information set I_i; line 5).

A cut-off occurs in this part of the search algorithm if the maximal possible value v_a^h is smaller than the lower bound λ after evaluating node h. This means that regardless of which action will be selected as the best action in this information set, the lower bound for node h will not be reached; hence, the cut-off occurs (line 27). If a cut-off occurs in an information set, this information set cannot be reached again and the sequences of the searching player leading to this information set cannot be a part of the best response. This is due to propagating the cut-off to at least one previous information set of the searching player, otherwise there will be no tight lower bound set (the bound is first set only in the

Require: i - searching player, h - current node, I_i^k - current information set, r_{-i} - opponent's strategy, Min/MaxUtility - bounds on utility values, $\lambda-$ lower bound for a node h

if $h \in Z$ **then**
    return $u_i(h) \cdot \overline{r}'_{-i}(\mathsf{seq}_{-i}(h)) \cdot \mathcal{C}(h)$
end if
if v^h is already calculated **then**
    return v^h
end if
$H' \leftarrow \{h';\ h' \in I_i\}$
sort H' descending according to value $\overline{r}_{-i}(\mathsf{seq}_{-i}(h')) \cdot \mathcal{C}(h')$
$w \leftarrow \sum_{h' \in H'} \overline{r}_{-i}(\mathsf{seq}_{-i}(h')) \cdot \mathcal{C}(h')$
$v_a \leftarrow 0\ \ \forall a \in A(h)$; $maxAction \leftarrow \emptyset$
for $h' \in H'$ **do**
    $w_{h'} \leftarrow \overline{r}'_{-i}(\mathsf{seq}_{-i}(h')) \cdot \mathcal{C}(h')$
    for $a \in A(h')$ **do**
        if $maxAction$ is empty **then**
            $\lambda' \leftarrow w_{h'} \cdot$MinUtility
        else
            $\lambda' \leftarrow (v_{maxAction} + w \cdot \text{MinUtility}) - (v_a + (w - w_{h'}) \cdot \text{MaxUtility})$
        end if
        if $\lambda' \leq w_{h'} \cdot \text{MaxUtility}$ **then**
            $v_a^{h'} \leftarrow \text{BRS}_i(h'a, \lambda')$
            $v_a \leftarrow v_a + v_a^{h'}$
        end if
    end for
    $maxAction \leftarrow \arg\max_{a \in A(h')} v_a$
    $w \leftarrow w - w_{h'}$
    if h was evaluated $\wedge$ $\left(\max_{a \in A(h)} v_a^h < \lambda\right)$ **then**
        return $-\infty$
    end if
end for
store $v^{h'}_{maxAction}$ as $v^{h'}$ $\ \forall h' \in H'$
return $v^h_{maxAction}$

Figure 5: BRS_i in the nodes of the searching player.

information sets of the searching player). Therefore, there exists at least one action of the searching player that will never be evaluated again (after a cut-off, the value v_a for this action is set to $-\infty$) and cannot be selected as the best action in the information set. Since we assume perfect recall, all nodes in information set I_i share the same sequence of actions $\mathsf{seq}_i(I_i)$; hence, no node $h' \in I_i$ can be reached again.

4.3 Main Loop Alternatives

We now introduce several alternative formulations for the main loop of the sequence-form double-oracle algorithm. The general approach in the double-oracle algorithm is to solve the restricted game to find the equilibrium strategy for each player, compute the best responses in the original game for both of the players, and continue with the next iteration. However, the sequence-form LP is formulated in our double-oracle scheme in such a way that on each iteration the algorithm can solve the restricted game only from the perspective of a single player i. In other words, we formulate a single LP as described in Section 3.3 that computes the optimal strategy of the opponent in the restricted game (player $-i$), and then compute the best response of player i to this strategy. This means that on each iteration we can select a specific player i, for whom we compute the best response in this iteration. We call this selection process the *player-selection policy.*

There are several alternatives for the player-selection policy that act as a domain-independent heuristics in double-oracle algorithm. We consider three possible policies: (1) the *standard double-oracle player-selection policy* of selecting both players on each iteration, (2) an *alternating policy*, where the algorithm selects only one player and switches between the players regularly (player i is selected in one iteration, player $-i$ is selected in the following iteration), and finally (3) a *worse-player-selection policy* that selects the player who currently has the worse bound on the solution quality. At the end of an iteration the algorithm selects the player i for whom the upper bound on utility value is further away from the current value of the restricted game. More formally,

$$\arg\max_{i \in N} \left| V_i^{UB} - V_i^{LP} \right| \tag{13}$$

where V_i^{LP} is the last calculated value of the restricted game for player i. The intuition behind this choice is that either this bound is precise and there are some missing sequences of this player in the restricted game that need to be added, or the upper bound is overestimated. In either case, the best-response sequence algorithm should be run for this player in the next iteration, either to add new sequences or to tighten the bound. In case of a tie, the alternating policy is applied in order to guarantee regular switching of the players. We experimentally compare these policies to show their impact on the overall performance of the sequence-form double-oracle algorithm (see Section 6).

5. Theoretical Results

In this section we prove that our sequence-form double-oracle algorithm will always converge to a Nash equilibrium of the original unrestricted game. First, we formally define the strategy computed by the best-response sequence (BRS) algorithm, then we prove lemmas about the characteristics of the BRS strategies, and finally we prove the main convergence result. Note that variations of the main loop described in Section 4.3 do not affect the correctness of the algorithm as long as the player-selection policy ensures that if no improvement is made by the BRS algorithm for one player that the BRS algorithm is run for the opponent on the next iteration.

Lemma 5.1 *Let r'_{-i} be a realization plan of player $-i$ in some restricted game G'. $BRS(r'_{-i})$ returns sequences corresponding to a realization plan r_i^{BR} in the unrestricted game, such that r_i^{BR} is part of a pure best response strategy to $\overline{r}'_{-i}$. The value returned by the algorithm is the value of executing the pair of strategies $u_i(\overline{r}'_{-i}, r_i^{BR})$.*

Proof $BRS(r'_{-i})$ searches the game tree and selects the action that maximizes the value of the game for player i in all information sets I_i assigned to player i reachable given the strategy of the opponent $\overline{r}'_{-i}$. In the opponent's nodes, it calculates the expected value according to r'_{-i} where it is defined and the value according to the pure action of the default strategy π_{-i}^{DEF} where r'_{-i} is not defined. In chance nodes, it returns the expected value of the node as the sum of the values of the successor nodes weighted by their probabilities. In each node h, if the successors have the maximal possible value for i then node h also has the maximal possible value for i (when playing against $\overline{r}'_{-i}$). The selections in the nodes that belong to i achieves this maximal value; hence, they form a best response to strategy $\overline{r}'_{-i}$. □

For brevity we use $v(BRS(r'_{-i}))$ to denote the value returned by the BRS algorithm, which is equal to $u_i(\overline{r}'_{-i}, r_i^{BR})$.

Lemma 5.2 *Let r'_{-i} be a realization plan of player $-i$ in some restricted game G' and let V_i^* be the value of the original unrestricted game G for player i, then*

$$v(BRS(r'_{-i})) \geq V_i^*. \tag{14}$$

Proof Lemma 5.1 showed that $v(BRS(r'_{-i}))$ is a value of the best response against $\overline{r}'_{-i}$ which is a valid strategy in the original unrestricted game G. If $v(BRS(r'_{-i})) < V_i^*$ then V_i^* cannot be the value of the game since player $-i$ has a strategy $\overline{r}'_{-i}$ that achieves better utility, which is a contradiction. □

Lemma 5.3 *Let r'_{-i} be a realization plan of player $-i$ that is returned by the LP for some restricted game G' and let V_i^{LP} be the value of the restricted game returned by the LP, then*

$$v(BRS(r'_{-i})) \geq V_i^{LP}. \tag{15}$$

Proof The realization plan r'_{-i} is part of the Nash equilibrium strategy in a zero-sum game that guarantees value V_i^{LP} in G'. If the best response computation in the original unrestricted game G selects only the actions from restricted game G', it creates the best response in game G' as well obtaining value V_i^{LP}. If the best response selects an action that is not allowed in the restricted game G', there are two cases.

Case 1: The best response strategy uses an action in a temporary leaf $h \in Z' \setminus Z$. Player i makes the decision in the leaf, because otherwise the value of the temporary leaf would be directly returned by BRS. The value of the temporary leaf has been underestimated for player i in the restricted game by the modified utility function u' and it is over-estimated in the BRS computation as the best response to the default strategy π_{-i}^{DEF}. The value of the best response can only increase by including this action.

Case 2: The best response strategy uses an action not allowed in G' in an internal node of the restricted game $H' \setminus Z'$. This can occur in nodes assigned to player i, because the actions of player $-i$ going out of G' have probability zero in $\overline{r}'_{-i}$. BRS takes the action with maximum value in the nodes assigned to player i, so the reason for selecting an action leading outside G' is that it has greater or equal value to the best action in G'. □

Lemma 5.4 *Under the assumptions of the previous lemma, if $v(BRS(r'_{-i})) > V_i^{LP}$ then it returns sequences that are added to the restricted game G' in the next iteration.*

Proof Based on the proof of the previous Lemma, BRS for player i can improve over the value of the LP (V_i^{LP}) only by selecting an action a that is not present in G' but is performed in a node h that is included in G' (in which i makes decision). Let (σ_i, σ_{-i}) be the pair of sequences leading to h. Then in the construction of the restricted game for the next iteration, sequence σ_{-i} is the sequence that ensures that $\sigma_i a$ can be executed in full and will be part of the new restricted game. □

Note, that Lemmas 5.2 and 5.4 would not hold if the utility values u' for temporary leaves ($h \in Z' \setminus Z$) are set arbitrarily. The algorithm sets the values in temporary leaf h as if the player $p(h)$ continues by playing the default strategy and the opponent ($-p(h)$) is playing the best response. If the utility values for the temporary leaves are set arbitrarily and used in the BRS algorithms to speed-up the calculation as proposed (see the algorithm in Figure 4, line 5), then Lemma 5.2 does not need to hold in cases where the value in node h strictly overestimates the optimal expected value for player $p(h)$. In this case, the best-response value of the opponent may be lower than the optimal outcome,

$$v\left(BRS(r_{p(h)})\right) < V^*_{-p(h)} \tag{16}$$

On the other hand, if the BRS algorithm does not use the temporary values u' for such a node, then Lemma 5.4 is violated because the best-response value will be strictly higher for player $-p(h)$ even though no new sequences are to be added into the restricted game.

Theorem 5.5 *The sequence-form double-oracle algorithm for extensive-form games described in the previous section terminates if and only if*

$$v(BRS(r'_{-i})) = -v(BRS(r'_i)) = V_i^{LP} = V_i^*, \tag{17}$$

which always happens after a finite number of iterations (because the game is finite), and strategies $(\overline{r}'_i, \overline{r}'_{-i})$ are a Nash equilibrium of the original unrestricted game.

Proof First we show that the algorithm continues until all equalities (17) hold. If $v(BRS(r'_{-i})) \neq -v(BRS(r'_i))$ then from Lemma 5.2 and Lemma 5.4 we know that for some player i it holds that $BRS(r'_{-i}) > V_i^{LP}$, so the restricted game in the following iteration is larger by at least one action and the algorithm continues. In the worst case, the restricted game equals the complete game $G' = G$, and it cannot be extended any further. In this case the BRS cannot find a better response then V_i^* and the algorithm stops due to Lemma 5.4.

If the condition in the theorem holds the algorithm has found a NE in the complete game, because from Lemma 5.1 we know that $r^{BR}_{-i} = BRS(r'_i)$ is the best response to $\overline{r}'_i$ in the complete game. However, if the value of the best response to a strategy in a zero-sum game is the value of the game, then the strategy $\overline{r}'_i$ is optimal and it is part of a Nash equilibrium of the game. □

6. Experiments

We now present our experimental evaluation of the performance of the sequence-form double-oracle algorithm for EFGs. We compare our algorithm against two state-of-the-art baselines, the full sequence-form LP (referred to as FullLP from now on), and Counterfactual Regret Minimization (CFR). The first baseline is the standard exact method for solving sequence-form EFG, while CFR is one of the leading approximate algorithms applied to EFG. Our experimental results demonstrate the advantages of the double-oracle algorithm on three different classes of realistic EFGs. We also test the impact of the different variants of the main loop of the algorithm described in Section 4.3.

We compare three variants of the sequence-form double-oracle algorithm: (1) DO-b is a variant in which the best-responses are calculated for both players in each iteration; (2) DO-sa calculates the best-response for a single player on each iteration according to a simple alternating policy; and (3) DO-swp is a variant in which the best-response is calculated for a single player according to the worse-player selection policy. For all of the variants of the double-oracle algorithm we use the same default strategy where the first action applicable in a state is played by default.

Since there is no standardized collection of zero-sum extensive-form games for benchmark purposes, we use several specific games to evaluate the double-oracle algorithm and to identify the strengths and weaknesses of the algorithm. The games were selected to evaluate the performance under different conditions, so the games differ in the maximal utility the players can gain, in the causes of the imperfect information, and in the structure of the information sets. One of the key characteristics that affects the performance of the double-oracle algorithm is the relative size of the support of Nash equilibria (i.e., the number of sequences used in a NE with non-zero probability). If there does not exist a NE with small support, the algorithm must necessarily add a large fraction of the sequences into the restricted game to find a solution, mitigating the advantages of the double-oracle approach.

We present results for two types of games where the double-oracle significantly outperforms the FullLP on all instances: a search game motivated by border patrol and Phantom Tic-Tac-Toe. We also present results on a simplified version of poker for which the double-oracle algorithm does not always improve the computation time. However, the FullLP also has limited scalability due to larger memory requirements and cannot find solutions for larger variants of poker, while the double-oracle algorithm is able to solve these instances.

Our principal interest is in developing new generic methods for solving extensive-form games. Therefore, we implemented the algorithm in a generic framework for modeling arbitrary extensive-form games.[1] The algorithms do not use any domain-specific knowledge in the implementation, and do not rely on any specific ordering of the actions. The drawbacks of this generic implementation are higher memory requirements and additional overhead for the algorithms. A domain-specific implementation could improve the performance by eliminating some of the auxiliary data structures. We run all of the experiments using a single thread on an Intel i7 CPU running at 2.8 GHz. Each of the algorithms was given a maximum of 10 GB of memory for Java heap space. We used IBM CPLEX 12.5 for solving the linear programs, with parameter settings to use a single thread and the barrier solution algorithm.

In addition to runtimes, we analyze the speed of convergence of the double-oracle algorithms and compare it to one of the state-of-the-art approximative algorithms, Counterfactual Regret Minimization (CFR). We implemented CFR in a domain independent way

1. Source code is available at the home pages of the authors.

based on the pseudocode in the work of Lanctot (2013, p. 22). In principle, it is sufficient for CFR to maintain only a set of information sets and apply the no-regret learning rule in each information set. However, maintaining and traversing such a set effectively in a domain independent manner could be affected by our implementation of generic extensive-form games data structures (i.e., generating applicable actions in the states of the game, applying the actions, etc.). Therefore we use an implementation where CFR traverses the complete game tree that is held in memory to maintain the fairness of the comparison, and to guarantee the maximal possible speed of convergence of the CFR algorithm. The time necessary to build the game tree is *not* included in the computation time of CFR.

6.1 Test Domains

Search Games Our first test belongs to the class of search (or pursuit-evasion) games, often used in experimental evaluation of double-oracle algorithms (McMahan et al., 2003; Halvorson et al., 2009). The search game has two players: the *patroller* (or the defender) and the *evader* (or the attacker). The game is played on a directed graph (see Figure 6), where the evader aims to cross safely from a starting node (E) to a destination node (D). The defender controls two units that move in the intermediate nodes (the shaded areas) trying to capture the evader by occupying the same node as the evader. During each turn both players move their units simultaneously from the current node to an adjacent node, or the units stay in the same location. The only exception is that the evader cannot stay in the two leftmost nodes. If a pre-determined number of turns is made without either player winning, the game is a draw. This is an example of a win-tie-loss game and the utility values are from the set $\{-1, 0, 1\}$.

Players are unaware of the location and the actions of the other player with one exception – the evader leaves tracks in the visited nodes that can be discovered if the defender visits the nodes later. The game also includes an option for the evader to avoid leaving the tracks using a special move (a *slow move*) that requires two turns to simulate the evader covering the tracks.

Figure 6 shows examples of the graphs used in the experiments. The patrolling units can move only in the shaded areas (P1,P2), and they start at any node in the shaded areas. Even though the graph is small, the concurrent movement of all units implies a large branching factor (up to ≈ 50 for one turn) and thus large game trees (up to $\approx 10^{11}$ nodes). In the experiments we used three different graphs, varied the maximum number of turns of the game (from 3 to 7), and we altered the ability of the attacker to perform the slow moves (labeled *SA* if the slow moves are allowed, *SD* otherwise).

Phantom Tic-Tac-Toe The second game is a blind variant of the well-known game of Tic-Tac-Toe (e.g., used in Lanctot et al., 2012). The game is played on a 3×3 board, where two players (cross and circle) attempt to place 3 identical marks in a horizontal, vertical, or diagonal row to win the game. In the blind variant, the players are unable to observe the opponent's moves and each player only knows that the opponent made a move and it is her turn. Moreover, if a player tries to place her mark on a square that is already occupied by an opponent's mark, the player learns this information and can place the mark in some other square. Again, the utility values of this game are from the set $\{-1, 0, 1\}$.

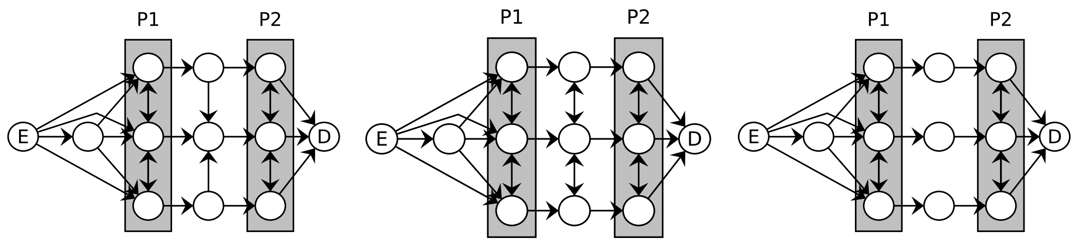


Figure 6: Three variants of the graph used in the experiments on the search game; we refer to them as G1 (left), G2 (middle), and G3 (right).

The uncertainty in phantom Tic-Tac-Toe makes the game large ($\approx 10^9$ nodes). In addition, since one player can try several squares before her move is successful, the players do not necessarily alternate in making their moves. This rule makes the structure of the information sets rather complicated and since the opponent never learns how many attempts the first player actually performed, a single information set can contain nodes at different depths in the game tree.

Poker Games Poker is frequently studied in the literature as an example of a large extensive-form game with imperfect information. We include experiments with a simplified two-player poker game inspired by Leduc Hold'em.

In our version of poker, each player starts with the same amount of chips and both players are required to put some number of chips in the pot (called the *ante*). In the next step, the Nature player deals a single card to each player (the opponent is unaware of the card) and the betting round begins. A player can either *fold* (the opponent wins the pot), *check* (let the opponent make the next move), *bet* (being the first to add some amount of chips to the pot), *call* (add the amount of chips equal to the last bet of the opponent into the pot), or *raise* (match and increase the bet of the opponent). If no further raise is made by any of the players, the betting round ends, the Nature player deals one card on the table, and the second betting round with the same rules begins. After the second betting round ends, the outcome of the game is determined – a player wins if: (1) her private card matches the table card and the opponent's card does not match, (2) none of the players' cards matches the table card and her private card is higher than the private card of the opponent, or (3) the opponent folds. The utility value is the amount of chips the player has won or lost. If no player wins, the game is a draw and the pot is split.

In the experiments we alter the number of types of the cards (from 3 to 4; there are 3 types of cards in Leduc), the number of cards of each type (from 2 to 3; set to 2 in Leduc), the maximum length of sequence of raises in a betting round (ranging from 1 to 4; set to 1 in Leduc), and the number of different sizes of bets (i.e., amount of chips added to the pot) for *bet/raise* actions (ranging from 1 to 4; set to 1 in Leduc).

6.2 Results

Search Games The results for the search game scenarios show that the sequence-form double-oracle algorithm is particularly successful when applied to games where NEs with small support exist. Figure 7 shows a comparison of the running times for FullLP and variants of the double-oracle algorithm (note the logarithmic y-scale). All variants of the

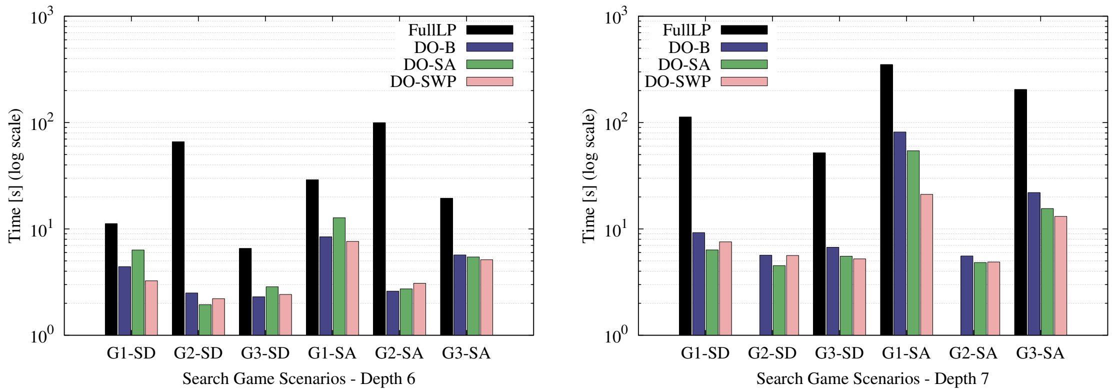


Figure 7: Comparison of the running times on 3 different graphs with either slow moves allowed (SA) or disallowed (SD), the depth is set to 6 (left subfigure) or 7 (right subfigure). Missing values for the FullLP algorithm indicate that the algorithm runs out of memory.

double-oracle algorithm are several orders of magnitude faster than FullLP. This is most apparent on the fully-connected graph (G2) that generates the largest game tree. When slow moves are allowed and the depth is set to 6, it takes almost 100 seconds for FullLP to solve the instance of the game but all variants of the double-oracle algorithms solve the game in less than 3 seconds. Moreover, when the depth is increased to 7, FullLP was unable to solve the game due to the memory constraints, while the fastest variant DO-swp solved the game in less than 5 seconds. Similar results were obtained for the other graphs.

The graph G1 induced a game that was the most difficult for the double-oracle algorithm: when the depth is set to 7, it takes almost 6 minutes for FullLP to solve the instance, while the fastest variant DO-swp solved the game in 21 seconds. The reason is that even though the game tree is not the largest, there is a more complex structure of the information sets. This is due to limited compatibility among the sequences of the players; when the patrolling unit P1 observes the tracks in the top-row node, the second patrolling unit P2 can capture the evader only in the top-row node, or in the middle-row node.

Comparing the different variants of the sequence-form double-oracle algorithm does not show consistent results. There is no variant consistently better in this game since all the double-oracle variants are typically able to compute a Nash equilibrium very quickly. However, DO-swp is often the fastest and for some settings the difference is quite significant. The speed-up this variant offers is most apparent on the G1 graph. On average through all instances of the search game, DO-sa uses 92.59% of the computation time of DO-b, and DO-swp uses 88.25%.

Table 3 shows a breakdown of the cumulative computation time spent in different components of the double-oracle algorithm: solving the restricted game (LP), calculating best responses (BR), and creating a valid restricted game after selecting new sequences to add (Validity). The results show that due to the size of the game, the computation of the best-response sequences takes the majority of the time (typically around 75% on larger instances), while creating the restricted game and solving it takes only a small fraction of the total time. It is also noticeable that the size of the final restricted game is very small

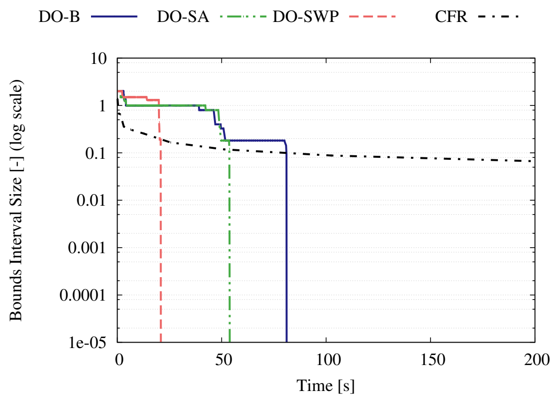


Figure 8: Convergence of variants of the double-oracle algorithm and CFR on the search game domain: y-axis displays the current approximation error.

Algorithm	Overall [s]	LP [s]	BR [s]	Validity [s]	Iterations	$\lvert\Sigma'_1\rvert(\frac{\lvert\Sigma'_1\rvert}{\lvert\Sigma_1\rvert})$	$\lvert\Sigma'_2\rvert(\frac{\lvert\Sigma'_2\rvert}{\lvert\Sigma_2\rvert})$
FullLP	351.98	–	–	–	–	–	–
DO-b	81.51	6.97	63.39	10.58	187	252 (17.22%)	711 (0.26%)
DO-sa	54.32	5.5	39.11	9.09	344	264 (18.05%)	649 (0.24%)
DO-swp	21.15	1.93	16.28	2.47	209	193 (13.19%)	692 (0.25%)

Table 3: Cumulative running times for different components of the double-oracle algorithm, iterations, and size of the restricted game in terms of the number of sequences compared to the size of the complete game. The results are shown for scenario G1, depth 7, and allowed slow moves.

compared to the original game, since the number of sequences for the second player (the defender) is less than 1% (there are 273,099 sequences for the defender).

Finally, we analyze the convergence rate of the variants of the double-oracle algorithm. The results are depicted in Figure 8, where the size of the interval given by the bounds V_i^{UB} and V_i^{LB} defines the current error of the double-oracle algorithm as $|V_i^{UB} - V_i^{LB}|$. The convergence rate of the CFR algorithm is also depicted. The error of CFR is calculated in the same way, as a sum of the best-response values to the current mean strategies from the CFR algorithm. We can see that all variants of the double-oracle algorithm perform similarly – the error drops very quickly to 1 and a few iterations later each version of the algorithm quickly converges to an exact solution. These results show that in this game the double-oracle algorithm can very quickly find the correct sequences of actions and compute an exact solution, in spite of the size of the game. In comparison, the CFR algorithm can also quickly learn the correct strategies in most of the information sets, but the convergence has a very long tail. After 200 seconds, the error of CFR is equal to 0.0657 and it is dropping very slowly (0.0158 after 1 hour). The error of CFR is quite significant considering the value of the game in this case ($-0.\overline{3333}$).

Phantom Tic-Tac-Toe The results on Phantom Tic-Tac-Toe confirm that this game is also suitable for the sequence-form double-oracle algorithm. Due to the size of the game, both baseline algorithms (the FullLP and CFR) ran out of memory and were not able

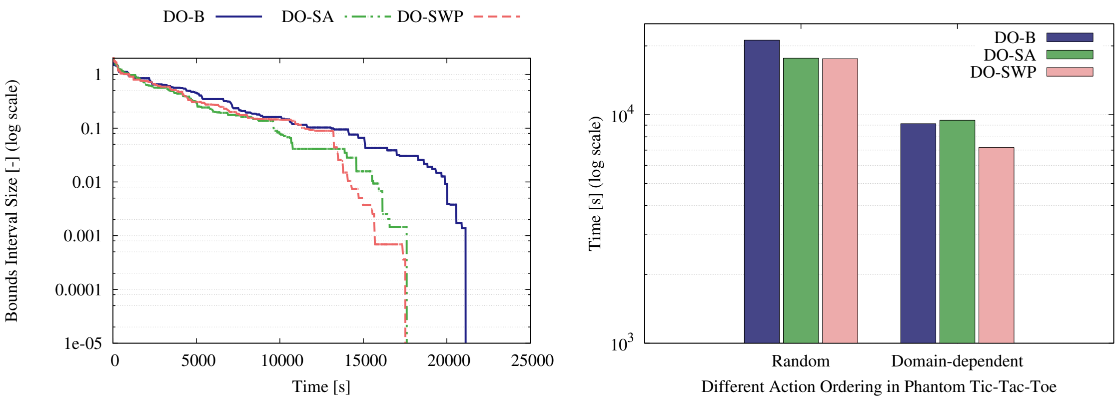


Figure 9: (left) Comparison of the convergence rate of the double-oracle variants for Phantom Tic-Tac-Toe; (right) Comparison of the performance of the double-oracle variants for Phantom Tic-Tac-Toe when domain-specific move ordering and default strategy is used.

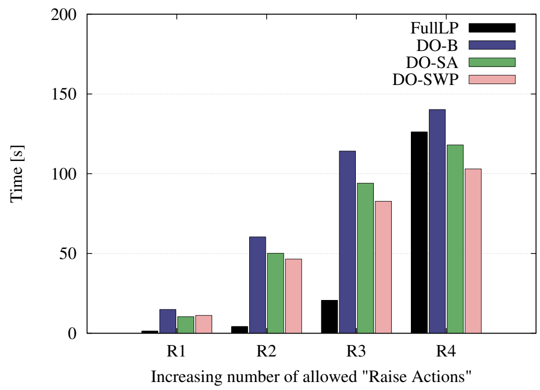
Algorithm	Overall [s]	LP [s]	BR [s]	Validity [s]	Iterations	$\lvert\Sigma'_1\rvert(\frac{\lvert\Sigma'_1\rvert}{\lvert\Sigma_1\rvert})$	$\lvert\Sigma'_2\rvert(\frac{\lvert\Sigma'_2\rvert}{\lvert\Sigma_2\rvert})$
FullLP	N/A	–	–	–	–	–	–
DO-b	21,197	2,635	17,562	999	335	7,676 (0.60%)	10,095 (0.23%)
DO-sa	17,667	2,206	14,560	900	671	7,518 (0.59%)	9,648 (0.22%)
DO-swp	17,589	2,143	14,582	864	591	8,242 (0.65%)	8,832 (0.20%)

Table 4: Cumulative running times for different components of the double-oracle algorithm for the game of Phantom Tic-Tac-Toe.

to solve the game. Therefore, we only compare the times for different variants of the double-oracle algorithm. Figure 9 (left subfigure) shows the overall performance of all three variants of the double-oracle algorithm in the form of a convergence graph. We see that the performance of two of the variants is similar, with the performance of DO-sa and DO-swp almost identical. On the other hand, the results show that DO-b converges significantly slower.

The time breakdown of the variants of the double-oracle algorithm is shown in Table 4. Similarly to the previous case, the majority of the time ($\approx$ 83%) is spent in calculating the best responses. Out of all variants of the double-oracle algorithm, the DO-swp variant is the fastest one. It converged in significantly fewer iterations compared to the DO-sa variant (iterations are twice as expensive in the DO-b variant).

We now present the results that demonstrate the potential of combining the sequence-form double-oracle algorithm with domain-specific knowledge. Every variant of the double-oracle algorithm can use a move ordering based on domain-specific heuristics. The move ordering determines the default strategy (recall that our algorithm uses the first action as the default strategy for each player), and the direction of the search in the best response algorithms. By replacing the randomly generated move ordering with a heuristic one that chooses better actions first, the results show a significant improvement in the performance of all of the variants (see Figure 9, right subfigure), even though there are no changes to the rest of the algorithm. Each variant was able to solve the game in less than 3 hours, and it took 2 hours for the fastest DO-swp variant.



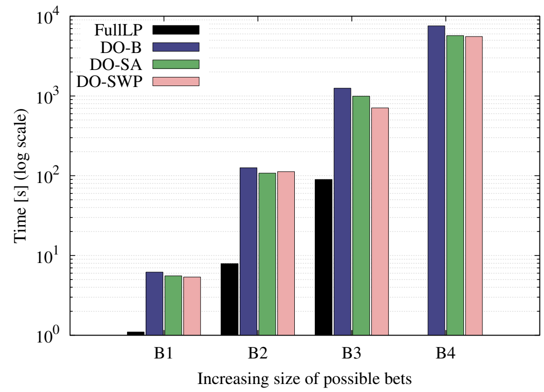


Figure 10: Comparison of the running times on different variants of the simplified poker game. The left subfigure shows the computation times with an increasing number of raise actions allowed, the right subfigure shows the computation times with an increasing number of different bet sizes for raise/bet actions.

Poker Games Poker represents a game where the double-oracle algorithms do not perform as well and the sequence-form LP is often faster on smaller instances. One significant difference compared to the previous games is that the size of the NE support is larger (around 5% of sequences for larger instances). Secondly, the game trees of poker games are relatively shallow and the only imperfect information in the game is due to Nature. As a result, the double-oracle algorithms require a larger number of iterations to add more sequences into the restricted game (up to 10% of all sequences for a player are added even for the largest poker scenarios) in order to find the exact solution. However, with increasing depth and/or branching factor, the size of the game grows exponentially and FullLP is not able to solve the largest instances due to the memory constraints.

Figure 10 shows the selected results for simplified poker variants. The results in the left subfigure show the computation times with increasing depth of the game by allowing the players to re-raise (players are allowed to re-raise their opponent a certain number of times). The remaining parameters are fixed to 3 types of cards, 2 cards of each type, and 2 different betting sizes. The size of the game grows exponentially, with the number of possible sequences increasing to 210,937 for each player for the $R4$ scenario. The computation time for FullLP is directly related to the size of the tree and increases exponentially with the increasing depth (note that there is a standard y scale). On the other hand, the increase is less dramatic for all of the variants of the double-oracle algorithm. The DO-swp variant is the fastest for the largest scenario – while FullLP solved this instance in 126 seconds, it took only 103 seconds for DO-swp. Finally, FullLP is not able to solve the games if we increase the length to $R5$ due to memory constraints, while the computation time of all of the double-oracle algorithms increases only marginally.

The right subfigure of Figure 10 shows the increase in computation time with an increasing number of different bet sizes for raise/bet actions. The remaining parameters were fixed to 4 types of cards, 3 cards of each type, and 2 raise actions allowed. Again, the game grows exponentially with the increasing branching factor. The number of sequences increases up to 685,125 for each player for the $B4$ scenario, and the computation time of

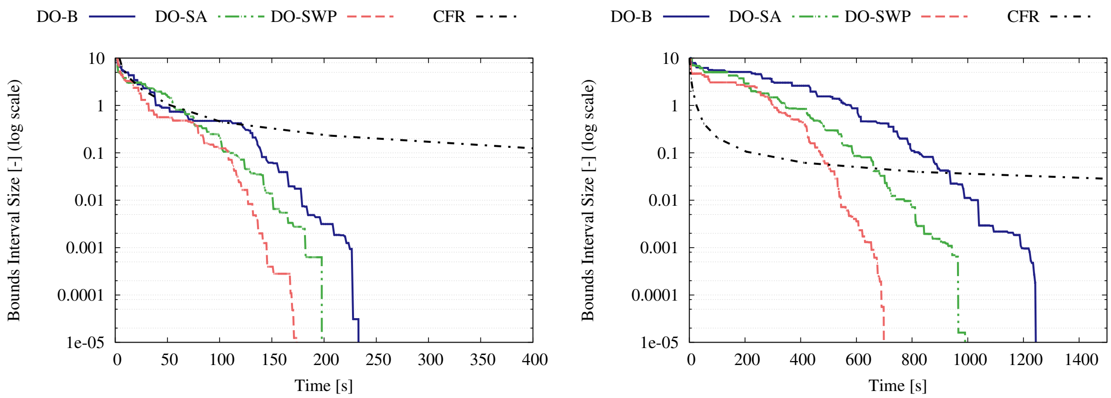


Figure 11: Comparison of the convergence of the variants of the double-oracle algorithm and CFR for two variants of the simplified poker with 4 types of cards, and 3 cards of each type. There are 4 raise actions allowed, 2 different bet sizes in the left subfigure; there are 2 raise actions allowed, 3 different bet sizes in the right subfigure.

Algorithm	Overall [s]	LP [s]	BR [s]	Validity [s]	Iterations	$\lvert\Sigma'_1\rvert(\frac{\lvert\Sigma'_1\rvert}{\lvert\Sigma_1\rvert})$	$\lvert\Sigma'_2\rvert(\frac{\lvert\Sigma'_2\rvert}{\lvert\Sigma_2\rvert})$
FullLP	278.18	–	–	–	–	–	–
DO-b	234.60	149.32	56.04	28.61	152	6,799 (1.81%)	6,854 (1.83%)
DO-sa	199.24	117.71	51.25	29.59	289	6,762 (1.80%)	6,673 (1.78%)
DO-swp	182.68	108.95	48.25	24.8	267	6,572 (1.75%)	6,599 (1.76%)

Table 5: Cumulative running times for different components of the double-oracle algorithm, iterations, and sizes of the restricted game in terms of the number of sequences compared to the size of the complete game. The results are shown for poker scenario with 4 raise actions allowed, 2 different betting values, 4 types of cards, and 3 cards of each type.

all algorithms increases exponentially as well (note logarithmic y scale). The results show that even with the increasing branching factor, the double-oracle variants tend to be slower than solving the FullLP. However, while the FullLP ran out of memory for the largest $B4$ setting, all of the double-oracle variants were able to find the exact solution using less memory.

Comparing the different variants of the double-oracle algorithm using the convergence graph (see Figure 11) and the decomposition of the computation times (see Table 5) shows that DO-swp is the fastest variant in the selected scenario (and in nearly all of poker scenarios). Decomposition of the overall time shows that the majority of the computation time is spent in solving the restricted game LP (up to 65%). The decomposition also shows that DO-swp is typically faster due to the lower number of iterations. In addition, the final size of the restricted game is typically the smallest for this variant. On average over all instances of the poker games, DO-sa uses 86.57% of the computation time of DO-b, and DO-swp uses 82.3% of the computation time.

Convergence in poker games is slower compared to search games of similar size (note the logarithmic scale in Figure 11). Comparing the double-oracle algorithm variants with CFR shows an interesting result in the left subfigure. Due to the size of the game, the speed of the CFR convergence is nearly the same as for the double-oracle algorithms during the first iterations. However, while the double-oracle algorithms continue to converge at roughly the same rate and are able to find an exact solution, the error of the CFR algorithm decreases very slowly. In the scenario depicted in the left subfigure, the CFR algorithm converged to an error of 0.1212 (the value of the game in this case is ≈ -0.09963) after 400 seconds. After 1 hour, the error dropped to 0.0268. For scenarios with more shallow game trees and larger branching factor, the convergence of CFR is faster at the beginning compared to the double-oracle algorithms (right subfigure of Figure 11). However, the main disadvantage of CFR having a long tail for convergence is still the case and the error after 1600 seconds is still over 0.0266 (the value of this game is ≈ -0.09828).

6.3 Discussion of the Results

The experimental results support several conclusions. The results demonstrate that the sequence-form double-oracle algorithm is able to compute an exact solution for much larger games compared to the state-of-the-art exact algorithm based on the sequence-form linear program. Moreover, we have experimentally shown that there are realistic games where only a small fraction of sequences are necessary to find a solution of the game. In these cases, the double-oracle algorithms also significantly speed up the computation time. Our results indicate that the DO-swp variant is typically the fastest, but not in all cases. By selecting the player that currently has the worse bound on performance, the DO-swp version can add more important sequences, or prove that there are not any better sequences and adjust the upper bound on the value faster.

Comparing the speed of convergence of the double-oracle algorithms with the state-of-the-art approximative algorithm CFR showed that CFR quickly approximates the solution during the first iterations. However, the convergence of CFR has a very long tail and CFR is not able to find an exact solution for larger games in a reasonable time. Another interesting observation is that for some games the convergence rate of the double-oracle algorithms and CFR is similar in the first iterations, and while the double-oracle algorithms continue at this rate and find an exact solution, the long tail convergence remains for CFR. This is despite the fact that our implementation of CFR has an advantage of having the complete game tree including the states for all histories in memory.

Unfortunately, it is difficult to characterize the exact properties of the games for which the double-oracle algorithms perform better in terms of computation time compared to the other algorithms. Certainly, the double-oracle algorithm is not suitable for games were the only equilibria have large support due to the necessity of large number of iterations. However, having a small support equilibrium is not a sufficient condition. This is apparent due to two graphs shown in the poker experiments, where either the depth of the game tree or the branching factor was increased. Even though the game grows exponentially and the size of the support decreases to $\approx 2.5\%$ in both cases, the behavior of the double-oracle algorithms is quite different. Our conjecture is that games with longer sequences suit the double-oracle algorithms better, since several actions that form the best-response sequences can be added during a single iteration. This contrasts with shallow game trees with large branching factors, where more iterations are necessary to add multiple actions. However, a deeper analysis to identify the exact properties of the games that are suitable is an open question that must be analyzed for normal-form games first.

7. Conclusion

We present a novel exact algorithm for solving two player zero-sum extensive-form games with imperfect information. Our approach combines the compact sequence-form representation for extensive-form games with the iterative algorithmic framework of double-oracle methods. This integrates two successful approaches for solving large scale games that have not yet been brought together for the general class of games that our algorithm addresses. The main idea of our algorithm is to restrict the game by allowing players to play only a restricted set of sequences from the available sequences of actions, and to iteratively expand the restricted game over time using fast best-response algorithms. Although in the worst case the double-oracle algorithm may need to add all possible sequences, the experimental results on different domains prove that the double-oracle algorithm can find an exact Nash equilibrium prior to constructing the full linear program for the complete game. Therefore, the sequence-form double-oracle algorithm reduces the main limitation of the sequence-form linear program—memory requirements—and it is able to solve much larger games compared to state-of-the-art methods. Moreover, since our algorithm is able to identify the sequences of promising actions without any domain-specific knowledge, it can also provide a significant runtime improvements.

The proposed algorithm also has another crucial advantage compared to the current state of the art. The double-oracle framework offers a decomposition of the problem of computing a Nash equilibrium into separate sub-problems, including the best-response algorithms, the choice of the default strategy, and the algorithms for constructing and solving the restricted game. We developed solutions for all of these sub-problems in a domain-independent manner. However, we can also view our algorithm as a more general framework that can be specialized with domain-specific components that take advantage of the structure of specific problems to improve the performance of these sub-problems. This can lead to substantial improvements in the speed of the algorithm, the number of iterations, as well as reducing the final size of the restricted game. We demonstrated the potential of the domain-specific approach on the game of Phantom Tic-Tac-Toe. Another example is that fast best-response algorithms that operate on the public tree (i.e., a compact representation of games with publicly observable actions; see Johanson, Bowling, Waugh, & Zinkevich, 2011) can be exploited for games like poker. Finally, our formal analysis identifies the key properties that these domain-specific implementations need to satisfy to guarantee the convergence to the correct solution of the game.

Our algorithm opens up a large number of directions for future work. It represents a new class of methods for solving extensive-form games with imperfect information that operates very differently than other common approaches (e.g., counterfactual regret minimization), and many possible alternatives to improve the performance of the algorithm remain to be investigated. Examples include more sophisticated calculation of utility values for the temporary leaves, alternative strategies for expanding the restricted game, and removing unused sequences from the restricted game. A broader analysis of using the sequence-form double-oracle algorithm as an approximation technique should be performed, possibly by exploring alternative approximative best-response algorithms based on sampling (e.g., Monte Carlo) techniques.

There are also several theoretical questions that could be investigated. First, the performance of the double-oracle algorithm depends strongly on the number of iterations and sequences that need to be added. However, the theoretical question regarding the expected number of iterations and thus the speed of the convergence of the double-oracle algorithm have not been explored even for simpler game models (e.g., games in the normal form). An analysis of these simpler models is needed to identify the general properties of games where the double-oracle methods tend to be faster and to identify the optimal way of expanding the restricted game.

Acknowledgements

Earlier versions of this paper were published at the European Conference on Artificial Intelligence (ECAI) (Bosansky, Kiekintveld, Lisy, & Pechoucek, 2012) and the conference on Autonomous Agents and Multi Agent Systems (AAMAS) (Bosansky, Kiekintveld, Lisy, Cermak, & Pechoucek, 2013). The major additions to this full version include (1) a novel, more detailed description of all parts of the algorithm, (2) introduction and analysis of different policies for the player selection in the main loop of the double-oracle algorithm, (3) new experiments on the phantom tic-tac-toe domain together with a more thorough analysis of the experimental results on all domains, including the analysis of the convergence of the algorithm, (4) experimental comparison with CFR, and finally (5) extended analysis of related work.

This research was supported by the Czech Science Foundation (grant no. P202/12/2054) and by U.S. Army Research Office (award no. W911NF-13-1-0467).

References

Barnhart, C., Johnson, E. L., Nemhauser, G. L., Savelsbergh, M. W. P., & Vance, P. H. (1998). Branch-And-Price: Column Generation for Solving Huge Integer Programs. *Operations Research*, *46*, 316–329.

Bosansky, B., Kiekintveld, C., Lisy, V., Cermak, J., & Pechoucek, M. (2013). Double-oracle Algorithm for Computing an Exact Nash Equilibrium in Zero-sum Extensive-form Games. In *Proceedings of International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 335–342.

Bosansky, B., Kiekintveld, C., Lisy, V., & Pechoucek, M. (2012). Iterative Algorithm for Solving Two-player Zero-sum Extensive-form Games with Imperfect Information. In *Proceedings of the 20th European Conference on Artificial Intelligence (ECAI)*, pp. 193–198.

Cermak, J., Bosansky, B., & Lisy, V. (2014). Practical Performance of Refinements of Nash Equilibria in Extensive-Form Zero-Sum Games. In *Proceedings of European Conference on Artificial Intelligence (ECAI)*, pp. 201–206.

Dantzig, G., & Wolfe, P. (1960). Decomposition Principle for Linear Programs. *Operations Research*, *8*, 101–111.

Ganzfried, S., & Sandholm, T. (2013). Improving Performance in Imperfect-Information Games with Large State and Action Spaces by Solving Endgames. In *Computer*

Poker and Imperfect Information Workshop at the National Conference on Artificial Intelligence (AAAI).

Gibson, R., Lanctot, M., Burch, N., Szafron, D., & Bowling, M. (2012). Generalized Sampling and Variance in Counterfactual Regret Minimization. In *Proceedings of the 26th AAAI Conference on Artificial Intelligence*, pp. 1355–1361.

Halvorson, E., Conitzer, V., & Parr, R. (2009). Multi-step Multi-sensor Hider-seeker Games. In *Proceedings of the Joint International Conference on Artificial Intelligence (IJCAI)*, pp. 159–166.

Hoda, S., Gilpin, A., Peña, J., & Sandholm, T. (2010). Smoothing Techniques for Computing Nash Equilibria of Sequential Games. *Mathematics of Operations Research*, *35*(2), 494–512.

Jain, M., Conitzer, V., & Tambe, M. (2013). Security Scheduling for Real-world Networks. In *Proceedings of the International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 215–222.

Jain, M., Korzhyk, D., Vanek, O., Conitzer, V., Tambe, M., & Pechoucek, M. (2011). Double Oracle Algorithm for Zero-Sum Security Games on Graph. In *Proceedings of the 10th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 327–334.

Johanson, M., Bowling, M., Waugh, K., & Zinkevich, M. (2011). Accelerating Best Response Calculation in Large Extensive Games. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 258–265.

Koller, D., & Megiddo, N. (1992). The Complexity of Two-Person Zero-Sum Games in Extensive Form. *Games and Economic Behavior*, *4*, 528–552.

Koller, D., Megiddo, N., & von Stengel, B. (1996). Efficient Computation of Equilibria for Extensive Two-Person Games. *Games and Economic Behavior*, *14*(2), 247–259.

Koller, D., & Megiddo, N. (1996). Finding Mixed Strategies with Small Supports in Extensive Form Games. *International Journal of Game Theory*, *25*, 73–92.

Kreps, D. M., & Wilson, R. (1982). Sequential Equilibria. *Econometrica*, *50*(4), 863–94.

Lanctot, M. (2013). *Monte Carlo Sampling and Regret Minimization for Equilibrium Computation and Decision Making in Large Extensive-Form Games*. Ph.D. thesis, University of Alberta.

Lanctot, M., Gibson, R., Burch, N., Zinkevich, M., & Bowling, M. (2012). No-Regret Learning in Extensive-Form Games with Imperfect Recall. In *Proceedings of the 29th International Conference on Machine Learning (ICML 2012)*, pp. 1–21.

Lanctot, M., Waugh, K., Zinkevich, M., & Bowling, M. (2009). Monte Carlo Sampling for Regret Minimization in Extensive Games. In *Advances in Neural Information Processing Systems (NIPS)*, pp. 1078–1086.

Lee, C.-S., Wang, M.-H., Chaslot, G., Hoock, J.-B., Rimmel, A., Teytaud, O., Tsai, S.-R., Hsu, S.-C., & Hong, T.-P. (2009). The Computational Intelligence of Mogo Revealed in Taiwans Computer Go Tournaments. *IEEE Transactions on Computational Intelligence and AI in Games*, *1*, 73–89.

Letchford, J., & Vorobeychik, Y. (2013). Optimal Interdiction of Attack Plans. In *Proceedings of the 12th International Conference on Automonous Agents and Multiagent Systems (AAMAS)*, pp. 199–206.

Lisy, V., Kovarik, V., Lanctot, M., & Bosansky, B. (2013). Convergence of Monte Carlo Tree Search in Simultaneous Move Games. In *Advances in Neural Information Processing Systems (NIPS)*, Vol. 26, pp. 2112–2120.

McMahan, H. B. (2006). *Robust Planning in Domains with Stochastic Outcomes, Adversaries, and Partial Observability.* Ph.D. thesis, Carnegie Mellon University.

McMahan, H. B., & Gordon, G. J. (2007). A Fast Bundle-based Anytime Algorithm for Poker and other Convex Games. *Journal of Machine Learning Research - Proceedings Track*, *2*, 323–330.

McMahan, H. B., Gordon, G. J., & Blum, A. (2003). Planning in the Presence of Cost Functions Controlled by an Adversary. In *Proceedings of the International Conference on Machine Learning*, pp. 536–543.

Miltersen, P. B., & Sørensen, T. B. (2008). Fast Algorithms for Finding Proper Strategies in Game Trees. In *Proceedings of Symposium on Discrete Algorithms (SODA)*, pp. 874–883.

Miltersen, P. B., & Sørensen, T. B. (2010). Computing a Quasi-Perfect Equilibrium of a Two-Player Game. *Economic Theory*, *42*(1), 175–192.

Pita, J., Jain, M., Western, C., Portway, C., Tambe, M., Ordonez, F., Kraus, S., & Parachuri, P. (2008). Deployed ARMOR protection: The Application of a Game-Theoretic Model for Security at the Los Angeles International Airport. In *Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 125–132.

Ponsen, M. J. V., de Jong, S., & Lanctot, M. (2011). Computing Approximate Nash Equilibria and Robust Best-Responses Using Sampling. *Journal of Artificial Intelligence Research (JAIR)*, *42*, 575–605.

Sandholm, T. (2010). The State of Solving Large Incomplete-Information Games, and Application to Poker. *AI Magazine*, *special issue on Algorithmic Game Theory*, 13–32.

Selten, R. (1975). Reexamination of the Perfectness Concept for Equilibrium Points in Extensive Games. *International Journal of Game Theory*, *4*, 25–55.

Selten, R. (1965). Spieltheoretische Behandlung eines Oligopolmodells mit Nachfragetrgheit [An oligopoly model with demand inertia]. *Zeitschrift für die Gesamte Staatswissenschaft*, *121*, 301–324.

Shafiei, M., Sturtevant, N., & Schaeffer, J. (2009). Comparing UCT versus CFR in Simultaneous Games. In *IJCAI Workshop on General Game Playing*.

Shieh, E., An, B., Yang, R., Tambe, M., Baldwin, C., Direnzo, J., Meyer, G., Baldwin, C. W., Maule, B. J., & Meyer, G. R. (2012). PROTECT : A Deployed Game Theoretic System to Protect the Ports of the United States. In *International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 13–20.

Shoham, Y., & Leyton-Brown, K. (2009). *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*. Cambridge University Press.

Tambe, M. (2011). *Security and Game Theory: Algorithms, Deployed Systems, Lessons Learned*. Cambridge University Press.

Tsai, J., Rathi, S., Kiekintveld, C., Ordóñez, F., & Tambe, M. (2009). IRIS - A Tool for Strategic Security Allocation in Transportation Networks Categories and Subject Descriptors. In *Proceedings of the 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, pp. 37–44.

van Damme, E. (1984). A Relation Between Perfect Equilibria in Extensive Form Games and Proper Equilibria in Normal Form Games. *Game Theory*, *13*, 1–13.

van Damme, E. (1991). *Stability and Perfection of Nash Equilibria*. Springer-Verlag.

von Stengel, B. (1996). Efficient Computation of Behavior Strategies. *Games and Economic Behavior*, *14*, 220–246.

Wilson, R. (1972). Computing Equilibria of Two-Person Games From the Extensive Form. *Management Science*, *18*(7), 448–460.

Zinkevich, M., Johanson, M., Bowling, M., & Piccione, C. (2008). Regret Minimization in Games with Incomplete Information. *Advances in Neural Information Processing Systems (NIPS)*, *20*, 1729–1736.

Zinkevich, M., Bowling, M., & Burch, N. (2007). A New Algorithm for Generating Equilibria in Massive Zero-Sum Games. In *Proceedings of National Conference on Artificial Intelligence (AAAI)*, pp. 788–793.