Using Double-oracle Method and Serialized Alpha-Beta Search for Pruning in Simultaneous Move Games

Branislav Bošanský, Viliam Lisý, Jiří Čermák, Roman Vítek, Michal Pěchouček
Agent Technology Center, Dept. of Computer Science and Engineering
Faculty of Electrical Engineering, Czech Technical University in Prague
{bosansky, lisy, cermak, vitek, pechoucek}@agents.fel.cvut.cz

Abstract

We focus on solving two-player zero-sum extensive-form games with perfect information and simultaneous moves. In these games, both players fully observe the current state of the game where they simultaneously make a move determining the next state of the game. We solve these games by a novel algorithm that relies on two components: (1) it iteratively solves the games that correspond to a single simultaneous move using a double-oracle method, and (2) it prunes the states of the game using bounds on the sub-game values obtained by the classical Alpha-Beta search on a serialized variant of the game. We experimentally evaluate our algorithm on the Goofspiel card game, a pursuit-evasion game, and randomly generated games. The results show that our novel algorithm typically provides significant running-time improvements and reduction in the number of evaluated nodes compared to the full search algorithm.

1 Introduction

Non-cooperative game theory provides a formal mathematical framework for analyzing the behavior of interacting self-interested agents. Recent advancements in computational game theory led to many successful applications of game-theoretic models in security domains [Tambe, 2011] and improved the performance of computer game-playing in a variety of board and computer games (such as Go [Enzenberger *et al.*, 2010], Poker [Gibson *et al.*, 2012], or Ms. Pac-Man [Nguyen and Thawonmas, 2012]).

We focus on fundamental algorithmic advances for solving large instances of an important general class of games: two player, zero-sum extensive-form games (EFGs) with perfect information and simultaneous moves. Games in this class capture sequential interactions that can be visualized as a game tree. The nodes correspond to the states of the game, in which both players act simultaneously. We can represent these situations using the normal form (i.e., as matrix games), where the values are computed from the succeeding sub-games. Many well known games are instances of this class, including card games such as Goofspiel [Rhoads and Bartholdi, 2012], variants of pursuit-evasion games [Littman, 1994], as well as several games from general game-playing competition [Genesereth *et al.*, 2005].

Zero-sum simultaneous move games can be solved in polynomial time by backward induction [Buro, 2003]. However, this straightforward approach is not applicable for larger instances of games. Therefore, variants of game-tree search algorithms without theoretical guarantees have been mostly used in practice [Kovarsky and Buro, 2005; Teytaud and Flory, 2011]. Only recently, the backward induction algorithm has been improved with the pruning techniques by [Saffidine *et al.*, 2012]. However, the presented experimental results did not show significant running time improvement compared to the full search. Alternatively, simultaneous move games can be addressed as generic imperfect-information games and solved using the compact sequence form [Koller *et al.*, 1996; von Stengel, 1996], however, this approach cannot exploit the specific structure of the game, thus having significantly higher memory and running-time requirements compared to the backward induction.

One of the main components of the backward induction algorithm is solving matrix games in each state of the game. The key to improve the performance of the algorithm is to solve these games using as few evaluated sub-games as possible, i.e., by considering only a limited number of moves for each player in each state of the game. This can be achieved using a *double-oracle* algorithm that was highly successful in practice when solving large normal-form games [McMahan *et al.*, 2003; Jain *et al.*, 2011; Vanek *et al.*, 2012]. The main idea of this method is to create a restricted game in which the players have a limited number of allowed strategies, and then iteratively expand the restricted game by adding best responses to the current solution of the restricted game.

In this paper we present a novel algorithm for solving zero-sum extensive-form games with simultaneous moves based on the double-oracle method. Additionally, we reduce the number of evaluated sub-games in each stage by estimating the sub-game values using the standard Alpha-Beta search on serialized variants of the game. After providing technical background, we describe our novel algorithm and give theoretical guarantees of its correctness and computational complexity. In contrast to the currently best known pruning algorithm for this class of games [Saffidine *et al.*, 2012], our experimental results show dramatic improvements, often requiring less than 7% of the time of the full search.

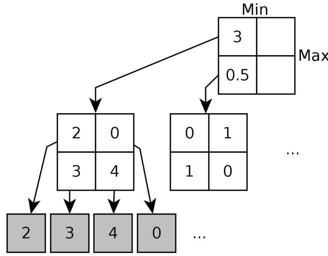


Figure 1: Example of a game tree for zero-sum simultaneous move game. Each stage is represented as a matrix game, solution of which is propagated to the predecessor.

2 Technical Background and Related Work

A perfect-information simultaneous-moves game can be visualized as a game tree, where each node consist of a normal-form game (or a *matrix game*, see Figure 1) representing simultaneous action of both players. We refer to the nodes as the *stages* of the game and thus term the matrix game representing this situation as a *stage game*; we denote S to be a set of all stages of the game. We are concerned with zero-sum games; hence, we specifically name the two players as *Max* and *Min*, where the first one is maximizing, the second one is minimizing the utility value. In each stage $s \in S$, both players have some number of possible actions to play; the *Max* player has n_s actions, indexed by i, and *Min* player has m_s actions to play in stage s, indexed by j. We omit the lower s index, if the stage is clear from the context. Performing a joint action (i, j) in stage s leads to another stage s' that we denote $s' = s_{i,j}$.

To solve a game is to find a strategy profile (one strategy for each player) that satisfies certain conditions. Nash equilibrium (NE) is the most common solution concept; a strategy profile is in NE if all players play the best response to the strategies of the opponents. For simultaneous-moves EFG, a *pure strategy* is a selection of an action in each stage, and a *mixed strategy* is a set of probability distributions over actions in each stage. The actions in a stage, that are used with non-zero probability, form the *support of NE*. The expected utility value gained by each player when both players follow the strategies from NE is termed the *value of the game*. We use $u(s)$ to denote both the utility value of leaf s in the game tree (when s is terminal) as well as the value of the sub-game of stage s (i.e., sub-tree of the original game tree rooted in s).

2.1 Solving EFGs with Simultaneous Moves

A zero-sum simultaneous-moves EFG can be solved by the backward-induction algorithm [Buro, 2003] – the algorithm searches through the game tree in the depth-first manner, and after computing the values of all the succeeding sub-games, it solves each stage game (i.e., computes a NE in each stage), and propagates the calculated value to the predecessor. The result of the backward-induction algorithm is a refinement of NE called *subgame-perfect Nash equilibrium* – the selected strategy profile forms the NE in each sub-game. NE in a zero-sum stage game can be found as a solution of a linear program (e.g., see [Shoham and Leyton-Brown, 2009, p. 88]).

Simultaneous Moves Alpha-Beta Search

The currently best known pruning method for the backward-induction algorithm for simultaneous-moves games was proposed by Saffidine et al. [2012]. The main idea of the algorithm is to reduce the number of the recursive calls of the backward-induction algorithm by removing dominated actions in every stage game. For each successor $s_{i,j}$ of a stage s, the algorithm maintains bounds on the game value $u(s_{i,j})$. The lower and upper bounds represent the threshold values, for which neither action i nor action j is dominated by any other action in the current stage game s. These bounds are calculated by linear programs in the stage s given existing exact values (or appropriate bounds) of the utility values of all the other successors of the stage s. If they form an empty interval (the lower bound is higher than the upper bound), a pruning occurs and the dominated action is not considered in this stage any more.

The algorithm proposed in this paper differs in two key aspects: (1) instead of removing the dominated strategies we start with a small restricted game that is iteratively expanded by adding best-response actions in this stage; and (2) we use the standard Alpha-Beta search on serialized variants of the game in order to obtain bounds on the game values of the successors rather than calculating these bounds solely from information in this stage.

3 Method

Our algorithm enhances the backward-induction algorithm with the double-oracle method: (1) in each stage of the game it creates a *restricted stage game* by allowing the players to play only one action in this stage, then (2) it iteratively solves the restricted stage game, and (3) expands the restricted game by adding, for each player, a best response to the current strategy of the opponent in the solution of the restricted game. When neither of players can improve the solution of the restricted game by the best response, the algorithm has found the solution of the full stage game, since both players are playing the best responses against each other.

While seeking for the best response, the algorithm estimates the quality of the actions using the classical Alpha-Beta search on *serialized variants of the game*. The serialized variants are modifications of the original game, where the players move sequentially and the second player to move knows what action the first player has made in the stage. Depending on the order of the serialization, solving this modified game yields either the upper or the lower bound on the value of the original game: if the *Max* moves second after the move has been made by the *Min* player, the maximizing player has an advantage (*Max* selects different best response to each action of the *Min* player) and the upper bound is calculated, if the serialization is reversed, the lower bound is calculated.

Definitions

In the description of the algorithm we use the following notation: in a stage s we use $p_{i,j}$ to denote the lower (or pessimistic) bound on the game value of the sub-game rooted in stage $s_{i,j}$; $o_{i,j}$ denotes the upper (or optimistic) bound. The restricted game is defined through the sets of actions added to the restricted game – $\mathcal{I}$ represents the set of actions of

Algorithm 1 double-oracle (stage,lower bound,upper bound)

Require: s – current stage; α,β – bounds for the game value
1: **if** s is terminal state **then**
2: **return** $u(s)$
3: **if** alpha-beta$_{Min}(s, minval, maxval)$ = = alpha-beta$_{Max}(s, minval, maxval)$ **then**
4: $u(s) \leftarrow$ alpha-beta$_{Min}(s, minval, maxval)$
5: **return** $u(s)$
6: initialize restricted action sets $\mathcal{I}$ and $\mathcal{J}$ with a first action in stage s
7: $p_{\mathcal{I}\mathcal{J}} \leftarrow$ alpha-beta$_{Min}(s_{\mathcal{I}\mathcal{J}}, minval, maxval)$
8: $o_{\mathcal{I}\mathcal{J}} \leftarrow$ alpha-beta$_{Max}(s_{\mathcal{I}\mathcal{J}}, minval, maxval)$
9: **repeat**
10: **for** $i \in \mathcal{I}, j \in \mathcal{J}$ **do**
11: **if** $p_{i,j} < o_{i,j}$ **then**
12: $u(s_{i,j}) \leftarrow$ double-oracle$(s_{i,j}, p_{i,j}, o_{i,j})$
13: $p_{i,j} \leftarrow u(s_{i,j})$; $o_{i,j} \leftarrow u(s_{i,j})$
14: $\langle u(s), (\mathbf{x},\mathbf{y})\rangle \leftarrow$ ComputeNE$(\mathcal{I}, \mathcal{J})$
15: $\langle i', v_{Max}\rangle \leftarrow \mathrm{BR}_{Max}(s, \alpha, \mathbf{y})$
16: $\langle j', v_{Min}\rangle \leftarrow \mathrm{BR}_{Min}(s, \beta, \mathbf{x})$
17: **if** i' = **null then**
18: **return** minval
19: **else if** j' = **null then**
20: **return** maxval
21: $\alpha \leftarrow \max(\alpha, v_{Min})$; $\beta \leftarrow \min(\beta, v_{Max})$
22: $\mathcal{I} \leftarrow \mathcal{I} \cup \{i'\}$; $\mathcal{J} \leftarrow \mathcal{J} \cup \{j'\}$
23: **until** $\alpha = \beta$
24: **return** $u(s)$

Max player, $\mathcal{J}$ the set of actions of Min player. We denote $\mathbf{x} \in \mathbb{R}^n_{+,0}$ to be the vector representing a mixed strategy of the Max player, $\mathbf{y} \in \mathbb{R}^m_{+,0}$ represent a mixed strategy for the Min player ($\sum_i x_i = \sum_j y_j = 1$). We use *minval* (or *maxval*) to refer to minimal (or maximal) utility value in the game (can be infinite). Methods alpha-beta$_{Max/Min}$ represent classical Alpha-Beta search and the lower index determines which player has the advantage and moves as the second one. The outcome of our algorithm is both the game value as well as the strategy profile that forms the NE in each stage, for which the double-oracle method was called.

3.1 Double-oracle in Stage Games

The pseudocode of the main method is depicted in Algorithm 1. The evaluation of a stage immediately ends if it is a terminal state of the game (line 1), or the bounds calculated by serialized alpha-beta are equal (line 3). Otherwise, the algorithm initializes the restricted game with an arbitrary action (line 6) together with the bounds on this successor (lines 7-8).

In the main loop, the algorithm calculates the exact values for all the successors included in the restricted game (lines 10-13), following by the computation of a NE by solving the linear program (line 14). The result of the computation is the value of the restricted game $u(s)$ and the strategy profile represented as a pair of vectors $(\mathbf{x}, \mathbf{y})$. Next, the best-response algorithms calculate new best-response actions in the current stage for each of the players (lines 15-16). If one of the best-response methods returns a null action, it means that none of

Algorithm 2 BR$_{Max}$(stage, bound, opponent's strategy)

Require: s – current stage; α – lower bound of the game value; $\mathbf{y}$ – strategy of the Min Player
1: $\mathrm{BR}_{value} \leftarrow \alpha$
2: $i_{BR} \leftarrow$ **null**
3: **for** $i = \{1, \ldots, n\}$ **do**
4: $p_{i,\mathcal{J}} \leftarrow$ alpha-beta$_{Min}(s_{i,\mathcal{J}}, minval, maxval)$
5: $o_{i,\mathcal{J}} \leftarrow$ alpha-beta$_{Max}(s_{i,\mathcal{J}}, minval, maxval)$
6: **for** $j \in \mathcal{J}; y_j > 0 \wedge p_{i,j} < o_{i,j}$ **do**
7: $p'_{i,j} \leftarrow \max\left(p_{i,j}, \mathrm{BR}_{value} - \sum_{j' \in \mathcal{J} \setminus \{j\}} y_{j'} \cdot o_{i,j'}\right)$
8: **if** $p'_{i,j} > o_{i,j}$ **then**
9: continue with next i
10: **else**
11: $u(s_{i,j}) \leftarrow$ double-oracle$(s_{i,j}, p_{i,j}, o_{i,j})$
12: $p_{i,j} \leftarrow u(s_{i,j})$; $o_{i,j} \leftarrow u(s_{i,j})$
13: **if** $\sum_{j \in \mathcal{J}} y_j \cdot u(s_{i,j}) \geq \mathrm{BR}_{value}$ **then**
14: $i_{BR} \leftarrow i$; $\mathrm{BR}_{value} \leftarrow \sum_{j \in \mathcal{J}} y_j \cdot u(s_{i,j})$
15: **return** $\langle i_{BR}, BR_{value}\rangle$

the actions in this stage has an expected utility at least as good as the bound (either α or β) that was found in a predecessor of the stage s and a pruning occurs (lines 17-20). If the best-response algorithms return valid best-responses, the bounds on the value of the game in the stage s may be updated – the lower bound may be increased, if the expected utility of the best response of Min player to some strategy $\mathbf{x}$ is higher than the current lower bound α (similarly for the upper bound β). Over the iterations the size of the interval $[\alpha, \beta]$ decreases and the algorithm terminates, when the best response algorithms return the same value; the pessimistic and optimistic values are equal, and the value of the sub-game rooted in stage s is returned (line 23).

3.2 Best-response Action Calculation

The pseudocode of the methods for calculating the best-response actions in a stage s of the game is depicted in Algorithm 2 (for brevity we describe only the variant for the Max player; the variant for Min is similar). The main idea is to select an action with maximal expected utility given the strategy of the opponent using as few recursive calls of the double-oracle method as possible.

The algorithm pessimistically initializes the current best value (line 1) and then evaluates each action of the Max player. Considering an action i of the Max player, the algorithm investigates those successors $s_{i,j}$, such that they can be reached considering the strategy of the opponent (i.e., $j \in \mathcal{J}; y_j > 0$), and the exact value $u(s_{i,j})$ is not known yet (i.e., $p_{i,j} < o_{i,j}$). The algorithm calculates new current pessimistic bound for the game value of the successor $s_{i,j}$ (line 7): $p'_{i,j}$ equals to maximum of either the original bound (i.e., $p_{i,j}$), or the minimal possible value in order for the action i to become the best response given the current strategy of the opponent $\mathbf{y}$ and the current value of the best-response (BR_{value}), if we assume that all other values $u(s_{i,\mathcal{J}\setminus\{j\}})$ are equal to the optimistic values. If the new pessimistic value $p'_{i,j}$ is higher than original optimistic value, the algorithm moves to the next action i (line 9). Otherwise, the exact game value of the succes-

sor is calculated, and if the expected value of action i is higher than the current best response value, the action is stored as the best response (lines 13-14).

4 Theoretical Analysis

In this section, we first prove the correctness of the proposed algorithm and then we formally analyze its performance.

4.1 Correctness of the Algorithm

First, we prove the correctness of the best-response algorithms, starting from the usage of the classical Alpha-Beta search on the serialized variants of the game.

Lemma 4.1. Let M be a matrix game with value $v(M)$; let p be the value of the perfect information game created form the matrix game by letting the maximizing player move first and the minimizing player second with the knowlege of the first player's action selection and o the value of the analogic game in which the minimizing player moves first, then

$$p \leq v(M) \leq o.$$

Proof.

$$v(M) = \max_{x \in \mathbb{R}^n_{+,0}} \min_{y \in \mathbb{R}^m_{+,0}} xMy^T = \max_{x \in \mathbb{R}^n_{+,0}} \min_{y \in \{0,1\}^m} xMy^T \geq$$
$$\geq \max_{x \in \{0,1\}^n} \min_{y \in \{0,1\}^m} xMy^T = p. \quad (1)$$

Assuming always $\sum_i x_i = \sum_j y_j = 1$, the first equality is the definition of the value of a zero-sum game, the second is the fact that a best response can always be found in pure strategies: if there was a mixed strategy best response with expected utility v and some of the actions form its support would have lower expected utility, removing the action from the support would increase the value of the best response, which is a contradiction. The inequality in (1) is a maximization over a subset of the original set, which can give only a lower value. Analogical proof holds for $v \leq o$. □

Corollary 4.2. The serialized alpha-beta$_{Max}$ (alpha-beta$_{Min}$) algorithm computes the upper (lower) bound on the value of the simultaneous moves sub-game defined by stage s.

Proof. Since it is known that the standard Alpha-Beta algorithm returns the same result as the MiniMax algorithm without pruning, we disregard the pruning in this proof. We use the previous Lemma inductively. Let s be the current stage and let M be the exact matrix game representing s. By induction we assume the algorithm computes for stage s some M' so that $\forall i, j\ M'_{i,j} \geq M_{i,j}$. It means that the worst case expected payoff of any fixed strategy in M' is larger than in M; hence, $v(M') \geq v(M)$. The serialization used in alpha-beta$_{Max}$ assumes the Min player moves first; hence, by Lemma 4.1 the algorithm returns value $o \geq v(M') \geq v(M)$. The proof for alpha-beta$_{Min}$ is similar. □

Corollary 4.2 shows that the bounds $p_{i,j}$ and $o_{i,j}$ in the algorithms are correct. Assuming inductively the values $u(s_{i,j})$ from all successors of a stage s are correct, the best-response algorithms always return either a correct best response i_{BR} with maximal expected value given the strategy of the opponent, or **null** in case no action is better than the given bound. Now we can formally prove the correctness of the main algorithm.

Lemma 4.3. Let v be the value of the sub-game defined by stage s then double-oracle(s, α, β) returns

$$v \text{ if } \alpha \leq v \leq \beta;\ \ minval \text{ if } v < \alpha;\ \ \text{and } maxval \text{ if } v > \beta.$$

Proof. The basis for the correctness of this part of the algorithm comes from the standard double-oracle algorithm for zero-sum normal form games [McMahan *et al.*, 2003].

Case 1: If the algorithm ends on line 24 and we assume that all the recursive calls of the algorithm are correct, the algorithm performs the standard double-oracle algorithm on exact values for sub-games. The only difference is that the best-response algorithms are bounded by α or β. However, if the algorithm did not end on lines 18 and 20, these bounds were never restrictive and the best-response algorithm has returned strategies of the same value as it would without any bounds.

Case 2: If the algorithm returns on lines 18 or 20, the situation is symmetric: WLOG we assume the condition on line 17 holds. We have to consider two possibilities. If the bound α has not been updated during the iterations of the main loop (line 21) and the algorithm finds a strategy $\mathbf{y}$ so that no best response to this strategy has a higher expected value than α, the strategy $\mathbf{y}$ proves that the value of the stage game is lower than α and *minval* is returned. If the bound α has been updated on line 21, the algorithm can never stop at line 18. Modifying α means the algorithm has found a strategy $\mathbf{x}$ for *Max* that guarantees the reward of α against any strategy of the opponent. For any strategy $\mathbf{y}$ for *Min* that can occur in further iterations of the main cycle, playing mixed strategy $\mathbf{x}$ gains at least α for *Max*. It means that either all pure strategies from the support of $\mathbf{x}$ gain payoff α, or there is at least one pure strategy that gains even more than α in order to achieve α in expectation. Either way, the BR$_{Max}$ algorithm does not return **null**. □

4.2 Performance Analysis

Two factors affect the performance of the proposed algorithm: (1) ordering of the actions in each stage, and (2) the size of the support of NE in stage games. The effect of action-ordering is similar as in classical Alpha-Beta search – if the best actions are considered first, the remaining actions are more-likely to be pruned. The size of the support (i.e., how many actions are actually used in NE with non-zero probability) represents the minimal number of actions to be added by the best-response algorithms and estimates the number of LPs computed per node. We denote $m = \max_{s \in S} m_s$; $n = \max_{s \in S} n_s$; and d the depth of the game.

The optimal case for the algorithm is under the existence of a pure subgame-perfect Nash equilibrium and the optimal move ordering. In this case the serialized Alpha-Beta algorithms in the root node return the same value for both serializations, since if there is a pure Nash equilibrium in a matrix game, each serialization of the game leads to the value of the Nash equilibrium [Osborne and Rubinstein, 1994, p. 22]. Alpha-Beta pruning effectively reduces the branching factor

to its square root in the best case [Russell and Norvig, 2009, p. 169]; hence, the search evaluates $O\left((mn)^{\frac{d}{2}}\right)$ serialized nodes, using a constant time for evaluating each of them.

In the worst case, the iterative algorithm in all stages of the game constructs the whole game matrix adding one action at a time. As a result, the computation in one node will require evaluating $(m+n)$ linear programs. Furthermore, the worst case estimate is that the serialized alpha-beta search does not prune anything and evaluates all $(mn)^d$ nodes of the serialized trees. If we assume that the results of both Algorithm 1 and 2 are cached and never re-computed with the exact same inputs, and LP_{mn} denotes the computational complexity of solving an $m \times n$ matrix game by linear programming, the worst case time complexity of the whole algorithm is

$$O\left((m+n)LP_{mn}(mn)^{d-1} + 2(mn)^d\right) \subseteq O\left((m+n)LP_{mn}(mn)^{d-1}\right)$$

The worst as well as the best case complexity of the full search algorithm is $O\left(LP_{mn}(mn)^{d-1}\right)$.

5 Experiments

We experimentally evaluate the proposed algorithm on two specific games and a set of randomly generated games. As a baseline algorithm we use the full search based on the backward induction that solves the full linear program (LP) in each stage of the game (denoted FullLP). The same baseline algorithm and one of the games was also used in the most related previous work [Saffidine *et al.*, 2012], which allows the direct comparison of the results, since their pruning algorithm introduced only minor improvement. We use two variants of our algorithm to measure the improvement introduced by each of the two components. Besides the variant of the algorithm described in Section 3, denoted DO$\alpha\beta$, we also run a variant without the serialized Alpha-Beta search (denoted as DO).

None of the algorithms uses any domain-specific knowledge or heuristics and a different random move ordering is selected in each run of the experiments. All the compared algorithms were implemented in Java, using a single thread on a 2.8 GHz CPU and CPLEX v12.4 for solving LPs.

5.1 Experiment Domains

Goofspiel

In the Goofspiel card game (also used in [Saffidine *et al.*, 2012]), there are 3 identical decks of cards with values $\{1, \dots, d\}$ (one for nature and one for each player). The game is played in rounds: at the beginning of each round, nature reveals one card from its deck and both players bid for the card by simultaneously selecting (and removing) a card from their hands. A player that selects a higher card wins the round and receives a number of points equal to the value of the nature's card. In case both players select the card with the same value, the nature's card is discarded. When there are no more cards to be played, the winner of the game is chosen based on the sum of card values he received during the whole game. We follow the assumption made in [Saffidine *et al.*, 2012] that both players know the sequence of the nature's cards. In the experiments we varied the size of the decks of cards, and in each experimental run we used a randomly selected sequence of the nature's cards.

Pursuit-evasion Game

The pursuit-evasion game is played on a graph for a predefined number of moves (d) by an evader and a pursuer that controls 2 pursuing units. The evader wins, if she successfully avoids the units of the pursuer for the whole game; pursuer wins, if her units successfully capture the evader. The evader is captured if either her position is the same as the position of a pursuing unit, or the evader used the same edge as a pursuing unit (in the opposite direction). We used a grid-graph for the experiments (5×5 nodes) and we altered the starting positions of the players (the distance between the pursuers and the evader was always at most $\left\lfloor \frac{2}{3}d \right\rfloor$ moves, in order to provide a possibility for the pursuers to capture the evader).

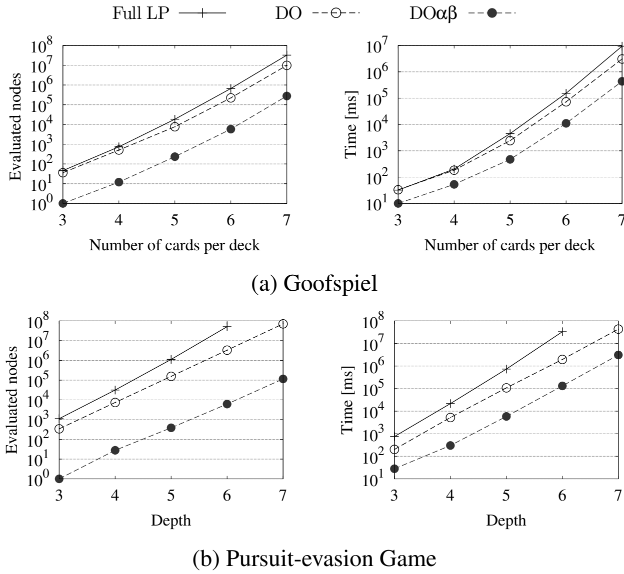
Random Games

In randomly generated games, we fixed the number of actions that players can play in each stage to 4 (the results were similar for different branching factors) and we varied depth of the game tree. We use 4 different methods for randomly assigning the utility values to the terminal states of the game: (1) the utility values are uniformly selected from the interval $[0, 1]$; (2) the utility values are binary, uniformly selected from the set $\{0, 1\}$; (3) we randomly assign either -1 or $+1$ value to each edge (i.e., joint move), and the utility value in a leaf is a sum of all values on edges on the path from the root of the game tree to the leaf; (4) as in the previous case, however, the utility is equal to the signum of the sum of all values on the edges, which corresponds to win/loose/tie games. The first two methods are difficult for pruning, since there is no correlation between actions and utility values in sibling leafs. The two latter methods are based on random *T-games* [Smith and Nau, 1995], that create more realistic games using the intuition of good and bad moves.

5.2 Results

We primarily compare the overall running time for each of the algorithms; secondly, we compare the number of *fully evaluated nodes*, i.e., the stage games on which Algorithm 1 is called. The reported results are averages of several runs of the algorithms – we used at least 50 runs for each of the variant of our algorithm and 10 runs of the FullLP algorithm (the variance of the FullLP algorithm is much less dependent on the specific structure and utility values in the game).

The results for both specific games are depicted in Figure 2 (note the logarithmic y-scale). In both cases the DO$\alpha\beta$ algorithm significantly outperforms the FullLP search. In larger instances of Goofspiel, it fully evaluates less than 1% of nodes in a fraction of time (less than 7%); FullLP solved the largest instance of Goofspiel with 7 cards in over 2 hours and evaluated $\approx 3 \cdot 10^8$ nodes, while DO$\alpha\beta$ solved it in 9 minutes fully evaluating $\approx 2 \cdot 10^6$ nodes. Our results highly contrast with the Goofspiel results of the pruning algorithm presented in [Saffidine *et al.*, 2012], where the number of evaluated nodes was at best around 20%, and the running time improvement was only marginal. Even the DO algorithm without the serialized Alpha-Beta search performs better compared to the FullLP – for the larger instances it fully evaluates only around 30% nodes in 40% time of FullLP.



(a) Goofspiel

(b) Pursuit-evasion Game

Figure 2: Comparison of evaluated nodes and running time for (2a) Goofspiel, and (2b) a pursuit-evasion game.

The improvement is even more apparent in the pursuit-evasion game, where our algorithm can easily exploit the fact that in many positions of the units a pure subgame-perfect equilibrium exists and it is found by the serialized Alpha-Beta searches. The largest instances of the pursuit-evasion games with depth 7 have up to 10^{12} nodes, out of which the DO$\alpha\beta$ algorithm on average fully evaluates only $\approx 1 \cdot 10^5$, and solves the game in 51 minutes, while a single run of the FullLP algorithm has not finished in 10 hours.

As indicated by the worst case computational complexity analysis in Section 4, there are games, on which the proposed algorithm performs worse than FullLP. If we assign completely random values to the leafs of the game tree, every action can lead to an arbitrary good or bad situation; hence, it is very hard to prune some parts of the game tree. When the utility values are real values from the interval $[0, 1]$, both variants of our algorithm were slower than FullLP; although on large instances DO$\alpha\beta$ fully evaluates less than 50% nodes, it takes more than 140% time of the FullLP. The situation already changes when we use binary utility values: the DO$\alpha\beta$ fully evaluates less than 27% nodes taking less than 83% time of the FullLP; DO evaluates around 40% nodes and it is slightly faster than FullLP.

In the more realistic class of random games with good and bad moves, the proposed algorithm again achieves substantial improvements. The results are depicted in Figure 3 with 'sgn' denoting the variant with utilities form the set $\{1, 0, -1\}$. The trends are similar to the experiments on specific games. As expected, less distinct utility values suits the pruning algorithm better. We plot the results for the FullLP only once, because of small differences for both types of utilities.

5.3 Further Improvements

Presented results show great computational savings compared to the full search achieved without any domain-specific knowledge. However, adding the domain-specific knowledge and heuristics known from classical Alpha-Beta search can improve the performance even further. For example, with a domain-dependent move-ordering heuristic, the modified algorithm required only $\approx 60\%$ of the original DO$\alpha\beta$ time in the pursuit-evasion game (the largest instances with depth 7 were on average solved in 30 minutes).

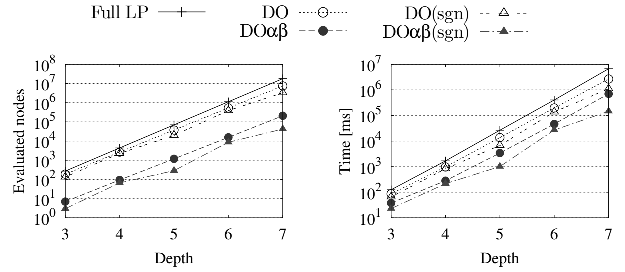


Figure 3: Comparison of evaluated nodes and running time for random games with correlated utility values.

We have also performed initial experiments with more advanced pruning techniques. We modified the best-response algorithm, such that in the computation of the exact value for a stage $s_{i,j}$ (line 11 in Algorithm 2), the algorithm calls the double-oracle method with the tighter bounds $(p'_{i,j}, o_{i,j})$. If the double-oracle returns *minval*, the algorithm can update the upper bound $o_{i,j} \leftarrow p'_{i,j}$ and immediately move to the next action i. On the domain of pursuit-evasion game, this modification slightly improved the performance of the DO$\alpha\beta$ algorithm, but the need for repeated evaluations of the same subtree with different bounds did not pay off in other domains. On the other hand, supporting this kind of pruning in the algorithm allows generalization of techniques such as null-window search for simultaneous-moves games.

6 Conclusions

In this paper we present a novel pruning algorithm for solving two-player zero-sum extensive-form games with simultaneous moves. The algorithm is based on (1) the double-oracle method applied on normal-form games that represent single simultaneous move by both players, and (2) the fast calculation of bounds using the classical Alpha-Beta search run on serialized variants of the original game. Our algorithm shows significant improvement in terms of running time as well as the number of fully evaluated nodes in the game tree without any domain-specific knowledge or heuristics.

The work presented in this paper stimulates a variety of future work. Primarily, the efficiency of the algorithm can be further improved by incorporating existing game-tree search heuristics (e.g., altering the bounds when recursively calling the double-oracle method, using a null-window, or using transposition-tables). Secondly, our algorithm can be easily modified to an approximative version that could be sufficient for many specific domains.

Acknowledgments

This research was supported by the Czech Science Foundation (grant no. P202/12/2054).

References

[Buro, 2003] M. Buro. Solving the oshi-zumo game. *Advances in Computer Games*, 10:361–366, 2003.

[Enzenberger *et al.*, 2010] M. Enzenberger, M. Muller, B. Arneson, and R. Segal. Fuegoan open-source framework for board games and go engine based on monte carlo tree search. *Computational Intelligence and AI in Games, IEEE Transactions on*, 2(4):259–270, 2010.

[Genesereth *et al.*, 2005] M. Genesereth, N. Love, and B. Pell. General game playing: Overview of the aaai competition. *AI magazine*, 26(2):62, 2005.

[Gibson *et al.*, 2012] R. Gibson, N. Burch, M. Lanctot, and D. Szafron. Efficient monte carlo counterfactual regret minimization in games with many player actions. In *Advances in Neural Information Processing Systems 25*, pages 1889–1897, 2012.

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

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

[Kovarsky and Buro, 2005] A. Kovarsky and M. Buro. Heuristic search applied to abstract combat games. *Advances in Artificial Intelligence*, pages 55–77, 2005.

[Littman, 1994] Michael L Littman. Markov games as a framework for multi-agent reinforcement learning. In *Proceedings of the eleventh international conference on machine learning*, pages 157–163, 1994.

[McMahan *et al.*, 2003] H.B. McMahan, G.J. Gordon, and A. Blum. Planning in the presence of cost functions controlled by an adversary. In *International Conference on Machine Learning*, pages 536–543, 2003.

[Nguyen and Thawonmas, 2012] K. Nguyen and R. Thawonmas. Monte-Carlo Tree Search for Collaboration Control of Ghosts in Ms. Pac-Man. *Computational Intelligence and AI in Games, IEEE Transactions on*, PP(99):1, 2012.

[Osborne and Rubinstein, 1994] M.J. Osborne and A. Rubinstein. *A course in game theory*. MIT press, 1994.

[Rhoads and Bartholdi, 2012] Glenn C. Rhoads and Laurent Bartholdi. Computer Solution to the Game of Pure Strategy. *Games*, 3(4):150–156, 2012.

[Russell and Norvig, 2009] S Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3rd edition, 2009.

[Saffidine *et al.*, 2012] A. Saffidine, H. Finnsson, and M. Buro. Alpha-beta pruning for games with simultaneous moves. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence (AAAI)*, pages 22–26, 2012.

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

[Smith and Nau, 1995] S.J.J. Smith and D.S. Nau. An analysis of forward pruning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 1386–1386, 1995.

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

[Teytaud and Flory, 2011] O. Teytaud and S. Flory. Upper confidence trees with short term partial information. In *Applications of Evolutionary Computation*, volume 6624 of *Lecture Notes in Computer Science*, pages 153–162. Springer Berlin Heidelberg, 2011.

[Vanek *et al.*, 2012] O. Vanek, B. Bosansky, M. Jakob, V. Lisy, and M. Pechoucek. Extending Security Games to Defenders with Constrained Mobility. In *In Proceedings of AAAI Spring Symposium GTSSH*, 2012.

[von Stengel, 1996] B. von Stengel. Efficient computation of behavior strategies. *Games and Economic Behavior*, 14:220–246, 1996.