CZECH TECHNICAL UNIVERSITY IN PRAGUE

Parallelization of an algorithm for solving imperfect information games

Martin Svatoš

May 2014

Thesis advisor: Mgr. Viliam Lisý, MSc.

Czech Technical University in Prague
Faculty of Electrical Engineering, Department of Cybernetics

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

BACHELOR PROJECT ASSIGNMENT

Student:	Martin S v a t o š
Study programme:	Open Informatics
Specialisation:	Computer and Information Science
Title of Bachelor Project:	Parallelization of an Algorithm for Solving Imperfect Information Games

Guidelines:

1. Get familiar with the formal model of imperfect information extensive-form games.
2. Understand the sequence-form double oracle algorithm for solving extensive-form games [2].
3. Survey the existing methods for parallelization of tree search algorithms.
4. Analyze the possibilities of parallelization of the algorithm from [2].
5. Implement parallel versions of selected parts of the algorithm in the existing framework.
6. Experimentally evaluate the speed-up achieved by the parallelization.

Bibliography/Sources:

[1] Shoham, Yoav, and Kevin Leyton-Brown: Multiagent systems: Algorithmic, game-theoretic, and logical foundations. Cambridge University Press, 2009.
[2] Bosansky, Branislav, et al.: "Double-oracle algorithm for computing an exact nash equilibrium in zero-sum extensive-form games." Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems. International Foundation for Autonomous Agents and Multiagent Systems, 2013.
[3] Xu, Yan Xu, et al. Alps: A Framework for Implementing Parallel Tree Search Algorithms. The Next Wave in Computing, Optimization, and Decision Technologies Operations Research/Computer Science Interfaces Series, Volume 29, 2005, pp 319-334.
[4] Marsland, T. Anthony, and Fred Popowich: "Parallel game-tree search." Pattern Analysis and Machine Intelligence, IEEE Transactions on 4 (1985): 442-452.

Bachelor Project Supervisor: Mgr. Viliam Lisý, MSc.

Valid until: the end of the summer semester of academic year 2014/2015

L.S.

doc. Dr. Ing. Jan Kybic	prof. Ing. Pavel Ripka, CSc.
Head of Department	**Dean**

Prague, January 10, 2014

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra kybernetiky

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Student:	Martin S v a t o š
Studijní program:	Otevřená informatika (bakalářský)
Obor:	Informatika a počítačové vědy
Název tématu:	Paralelizace algoritmu pro řešení her s neúplnou informací

Pokyny pro vypracování:

1. Seznamte se s formálním modelem her s neúplnou informací v extenzivní formě.
2. Porozumějte algoritmu „sequence-form double oracle“ pro řešení her v extenzivní formě [2].
3. Vytvořte přehled existujících metod pro paralelizaci algoritmů pro prohledávání stromových struktur.
4. Analyzujte možnosti paralelizace algoritmu z [2].
5. Implementujte paralelní verze vybraných částí algoritmu v existujícím softwarovém frameworku.
6. Experimentálně vyhodnoťte zrychlení dosáhnuté paralelizací.

Seznam odborné literatury:

[1] Shoham, Yoav, and Kevin Leyton-Brown: Multiagent systems: Algorithmic, game-theoretic, and logical foundations. Cambridge University Press, 2009.
[2] Bosansky, Branislav, et al.: "Double-oracle algorithm for computing an exact nash equilibrium in zero-sum extensive-form games." Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems. International Foundation for Autonomous Agents and Multiagent Systems, 2013.
[3] Xu, Yan Xu, et al. Alps: A Framework for Implementing Parallel Tree Search Algorithms. The Next Wave in Computing, Optimization, and Decision Technologies Operations Research/Computer Science Interfaces Series, Volume 29, 2005, pp 319-334.
[4] Marsland, T. Anthony, and Fred Popowich: "Parallel game-tree search." Pattern Analysis and Machine Intelligence, IEEE Transactions on 4 (1985): 442-452.

Vedoucí bakalářské práce: Mgr. Viliam Lisý, MSc.

Platnost zadání: do konce letního semestru 2014/2015

L.S.

doc. Dr. Ing. Jan Kybic	prof. Ing. Pavel Ripka, CSc.
vedoucí katedry	**děkan**

V Praze dne 10. 1. 2014

Acknowledgement

I would like to thank my advisor Viliam Lisý for his assitance and guidance during writing this thesis throught impasses of analysis, understanding and progression of solution as much as for introduction to game theory and approach to academic works.

This work would not be finished without my parents, Jana and Michael, my family and friends and colleagues, who supported me during my student's years. I would like to thank all of them.

Declaration

I declare that I worked out the presented thesis independently and I quoted all used sources of information in accord with Methodical instructions about ethical principles for writing academic thesis.

In Prague May 21, 2014 ....

Poděkování

Chtěl bych poděkovat svému vedoucímu Viliamu Lisému za jeho vedení a pomoc ve slepých uličkách analýzy, pochopení a vývoje řešení stejně tak, jako za zasvěcení do teorie her a přístupu ke zpracování akademické práce.

Tato práce by ovšem nevznikla ani bez mých rodičů, Jany a Michaela, mé rodiny a přátel a kolegů, kteří mě podporovali během celé doby studia, za což jim všem patří mé srdečné díky.

Prohlášení

Abstract

Game theory is science not only for modeling games such as Chess, Poker or Backgammon compute its solutions but it can also provide useful tools for modeling and solving real life problems that we are not aware of at the first sight. For example, one problem from this category is borders patrolling . These problems are linked because each of them includes huge state space which has to be searched in order to find an optimum solution that maximizes possible payoff under the assumption that opponent acts rationally.

The aim of this work is to design and experimentally evaluate the performance of a parallel version of the algorithm from framework developed by ATG group from FEE, CTU. This algorithm uses *double-oracle* framework, which narrows searched state space by allowing only some strategies to be played by players. The algorithm solves two-player zero-sum games with imperfect information. Before proposing a parallel design an analysis of the algorithm and a survey of known parallel search techniques on two-players game trees are investigated. Parallel version should contribute by shorter computation time allowing faster computation of *exploitability* of heuristic strategies in imperfect-information games. Simplified, exploitability expresses distance of a strategy from optimum. When comparing two strategies, the one with lower exploitability is better.

Experiments show that parallel design proposed in this work achieved speed up of 2.2.

Keywords

parallelization, extensive form game, Nash equilibrium

Abstrakt

Teorie her je obor zabývající se nejen modelováním her jako jsou šachy, poker nebo vrhcáby a jejich řešení, ale i modelováním a řešení realných problému, kterých si nemusíme na první pohled být vědomi. Jedním z těchto realných problému je například ochrana a patrolování hranic. Tyto problémy spojuje velký stavový prostor, který musí být prohledán pro nalezení optimálního řešení, které vede k největšímu možnému zisku za předpokladu, že protihráč hraje racionálně.

Tato práce si klade za cíl navrhnout a experimentálně ověřit výkonnost paralelní verze algoritmu z frameworku vyvinutého ve skupině ATG z FEL, ČVUT. Tento algoritmus využívá *double-oracle* framework, který zmenšuje prohledávaný stavový prostor omezením strategií, které můžou hráči používat. Jedná se o dvou hráčové hry s nulovým součtem a neúplnou informací. Před provedeném samotného návrhu je nejdříve analyzován algoritmus a poté je proveden průzkům paralizačních technik prohledávání dvouhráčových hracích stromů. Paralelní verze by měla přispět k rychlejším výpočtům, což umožňuje rychlejší vypočítávání *exploitability* heuristických strategií pro hry s neúplnou informací. Zjednodušeně řečeno, exploitability vyjadřuje vzálenost strategie od optima. Čím menší exploitability je, tím je daná strategie lepší.

Experimenty provedné na navržené paralelizaci, která je popsána v této práci, daného algoritmu dosáhly zrychlení 2.2.

Klíčová slova

paralelizace, hry v extenzivní formě, Nashovo equilibrium

Contents

Abbreviations

AB	Alpha-Beta
BRS	Best-Response Sequnce
BRSA	Best-Response Sequence Algorithm
LP	Linear programming
MWF	Mandatory-Work-First
NE	Nash equilibrium
PVS	Principle Variation Splitting
PVSA	Principle Variation Splitting Algorithm
YBWC	Young Brother Wait Concept

1 Introduction

Game theory is a useful mathematical tool for modeling and solving well known games such as Chess, Tic-Tac-Toe and Go but also for Poker or Phatnom Tic-Tac-Toe. The cardinal difference separating these two categories is what information each player has. The second category represents an imperfect-information game; those are games where a player is not ensured to know all information about the game state. For example, a player does not know what cards his opponent has in Poker or which coordinates the opponent has taken by his mark in Phantom Tic-Tac-Toe. Security games can be also formulated by game theory and this kind of games is relevant to humankind safety and real life problems [3] because these models can be used in real life situations to minimalize potential danger or loss.

1.1 Formulation of problem

Extensive form is a powerful framework since it can be used for modeling imperfect-information games and also security games or stochastic ones. However, even using extensive form the size of a game tree representing a game grows in most cases exponentially according to the number of choices that are available in a game state. In [11] authors have shown that two-player Texas hold'em has 9.17×10^{17} game states. Traversal of this kind of the tree is infeasible even on modern computers.

Therefore approaches solving large-scale problems have been investigated. Some of these approaches are focused on narrowing searched space. One of them is *double-oracle* framework whose high level and simplified idea is based on finding the best possible strategy against fixed opponent strategy in each iteration. If there is no better strategy to be added to player's strategies the algorithm ends; otherwise the player is allowed to play the best found strategy and another iteration continues. The player having a set of allowed strategies has restricted possibilites of actions. On the other hand, only maximizing strategies may propagate to a player allowed strategy set. Imagine a strategy space for above mentioned Texas hold'em. It is easy to see that a set of good strategies is smaller than full strategy space. By *good strategies* are meant those strategies that have propagated to a player strategy set in some iteration. Therefore computation with a set of good strategies is less computationally demanding than computation of all possible strategies against all opponent's strategies.

1.2 Motivation

Parallelization of the algorithm in [2] allow us to compute results faster. Despite that there still will be large instances of games for which it will be infeasible to compute even by the parallel algorithm. There are opportunities to use a heuristic search in this kind of large instances, so the given instance would be feasible. Monte-Carlo tree search is an example of a heuristic search.

Regardless that a parallel version of the algorithm will not be able to compute some large instances of some game, it is possible to use the parallel version to determine *exploitability* of some heuristic search in a given game state. Exploitability is a number greater or equal to zero representing distance from a Nash Equilibrium in a game. When comparing two heuristic strategies, the one with smaller exploitability is better. An incomplete search through a game tree can be seen as a heuristic search depending on the used utility function in non-terminal nodes. By an incomplete search such a search is meant that it does not evaluete full depth of a game tree but only a predefined depth is traversed.

2 Background and definitions

In this chapter, basic knowledge of game theory is defined and described to understand the algorithm later described in Chapter 3. Game theory is a study of conflict and cooperation of players. Mathematical models are used to represent conflict, cooperation and players who make desicions. Game theory is used for solving games as Poker, Phantom Tic-Tac-Toe and Bridge but it is also used in real life situations as economics, political science and biology [6, 25].

2.1 Game examples

Firstly, let's describe a few games from the category that the algorithm described in Chapter 3 solves. This category is called *impefect-information* games and better formulation of it is described later in this chapter. For simplicity, it difers from well known games such as Chess or Go by the information each palyer disposes. Comparing imperfect-inforamtion Phantom Tic-Tac-Toe to Tic-Tac-Toe, in the second one each player knows in which *game state* he is located, despite the first one where player knows only some information about how *game state* looks like.

A *game state* can be informally described as a tuple of player knowledge and situation of playing board. For example in Phantom Tic-Tac-Toe, a game state cannot be figured out by just looking on playing board without having history of one player's actions, since one could not figure out the order of moves leading to this deployment of playing board.

We will refer to a player as *he* throught this work.

2.1.1 Phantom Tic-Tac-Toe

Playing board is a 3×3 board on which two players are trying to put three marks in vertical, horizontal or diagonal row to win the game. One of the players uses a circle meanwhile the other uses a cross to mark each one one's taken positions on the board. Both players remember what action they played but are unable to see opponent's moves unless one of the players tries to put his mark on a position already taken by the opponent; in this case the player knows that the position is taken by the opponent and this information is added to the playing player's knowledge. Players alternate while playing, which is necesserary for any corruption of playing board, for example by putting two different marks on the same position in the same time.

2.1.2 Poker Games

Poker is well known card game and in this thesis we will focuse on two-player poker. Both players start with the same amount of chips and have to put some chips in the pot. Then, Nature player gives to each player a card, so the opponent cannot see the card. A player can quit the game and lose by folding or let the opponent play his move by checking or give some amout of money by betting, calling or raising. The second player can do the same. Thereafter, if no player folds, Nature player gives one card on the table. A player wins if his card matches the one on table and the opponent's does not match, or the card on table does not mach to any card of both players and the player disposes higher card than his opponent.

2.1.3 Search Games

The game consists of two players and a directed graph on which one tries to safely cross from the starting point to the destination point, the other player wants to capture the first one. The crossing player moves leave tracks on the graph but the player can use slow moves which erase these tracks. The other player is allowed to move only between some nodes of the graph. If the crossing player does not make his path to the destination point within a predefined number of steps and is not captured along his path, a draw occurs. Otherwise crossing player wins if he manages to get to the finishing point or loses if he is captured by the opponent.

2.2 Two-player zero-sum finite games with perfect-recall

Game theory can be used for formulations of many situations with multiple players. In this thesis we will considere only two-player zero-sum games. It is a branch of games where two players are playing. We will use N to refer set of players. So in case of two-player game $N = \{I, II\}$ where I and II refer to *Player I* and *Player II* respectively. Three possibilities can occure at the end of a game. Firstly, Player I wins and Player II loses. Secondly, Player I loses and Player II wins. Finally, a draw can occure meaning that no player wins or loses.

Each player can choose a strategy to play. According to strategies that both of players selected, the *end state* is reached. The end state is a state in which the game is terminated, so it is also sometimes called *terminal state*. In the end state, each player gains a payoff. In zero-sum games, sum of all palyer's payoffs is equal to zero; in other words, since describing two-player game, one player gains the same value as the second loses.

Later in this text, by a game a *finite game* is meant. A finite game is a game where all players have a finite set of strategies. The set of strategies is described later.

Also only games with *perfect-recall* are described in this work. In this type of games a player remembers all information that he has gained during playing. Let's demonstrate this on Phantom Tic-Tac-Toce - once a player has taken a position by putting his mark on the position, he knows that the position is taken by his mark to the end of the game and he does not forget this information.

2.3 Strategies

A strategy for a game is a complete description of actions for a player to perform in every possible state of the game that can occure during the game. S_i is a set of strategies of player i. There are *pure* and *mixed* strategies.

Player using a mixed strategy is focused on his average payoff, since the outcome is an average of some pure strategies. More precisely, mixed strategy consists of a set of pure strategies weighted by a probability distrubution.

2.4 Normal form

The normal form game is a triplet (S_I, S_{II}, A) cosisting of nonempty sets of strategies S_i for all $i \in N$ and a payoff function $A : (s_I, s_{II}) \to \mathbb{R}$ where $s_I \in S_I$, $s_{II} \in S_{II}$. Payoff function A represents payoff of Player I; payoff of Player II is the same but multiplied by -1 since the sum of both players payoff is equal to zero. This is a simple formalization of a game, yet still able to express games such as Poker and Rock-Paper-Scissors.

Let's make an example on Rock-Paper-Scissors game which payoff fucntion for Player I is in Table 1. A player can play rock, paper or scissors. After each player select his option, both of

Table 1 A payoff matrix for a two-player zero-sum game for Player I

		Player II		
		rock	paper	scissors
Player I	rock	0	1	-1
	paper	-1	0	1
	scissors	1	-1	0

them reveal selected options in the same time and one can win or lose or a draw occures. Rules are that the *paper* strategy beats the *rock* one, the *rock* one beats *scissors* one and the *scissors* strategy beats the *rock*. In Table 1 rows represent actions of Player I and columns represetn actions of Player II. For example, if both players play rock, profit for Player I is 0 because table cell on rock-rock. If Player I plays rock and Player II paper, Player I wins and gains 1, as seen in the second cell from left in the first row of the table. If Player I plays rock and Player II plays scissors, Player II loses and gains -1, as seen in the third cell from left in the first row of the table.

2.5 Method of solving

Knowing what a player can do and what strategy and normal form are, we can define a *rational player*. Rational player is such a player that is playing in a way to get the maximal possible outcome of a game. Let both players act rationaly. Player I is maximizing his payoff meanwhile Player II is minimizing payoff of Player I. In details, Player II is maximazing his payoff and since the game is zero-sum, it means that payoff of Player I is minimized.

Minimax theorem [6] tell us that for every two-player zero-sum finite game there is a value V and there is a mixed strategy for Player I that will gain at least V on average independently of the strategy of Player II and there is a mixed strategy of Player II that will lose at most V on average independtly of the strategy of Player I. Player I wins if V is greater then 0, loses if V is less then 0 and a draw occures if V is equal to 0. Later in the extensive form, we will see a minimax tree and a minimax algorithm that finds the maximal possible gain for a selected player in a game. This theorem is mentioned here for reasons of the *minimax algorithm* and the *minimax tree* since it corresponds to solving *perfect-information* games. Some algorithms in Chapter 4 are constructed to solve perfect-information games. Perfect-information games is a category of games in which each player knows all information of a game state he is currently situated; a player can use information of his opponent against him, because he knows them as well.

Nash equilibrium [6] can be also used for solving the type of games we are examining. In a Nash equilibrium each player's strategy is the *best response* to all opponent's strategies. The *best response* is described later in this chapter.

As shown in [6] Nash equilibrium can be computed by linear programming. But as we will see later, computing a Nash equilibrium of normal-form game is memory and CPU demanding and that another form can be used for computation of a Nash equilibrium by linear programming.

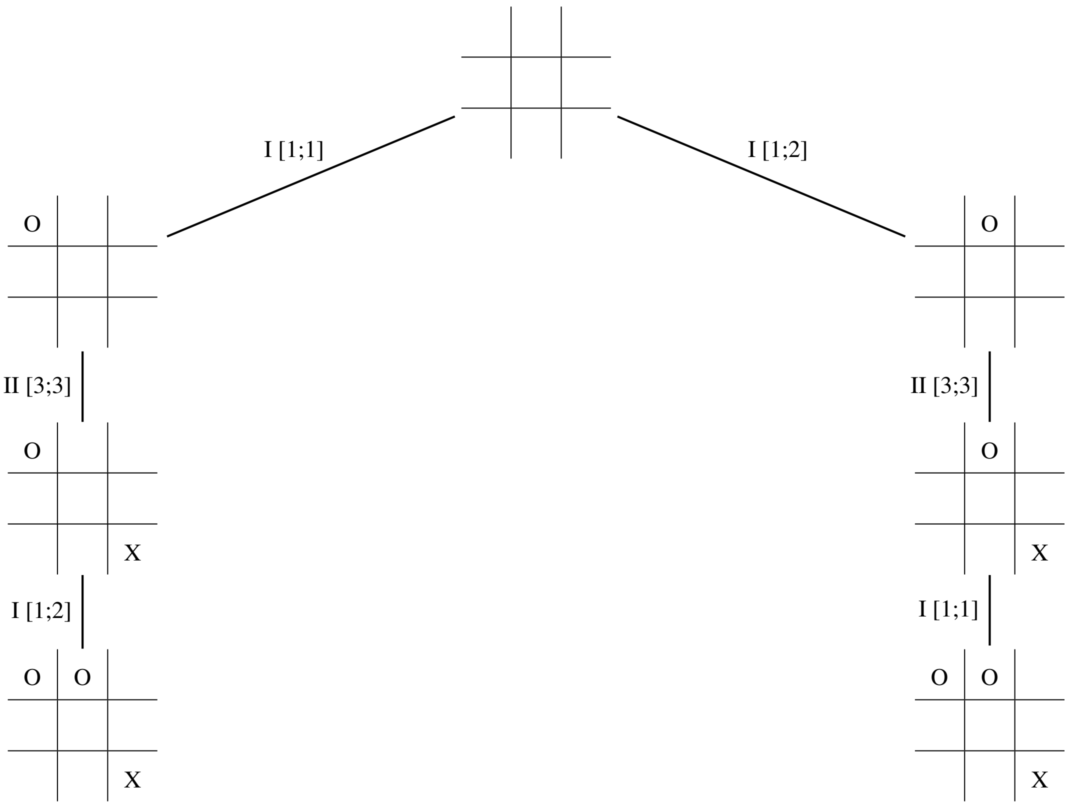

Figure 1 Example of a part of a game tree of Phantom Tic-Tac-Toe with players actions labeling edges and nodes representing playing board.

2.6 Extensive form

Extensive form is another concept for modeling games. In comparsion to normal form, it allows to add elements of chance that are represented by Nature player.

Firstly, we define a *game tree*. The Game tree is a tuple (H, A) where H is a finite set of nodes representing game states and A is a finite set of edges. A chance can be expresed in extensive form as a third player called Nature, yet the game is still two-player. Nodes can be divided into three separated groups according to thier owners; each node belongs to exactly one of these players - Player I, Player II and Nature.

Each leaf of the game tree is a terminal node. We use $Z \subseteq H$ as a set of terminal nodes. For each terminal node and a player i an utility funcion is defined, $u_i : Z \to \mathbb{R}$. The utility functions represent a payoff for player i.

From each non-terminal node h edges lead to others nodes. Each edge represents an action performed by a player to whom h belongs. We say that action $a \in A$ leads from node $h \in H$ to a *succesor / child ha*. Some actions of both players can have several combinations and since a playing board can look the same, it is not the game state. This is caused by the fact that a game state consists of a situation on playing board and information of a player. We can see this in Figure 1. In the figure, playing board on both terminal nodes looks the same but it is not the same game state since in the left one Player I knows *history* $[[1;1],[1;2]]$ meanwhile in the second he knows *history* $[[1;2],[1;1]]$. By history is meant the ordered list of actions that the palyer played.

A rational player will maximize his payoff. Let's ilustrate this on Figure 2. Player II is maximizing his payoff, so in node B he chooses action a and gains -4, in node F he chooses action

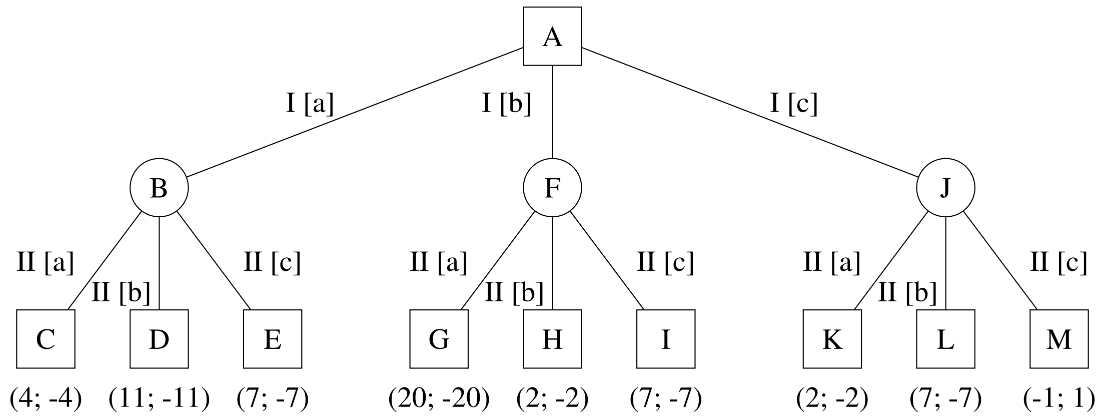

Figure 2 A minimax tree, circle nodes belong to Player II, rectangle nodes to Player I, in terminal nodes payoff is described in tuple (*payoff of Player I*,*payoff of Player II*). Graph edges are labeled by actions of players.

b and gains -2, finally in node *J* he chooses action *c* and gains 1. Since player I is maximizing but also acting as player II behaves rationaly, he chooses from node *A* action *a* leading to node *B*, because he gains the best possible payoff, in this case 4, under the assumption that both players act rationally. In a zero-sum game while II is maximazing he is also minimizing I's payoff. So we can see this as a *minimax tree* which means that in each two layers of the tree one player is maximizing and the second is minimizing.

A minimax algorithm is a basic approach to solve a extensive-form game. The minimax algorithm works on a minimax tree in the way we described above. In case of Nature player represented by a chance node *h*, payoff from this node consists of the maximal possible payoff of its succesors weighted by probability distribution of actions from node *h* leading to each succesor. Later in section 4.1 variants of minimax algorithm with some ehnacements are described because their analysis can be helpfull while designing an algorithm searching imperfect-information game tree.

2.7 Game with imperfect information

From the previous section we know what a game tree and a game state are. Now we define a new term *information set*.

In each node a player knows all moves he has performed to reach this node. A player is also aware of an order in which his actions were performed. Opposite to this, a player has a limited knowledge about opponent's actions. An infromation set is a set of nodes in which the player knows the same information. The game with imperfect information is such a game that its game tree has at least one information set having more than a single node.

An example of a game tree with information sets is in Figure 3. Actions of players label edges of the tree. In the root Player I put his mark on coordinates $[1;1]$ and $[1;3]$ and these actions lead to different nodes. But Player II is unable to distinguish between these two nodes, because it is one game state for him since he does not know what Player I has palyed. So Player II has to consider that he is situated in both of these two nodes. A palyer can know some information about opponent's action; for example in Phantom Tic-Tac-Toe by putting a mark on already taken position, as mentioned before.

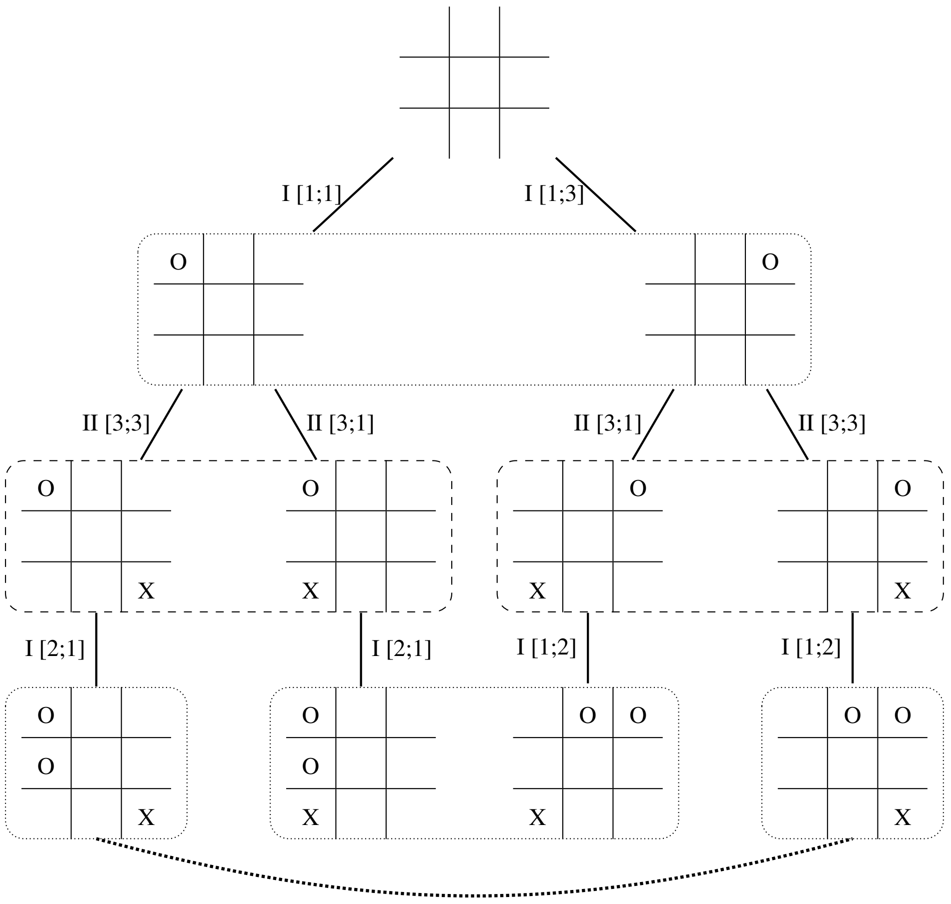

Figure 3 Example of a part of Phantom Tic-Tac-Toe game tree with highlighted information sets. Player's actions label edges. Information sets of Player I are are dashed; information sets of Player II are densely dotted. The densely dotted curve connecting two leaves represents an information set by connecting these two nodes.

2.8 Sequence form

We can transfer an extensive-form game to normal form. Since we know the method for sloving normal-form game, for example linear programming, we can compute Nash equilibrium by transfering game from extensive to normal form. This way is quite expensive since the normal form is exponential in the size of extensive form. This is due to all combinations of information set actions for each player and payoff have to be considered [25]. To compute a Nash equilibrium on exponentially larger strategy space also deals larger memory demands.

The sequence form is based on representing all paths from the root of a game tree using sequences. A sequence can be imagined as a list containing history of player's actions. Actions represent edges in a game tree, since that by applying a suequence to a node, one can go throught a part of a tree. This representation allows to use it for linear programming computation of a Nash equilibrium as well as normal form do [24]. In addition, memory demands of sequence form is linear to the size of extensive form and therefore more suitable for computation of large instances, opposite to exponential size of normal form in comparison to the size of extensive form. The sequence form is also suitable for imperfect-information games [25].

2.9 Realization plan

A realization plan expresses the probability of a sequence δ_i that i is playing assuming that opponent will play actions reaching the same information set. The function $r_i : \Sigma_i \rightarrow \mathbb{R}$ represents i's realization plans where Σ_i is set of all possible sequences in a game for player i and $\delta_i \in \Sigma_i$.

2.10 Best response

Now let's define the best response. Let δ_{-i} be opponent's strategy then δ_i^{BR} is a best response for player i if (1) holds.

$$u_i(\delta_i^{BR}, \delta_{-i}) \geq u_i(\delta_i^{'}, \delta_{-i}) \qquad \forall \delta_i^{'} \in \Delta_i \tag{1}$$

In other words, i's best response gains the best possible payoff for player i against opponent's fixed strategy under the assumption that both players acts rationally.

2.11 Restricted game

Restricted game is a game that arises from the original game by restricting some actions of one or both of players.

3 Double-oracle sequence-form algorithm

The aim of this thesis is to parallelize an algorithm from [2], therefore the algorithm is presented in this chapter.

3.1 Double-oracle algorithm

Finding an exact Nash equilibrium for a game with a large space of strategies is a computationally demanding task. The idea of oracle algorithms is to solve large-scale optimization problems. The principle of double-oracle is to search only a fraction of strategy space. Double-oracle is an iterative algorithm for solving two-players games. Pseudocode of the main idea of double-oracle algorithm sovling normal-form game is in Algorithm 1.

Before describing the algorithm, let's remember and define used notation. $N = I, II$ is the finite set of player. We use i to refer to a player from N, $i \in N$, and $-i$ to his opponent. Σ_i is a set of all possible sequences in a game for player i. We use NE_i to refer i's Nash equilibrium. *BRS* is an abbreviation for Best-response sequence algorithm that si described later. *LP* is an abbreviation for linear programming

```
Data: a two-player zero-sum game
Result: an exact Nash equilibrium and best stretegies for both players
1  $\Sigma_A$ = { arbitrary strategies }
2  $\Sigma_B$ = { arbitrary strategies }
3  while true do
4      ($NE_a$, $NE_b$) ← compute Nash equilibrium by LP for $\Sigma_A$ and $\Sigma_B$
5      $\sigma_b = BRS(NE_a)$
6      $\sigma_a = BRS(NE_b)$
7      if $\sigma_a \in \Sigma_A$ and $\sigma_b \in \Sigma_B$ then
8          break
9      end
10     $\Sigma_A = \Sigma_A \cup \sigma_a$
11     $\Sigma_B = \Sigma_B \cup \sigma_b$
12 end
```

Algorithm 1: Double-oracle pseudocode for normal-form game

First of all, it creates a new restricted game by assigning arbitrary strategies to both players (lines 1, 2). Then in iterations three main steps are executed.

1. Nash equilibrium for each player in the restricted game (line 4) is computed.
2. Best-response strategy for both players (lines 5 - 6) is found.
3. If each of calculated best-responses are already in appropriate startegies set, the algorithm terminates (lines 7 - 9), because there is no better best-response. Otherwise, each of calculated best-responses is added to the relevant player's strategies set (lines 10 - 11). In other words, restricted game is expanded by those strategies, and the algorithm continues in another iteration. The set of strategies is finite, therefore the algorithm stops after finite number of iterations.

In the worst case scenario, the entire strategy space is searched but usually only a fraction of the space is used during the search [3].

3.2 Double-oracle sequence-form algorithm

The algorithm from [2], on which this work is mainly focused, uses a sequence-form and double-oracle framework to solve two-player zero-sum games in extensive-form with imperfect information. By *Double-oracle sequence algorithm* we will refer to this algorithm. This algorithm uses *Best-response sequence algorithm* which is described in 3.4 and we will refer to it also with the capital starting letter.

Both sequence and normal forms can be used for computation an exact Nash equilibrium by LP. Sequence-form is linear in size of the extensive-form game in contrast to exponential size of normal-form of the same game. LP computation of a sequence-form representation is less memory demanding than normal-form because of the size of both representation. This allows us to compute larger instances of games with the same memory load.

Double-oracle using sequence-form representation is constructed by the same idea as the double-oracle algorithm for normal-form game but there are also differences. The main difference is in adding the best found strategies. Adding best-response sequences to the restricted game can cause inconsistencies and therefore an incorrect solution can occure. This is described in [2] in more details.

The algorithm uses a realization plan for representing mixed strategies of both players. A realization plan expresses the probability of a sequence δ_i that searching player is playing assuming that opponent will play appropriate actions leading to the same information set. An information set can be reached by sequentially applying actions of the first and then second player repeatedly. The function $r_i : \Sigma_i \rightarrow \mathbb{R}$ expresses player's i realization plans. With this formulation we can construct following LP that computes the equilibrium of realization plans.

$$\min v_{I_i(\emptyset)} \tag{2}$$

$$v_{I_i(\sigma_i)} - \sum_{I_i^k \in I_i : \mathrm{seq}_i(I_i^k) = \sigma_i} v_{I_i^k} = \sum_{\sigma_{-i} \in \sum_{-1}} \sum_{h \in \omega(\sigma_i, \sigma_{-i})} g_i(h) * r_{-i}(\sigma_{-i}) \qquad \forall \sigma_i \in \Sigma_i \tag{3}$$

$$r_{-i}(\emptyset) = 1 \tag{4}$$

$$\sum_{\forall a \in A(I_{-i}^k)} r_{-i}(\sigma_{-i} a) = r_{-i}(\sigma_{-i}) \qquad \forall \sigma_{-i} = \mathrm{seq}_i(I_{-i}^k), I_{-i}^k \in I_{-i} \tag{5}$$

$$r_{-i}(\sigma_{-i}) \geq 0 \qquad \forall \sigma_{-i} \in \Sigma_{-i} \tag{6}$$

The equation (3) provides that expected values in each information set are maximized by using the best possible action. Equations (4) to (6) represents player $-i$'s realization plan to minimize expected values.

3.3 Best-response algorithm

Best-response algorithm searches the best possible strategy against a fixed opponent's strategy and returns it.

3.4 Best-response sequence algorithm

Since now, we are describing Best-response sequence alogrithm that is proposed in [3], because this work is focused on its parallelization. Also we use an abbreviation *BRS* and *BRSA* for this algorithm. BRS is a version of the best-response algorithm adapted to a sequence-form game. The algorithm executes a depth-first search throught an entire game three.

The output of this algorithm is the best possible strategy for searching player and its expected value against the extended strategy of the opponent. An extended strategy of the opponent is an opponent's strategy from a restricted game extended by default strategy. Default strategy of a player is predefined on the start of the algorithm. Nodes of the opponent and Nature are examined in the same manner but differently from execution of searching player's nodes, therefore an analysis of both cases is described separately.

The idea of the algorithm is to traverse the complete game tree using depth-first search to find the best-response against opponent's fixed strategy given by realization plan $\overline{r}'_i$. Aslo a prunning technique is used to cut-off brachnes that definetely does not participate in the best response.

Before investigating of the algorithm let's notice that we do know the upper bound of the game. Since it is a zero-sum two-player game we also know the lower bound. We refer to these as *MaxUtility* and *MinUtility*. These values are used for cutt-offs of the algorithm shown in following sections.

In following subsections notation is used as in [2, 3]. $A(h)$ denotes a set of actions that can be applied in node h; $ha = h' \in H$ means that node h' is reachable from node h by use of action $a \in A(h)$. Realization plan in the restricted game is dentoed by $r'_i : \Sigma'_i \to \mathbb{R}$ for a sequence of player i. seq_i denotes sequence(s) of actions of player i leading to a node / a set of nodes / an information set. An information set of the player i is denoted by I_i. The realization plan for player i extended to the complete game is denoted by $\overline{r}'_i : \Sigma'_i \to \mathbb{R}$. $C : H \to \mathbb{R}$ denotes the probability of reaching a node w.r.t. Nature play. Finally, we extend utility function to be zero in non-terminal nodes - $g_i : H \to \mathbb{R}$.

$$g_i(h) = \begin{cases} u_i(h) * C(h), & \text{if } h \in Z \\ 0, & \text{if } h \notin Z \end{cases} \tag{7}$$

3.4.1 Nodes of opponent and Nature

In opponent's and Nature's node the algorithm traverses a game tree in depth-first strategy manner and an opponent is playing by his default strategy or strategy given by his realization plan r_{-i} computed by LP in information sets which are in the restricted game. This case is in Algorithm 2.

The algorithm works in following way. It is supposed to searched a node h that belongs to the opponent or Nature.

Firstly, if h is a terminal node, a value of the utility function of playing player for this node is weighted by a propabability that the opponent will play actions leading to h and by the Nature probability of h is returned (lines 1 - 3).

Otherwise algorithm sorts action in descending order (lines 4 - 5) and traverses through succesors of node h (lines 7 - 16). Finally it returns a sum of values given as a result of examination of succesors by Best-response sequence algorithm (lines 11, 17).

```
Data: $i$ - playing player, $h$ - current node, $\overline{r}'_{-i}$ - opponent's strategy, Min/MaxUtility -
        bounds on utility values, $\lambda$ - lower bound for a node $h$
Result: expected value of strategy against the extended strategy of the opponent
1  if $h \in Z$ then
2      return $u_i(h) * \overline{r}'_{-i}(seq_{-i}(h)) * C(h)$
3  end
4  $w \leftarrow \sum_{a \in A(h)} \overline{r}_{-i}(seq_{-i}(ha)) * C(ha)$
5  sort $A(h)$ according to probability
6  $v^h \leftarrow 0$
7  for $a \in A(h)$ do
8      $w_a \leftarrow \overline{r}'_{-i}(seq_{-i}(ha)) * C(ha)$
9      $\lambda' \leftarrow \lambda - [v^h + (w - w_a) *$MaxUtility$]$
10     if $\lambda' \leq w_a * MaxUtility$ then
11         $v^h \leftarrow v^h +$ BRS$_i(ha, \lambda')$
12         $w \leftarrow w - w_a$
13     else
14         return MinUtility $*w$
15     end
16 end
17 return $v^h$
```

Algorithm 2: BRS$_i$ in nodes of the opponent, taken from [3]

Cut-off technique

In previous description of Algorithm 2 cut-off was ommited. The cut-off in this algorithm acts in following way.

A lower bound, represented by λ, is an input parameter as well as a node h. This lower bound expresses a minimal value for the returned value in order to get node h to the best-response. More presice, if value of h is higher than λ this node can parcipate in best-response that is currently computed because parent of h can particpate in best-response etc.; but it does not ensure that h participates in best-response definitely. The only thing that we do know that there is node h does not participate in best-response if his value cannot exeed λ. A new bound λ' for each action is calculated (line 9) under the assumption that all other actions yield their maximal values. If λ' exceeds the maximal possible value of a succesor, this h cannot be a part of best-response and therefore the search terminates (line 14). Otherwise another succesor of h is examined.

3.4.2 Nodes of searching player

Algorithm 3 searches a node h of a playing player. To be precise, since 3.2 solves imperfect-information games, Best-response sequence algorithm searches all nodes of an information set that node h belongs to.

Firstly, if h is a terminal state, its utility function weighted by opponent's and Nature's probabalities is returned as in previous case. If h is already computed, a technique similar to *trans-pozition table* is used to return a stored result from the previous examination. The reason for this is that a requirement to compute an information set can be sumbited repeadly.

Secondly, it takes all nodes from h's information set and sorts them in order decreasing according to a probability of the opponent and Nature (lines 4 - 5).

Thirdly, each action of each node from h's information set is searched (lines 10 - 25).

Finally, to each node of the searched information set a value of the best action leading from this information set is assigned and the same value is returned (lines 26 - 27).

```
Data: $i$ - playing player, $h$ - current node, $\overline{r}'_{-i}$ - opponent's strategy, Min/MaxUtility -
    bounds on utility values
Result: expected value of strategy against the extended strategy of the opponent
1  if $h \in Z$ then
2      return $u_i(h) * \overline{r}'_{-i}(seq_{-i}(h)) * C(h)$
3  end
4  $H' \leftarrow \{h'; h' \in I_i^k\}$
5  sort $H'$ descending according to value $\overline{r}'_{-i}(seq_{-i}(h)) * C(h)$
6  $w \leftarrow \sum_{h' \in H'} \overline{r}'_{-i}(seq_{-i}(h)) * C(h)$
7  $maxVal \leftarrow -\infty$
8  $v_a \leftarrow 0 \ \forall a \in A(h')$
9  $w \leftarrow \sum_{a \in A(h)} \overline{r}_{-i}(seq_{-i}(ha)) * C(ha)$
10 for $h' \in H'$ do
11     $w_{h'} \leftarrow \overline{r}'_{-i}(seq_{-i}(h')) * C(h')$
12     for $a \in A(h')$ do
13         if $maxAction\ is\ empty$ then
14             $\lambda \leftarrow w_{h'} *$ MinUtility
15         else
16             $\lambda \leftarrow (v_{maxAction} + w *$ MinUtility $) - [v_a + (w - w_{h'}) *$ MaxUtility $]$
17         end
18         if $\lambda \leq w_{h'} * MaxUtility$ then
19             $v_a^{h'} \leftarrow \mathrm{BRS}_i(h'a, \lambda)$
20             $v_a \leftarrow v_a + v_a^{h'}$
21         end
22     end
23     $maxAction \leftarrow \mathrm{argmax}_{a \in A(h')} v_a$
24     $w \leftarrow w - w_{h'}$
25 end
26 store $v_{maxAction}^{h'} \ \forall h' \in H'$
27 return $v_{maxAction}^{h}$
```

Algorithm 3: BRS$_i$ in a node of the playing player, taken from [3]

Cut-off technique

Now we describe prunning technique that is used in Best-response sequence algorithm in nodes of a playing player.

In each iteration a new lower bound λ is calculated for an action (lines 13 - 17). This bound is used for refusing a succesor that will not get to the best-response because it's best possible value is lower than the bound under the assumption that this action will yield maximal possible values from other nodes of the currently examined information set.

In case that an action a from a node h' from the currently examined information set has got a possibility to get to the best-response, BRS$_i(h'a, \lambda)$ is invoked and the lower bound is forwarded for further prunning that is decribed in section 3.4.1.

4 Survey of parallelization of tree search algorithms

In this chapter, we describe several parallelization approaches of a few tree searches. Most of sources are focused on solving games like Go or Chess using a parallel enrichment of the *Alpha-Beta* prunning technique. Few approaches are oriented on selecting the best move within a time limit or different game trees [23, 22, 5]. Since both of these approaches are more restricted than our goal, they are mentioned in short. In section describing Alpha-Beta algorithm, techniques as *best-first ordering* and *transpozition table* are covered because these techniques are used in algorithm in Chapter 3.

Throughout this chapter we will use the term *first son* / *child* for leftmost succesor of a node.

4.1 Alpha-Beta

Alpha-Beta is an algorithm that traverses a minimax game tree in a way to reduce number of searched nodes as much as possible. The algorithm is constructed to find the same result on a two-player game tree like minimax algorithm. Minimax algorithm performs a complete depth-first search on a two-player game tree. Alpha-Beta switches between maximizing and minimizing player. Two bounds are used.

- α is a bound used for remembering the highest score. α is used for cut-offs in minimizing nodes.
- β is a bound used for remembering the lowest score. β is used for cut-offs in maximizing nodes.

The bounds are sometimes called a *window* because they are supposed to form an interval from α to β. Pseudocode for Alpha-Beta is in Algorithm 4.

4.1.1 Node types

For designing any kind of parallel version of Alpha-Beta, an analysis of types of nodes is needed. Authors in [14] showed that in an arbitrary two-player game tree there is always a minimal tree that is traversed by Alpha-Beta. Figure 4 contains a complete two-player game three with depth 3; the minimal Alpha-Beta tree is highlighted by thick edges. In the mentioned paper three types of nodes are defined.

- Type 1 - every first child of type 1 is a type 1. Root node is a type 1.
- Type 2 - all children of a type 1 node are a type 2 except of the first child. All succesors of a type 3 node are a type 2.
- Type 3 - the first child of type 2 node is a type 3.

It holds that every 1, 2 and 3 node type is traversed by Alpha-Beta independently of terminal nodes values; in other words, node of these types forms a minimal Alpha-Beta tree [14].

```
Data: $h$ - node, $\alpha, \beta$, $i$ - player
Result: the best value for player in a zero-sum two-player game tree
if $h \in Z$ then
    return $u_i(h)$
end
if $i$ is maximizing player then
    for $a \in A(h)$ do
        $\alpha \leftarrow \max(\alpha, alphabeta(ha, \alpha, \beta, -i))$
        if $\alpha \geq \beta$ then
            break
        end
    end
    return $\alpha$
end
else
    for $a \in A(h)$ do
        $\alpha \leftarrow \min(\alpha, alphabeta(ha, \alpha, \beta, -i))$
        if $\alpha \geq \beta$ then
            break
        end
    end
    return $\beta$
end
```

Algorithm 4: Alpha-Beta pseudocode

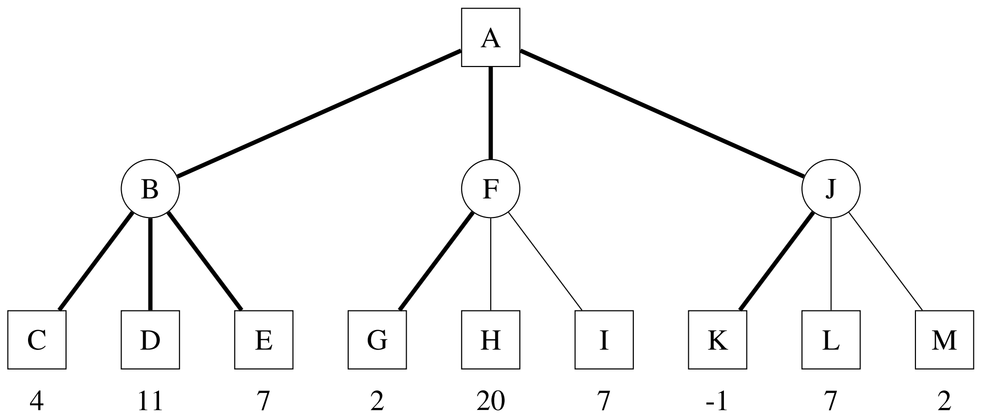

Figure 4 A highlighted minimal tree of a game tree searched by Alpha-Beta indepedently of values of terminal nodes. Rectangle nodes belongs to Player I, circle nodes belongs to Player II. In terminal nodes I player's payoff is shown.

These three types are ctritical for executing Alpha-Beta's cut-offs. Let's make an example to see a cut-off. Nodes A, B, C are type 1. Nodes D, E, F, J are type 2. Nodes G and K are type 3. Examination of B, the first child of the root, with $\alpha = -\infty$ and $\beta = \infty$ will yield 4 and all of its children, C, D, E, will be examined as well. Thereafter Alpha-Beta will start to evaluate node F with $\alpha = 4$ and $\beta = \infty$. Node G has got value 2 and therefore a cut-off occures and examination of node F yeilds 4. Then Alpha-Beta will evaluate node J with $\alpha = 4$ and $\beta = \infty$ and another cut-off will occure because K has got value -1. If node K or G have got bigger values than α, cut-off would not occure during computation of K and G.

4.1.2 Best-first ordering

Selecting the first child is crucial to ensure that Alpha-Beta would not do more work than is needed. Authors of [18] called this *strong ordering*. There are a few ways to approach a game tree to be a *strongly ordered tree*. A strongly ordered tree is such a tree having best-first ordering [18].

First of these approaches, the most straightforward can be used in games where utility function is non-zero for non-terminal nodes; children are sorted according their uitility function values in this case.

Second one, is to use *history heuristic*, which requires to store history of sufficiency of each action. A weight of sufficiency of each action is assigned to each action in [23]. Then, the weight of an action is increased if the node expanded by this action was usefull during the search. But this technique does not ensure that the most sufficient in a given node is the best-first move.

Last one was used in [18]. In this paper iteratively deepening was used to get weights for each action but sufficiency of each action was kept only as a node information, not a global history shared throught the whole game tree as in the previous case. In each iteration children are expandend in order according to their results from the previous iteration.

4.1.3 Other enhancements

In games, for example Go and Chess, a game state can be represented by multiple nodes. This is caused by representation of a game by a tree. To prevent multiple examination of a game state *transpozition table* can be used. This technique was used in [18, 23, 22, 19]. Transpozition table is a table that contains values of already computed game states. After a game state is examined for the first time, its value is assigned to the table. This method saves computation time in contrast to memory demands.

4.1.4 Complexity

Using the best-first ordering Alpha-Beta can traverse only the minimal tree. In [14] authors showed that in best case scenario, that is finding the best-first move ordering, the complexity is (8) where w is branching factor and d is depth the traversed tree.

$$w^{\lceil d/2 \rceil} + w^{\lfloor d/2 \rfloor} - 1 \tag{8}$$

In the worst case scenario Alpha-Beta would not cut-off any node and the complexity would be the same as minimax algorithm complexity, (9).

$$w^d \tag{9}$$

4.2 Losses

During a parallel computation of an algorithm like Alpha-Beta a few kinds of losses can occur and these are:

- starvation loss
- interference loss
- speculative loss

The starvation loss is caused by waiting a processor on a task, meanwhile the processor is doing nothing.

The interference loss is caused by critical sections and occurs when one processor is waiting for another to leave a critical section that the first one want to enter.

The speculative loss is caused by speculative work that is unnecessary. A parallel aglorithm uses speculative work to determine whether the work is mandatory.

4.3 Efficiency and speed-up

In this chapter also *efficiency* and *speed-up* are used to describe or classify an algorithm. In [8] speed-up and efficiency are defined as follow:

$$speed\text{-}up = \frac{time\ required\ by\ best\ serial\ algorithm}{timerequired\ by\ parallel\ algoritm} \tag{10}$$

$$efficiency = \frac{speed\text{-}up}{number\ of\ processors\ used} \tag{11}$$

4.4 Mandatory-Work-First

The main idea behind MWF is to reduce speculative loss. It is an algorithm based on Alpha-Beta idea. MWF starts its search with a parallel search of the minimal Alpha-Beta tree. Speculative work is permformed when more than the minimal tree must be searched. MWF was investigated in [7].

High level idea can be formulated as follow: minimal Alpha-Beta tree is traversed in parallel, thereafter if some of evaluated subtrees proves that more than minimal Alpha-Beta tree has to be traversed, then this subtree is evaluated again as this subtree belongs to a minimal Alpha-Beta tree. Repeated search over a subtree would not occure in best-first oredering trees. By *subtrees* children of a currently examined node are meant.

Let's have a parent node p of a type 1, its first child h and other children s. Firstly, the algorithm evaluates in parallel all children but differently according to whether a node is the first child or other child. The first child h is again evaluated in the same manner as his parent node p since both of them are of type 1. The rest of children s are evaluated in a way that only nodes of types 2 and 3 are examined in subtrees rooted in children s. After this parallel examination is completed another parallel examination of children s starts. The values of each children s are compared to the value gained by evaluation of the first child h; if any of these values are better than the first child's value, search on this particulary node (that has been evaluated as a node type 2) starts in a manner that this node is a type 1. Thereafter, gained value from a new search are stored and compared. The best gained value is returned.

In [7] comparsion of MWF with Palphabeta algorithm was performed. Palphabeta is a parallel version of Alpha-Beta algorithm on a tree of computers. Authors concluded that there is a significant iprovement of MWF over Palphabeta.

In [26] authors showed that best number of processors for MWF efficiency is around 6 on random game trees. More than 10 processors only lead to a starvation loss.

4.5 Tree-Splitting

This method investigated in [8] uses a tree of processors for parallel execution of Alpha-Beta. A master and slave relationship between processors is established. A master processor keeps its slaves busy as much as possible and also keep them informated about an actual window size. Each time a master processor examines a node, it expands all of the node succesors and sumbits them to other processors for evaluation. But these processors are both masters and slaves because they are masters to their slaves. Only a leaf processor is a slave and not the master one. In a leaf processors sequential Alpha-Beta is executed. If a node examined by a master processor has more children than number of available processors, then children are queued in a waiting queue until another processor is available.

In [8] authors shown that with best-first moves order tree-splitting reaches efficiency $O(1/\sqrt{k})$ where k is the number of used processors.

In comparison to MWF, no child of a node is evaluated multiple times and more interaction between processors is used. Also Tree-Splitting suffer because of speculative loss.

4.6 Principal Variation Splitting Algorithm

The parallel PVS is an Alpha-Beta whose main idea is to propagate a gained score back as soon as possible using a tree of processors. This approach was used in [22].

It traverses a game tree to a *splitting node* from which a parallel search on the node succesors is permformed. Always the first child is selected to be examined on the way to a splitting node. A splitting node is a node in a game tree whose distance from a terminal node is precisely the height of the processor tree used. The first child on the way from root to a splitting node is called a *principal variation candidate* in this case.

In [19] authors shown that efficiency drops approximately exponentially according to increasing number of processors.

One of the very first usage of PVS was called Palhphabeta and Calphabeta as mentioned in [18]. In this paper PVS variant with a narrow window $(score, score + 1)$ was used, where the *score* is the best found value so far. When this algorithm examines an unordered game tree, it traverses more nodes than Alpha-Beta. Enhancing the algorithm by iterative deepening brought better results due to the reordering of the traversed tree.

4.7 Aspiration Search

The window used for Alpha-Beta can be modified to give a better idea about the meaning of the bounds. When the examination of a node yields a value between α and β, the search was usefull and α is modified. A cut-off occured if the examination returns α, therefore the node has to be searched again with different window to determine the true value of the node.

With this method a node can be examined more than once, but the research is performed only when the previous search failed. The whole idea comes from the best-first ordering of moves, therefore reevaluating of a node is not considered to happen too often.

4.7.1 Parallel Aspiration Search

In [18] the parallel version of the aspiration search method was investigated. They proposed an algorithm that splits the window to disjoint intervals that are then examined in parallel.

The first possibility is to divide the window into two intervals where V is the estimated value of the search.

$$(-\infty, V) \qquad (V, \infty) \tag{12}$$

Another possiblity to divide the window into three intervals where V is the estimated value of the search and e is the expected error.

$$(-\infty, V - e) \qquad (V - e, V + e) \qquad (V + e, \infty) \tag{13}$$

In both of cases, each interval is examined by one processor.

This technique gains an advantage of sequential Alpha-Beta only in case that there is no good estimation for the used window. In comparison to sequential Alpha-Beta with a good window no speed-up is attained as shown in [19].

4.8 Young Brother Wait Concept

In [5, 17] YBWC was examined. The main idea of YBWC is to evaluate the first child of a node at the beginning of the search; only after that an examination of any of remaining children can be performed. In compare to PVS, the parallel search is possible at any node, but in PVS parallelism is allowed only in principal variation candidates.

There are two varaints of YBWC. For simplification of further reading, let's say that the *eldest brother* is the first child of a node and remaining nodes are *younger brothers* as in [17] were.

4.8.1 Weak YBWC

The master-slave relationship between processors is used. Examination of the root is assigned to a master processor. Then traversation of the game tree begins. After the eldest brother is evaluated, the processors select a split node, which means that a parallelization can start. Slave processors start to request for other processors to give them tasks and becomes a master processor to them. A *helpful master* concept can be used when a processor submitted more tasks than available processors are; in this case, the processor starts to computate one of the submitted tasks as well.

4.8.2 Strong YBWC

In this variant, four diferent types of nodes are established. The main difference is that all promising children must be searched before a parallism can be used instead of the eldest son. Promising children are supposed to produce a cut-off. Whether a node is promising or not depends on appliaction.

In [4] is shown that strong YBWC can perform better speed-up than weak YBWC, despite strong YBWC reduces possible parallelism.

4.9 ER

In [26] *ER* algorithm was introduced. Both MWF and PVS perform a selected speculative work. But both of them suffer for starvation loss, therefore the main idea of ER is to prove that speculative work is useful, so ER is ehnanced by a technique that no approaches described earlier.

The name comes from two types of nodes:

- E-node - an evaluating node.
- R-node - a refuting node.

Since an R-node is supposed to be refute in the search and all children of an E-node have to be evaluated, selection of an E-node is the important matter. Every non-terminal node has excatly one E-node among its children. They proposed to select E-node from the children accoring to their children values, so according to the grandchildren of the examined node. A slight similarity to the node type 1 and 2 can be seen here, because at least one child of node has to be evaluted to know whether the rest of the children can be prunned or not.

ER algorithm firstly evaluates most promising grandchildren of a node in parallel. Then an E-node is selected from the children of a node according the highest score of already evaluated grandchildren and the rest of the children are refuted.

4.10 Iterative deepening

Iterative deepening is a breadth-first search that runs in iterations with an increasing depth in each iteration. After the first terminal state is reached its value is yielded. This algorithm is less memory demanding and has the same asymptotic complexity as breadth-first search.

4.10.1 Iteratively deepening parallel Alpha-Beta

In [22] an iteratively deepening in parallel is described. Authors made a few stages in the computation.

The first step is to compute the first two iterations with all processors identical. Then moves from root are uniformly divided among procesors. In the third step, each processor computes a set of subtrees using Alpha-Beta with iterative deepening and with the same initial window (α and β). A time limit is used for computation in this case because the algorithm is constructed to yeild the result in a given time.

They compared this algorithm with PVS. The result was that PVS was slower in situations where mover ordering was poor.

4.11 Parallelization of AND/OR tree

In [13] a study of parallelization of AND/OR trees were performed. AND/OR tree is a game tree whose terminal nodes have assigned values 0 or 1. They proposed a parallel algorithm simplier than Alpha-Beta due to AND/OR tree structure but still similar. The main idea is to work in steps; in each step to evaluate w leaves in parallel and according to the results from this step the algorithm decide the next step. The algorithm stops when it computates the value of the root of a tree.

4.12 Expectiminimax

In this section expectiminimax algorithm and its possible parallelization are investigated. Expectiminimax algorithm is similar to minimax algorithm but it traverses a minimax tree with chance nodes. Chance nodes represent Nature player. The algorithm acts in the same way as minimax does in node of minimizing and maximizing player. In chance nodes the algorithm sums values of all children weighted by the probability distribution of actions. For example, backgammon is a game that can be formulated into a minimax tree with chance nodes and expectiminimax can be used for its solving.

In [15] an approach to expectiminimax paralelization was presented. The main idea is to examine all children of a chance node in parallel. This shows a significant improvement compared to a non-parallel version in non-deterministic games. But still, according to authors, investigation of paralelization of expectiminimax enriched with Alpha-Beta prunning technique or move ordering remains for future work.

5 Analysis of parallelization of Double-oracle sequence-form algorithm

In this chapter we describe possible parallelization of algorithm from Chapter 3. We can split this task into two separated areas - parallelization of Best-response sequence algorithm and parallelization of loops that can be executed independently.

5.1 Parallelization of Best-response sequence algorithm

Original version of Best-response sequence algorithm is described in section 3.4. The idea behind parallelization of this algorithm is to preserve used prunning techniques to perform only mandatory work. In this section we will refer with *searching player* to player for whom the best-response is calculated; other players, opponent and Nature, will be refered with *opponent player*.

5.1.1 Opponent's nodes

In opponent's and Nature's nodes Best-response sequence algorithm acts equally; only one node is searched. In this node type, a prunning technique described in section 3.4.1 ensures that mandatory work is performed in order to decide whether a currently examined node can participate in best-response. We refer to this node type as an opponent node no matter whether the opponent or Nature is playing in this node in this section.

Two situations during the evaluation of this node type can occure. The node is either fully evaluated or refused because a cut-off had occured. A cut-off is performed when a node cannot be a part of the best-response. Parallel examination of all children of an opponent's node can lead to more and longer work than is needed with the prunning technique, therefore another concept of parallelization is proposed in this section.

To preserve the prunning technique strong YBWC is used. The idea of strong Young Brother Wait Concept is that parallel examination of *young brothers* can start only if *elder brothers* are already examined. Firstly, let's remember that we do know bounds on utility values; these are *MinUtility* and *MaxUtility*. We also know weights of all children of a node. Finally, let's remember λ' bound representing the minimum necessary profit of the children ensuring that the currently examined node participates in the best-response assuming that all unsearched children yield maximal possible profit. Now, let's combine these things together to get a *split point* among the children of an opponent's node. A split point divides sequential and parallel examination of the children of an opponent's node; in other words the children before a split point are elder brothers that have to be examined sequentially and children behind a split point can be examined in parallel.

Let's label all children of an opponent's node h as elder brothers until the profit gained so far is bigger than the loss of remaining children under assumption that all remaining children yield their minimal possible profit, respectively their biggest loss since the algorithm solves a zero-sum game. The execution of elder brothers is performed sequentially in descending order of their probabilities. After execution of each elder brother λ' is updated for the next elder brother. Since (14), a split point can be formulated as (15) where w_h represents sum of weights of the unsearched children. When a split point is achieved the remaining children can be examined in parallel. Pseudocode for this proposal is in Algorithm 5. Notice that a split point does not have to be reached at all.

$$\lambda' + w_r * MinUtility > 0 \tag{14}$$

$$\lambda' > -w_r * MinUtility \tag{15}$$

In other words, more informally, until reaching a split point we do not know whether a currently examined node *h* can participate in the best-response. Therefore children of *h* are examined in such a way a cut-off may occure using the prunning technique. Once a split point is reached, *h* definitely may be a part of the best-response, thus all of its children have to be evaluated to know the true value of *h*.

The children of a node *h* are given to a queue in descending probabalities order (line 5). Then all elder brothers are executed in sequential order (lines 7 - 22). In each iteration λ' bound is computed (line 10). If all elders brothers are examined, sequential examination ends (lines 11 - 13) and parallel examination of the remaining children is executed (lines 23 - 28). In parallel search over younger brothers a *lock* is needed to avoid any data corruption (line 25 - 27). A queue is used; *peek()* returns the first element of a queue and *delFirst()* delete the first element of a queue.

The method proposed above ensures that no surplus work is performed compared to Best-response seqence algorithm. For this proposal, one more reason occurs when there are multiple actions from a node *h* leading to one information set. In this case, interference loss may occure because of one information set may be requested by a multiple set to evaluate. This loss is dependent on the usage of a parallelization concept search over an information set, respectively nodes of the searching player. We present a concept for parallelization of a search over an information set in the following section where multiple requests over evaluation of one information set leads to a computation loss.

5.1.2 Searching player's nodes

In this section we examine parallelization of a node *h* belonging to an information set H' in which the searching player performes his move. Let's remember that we have a set of nodes H', from each of them a set of action is leading to other nodes in a game tree.

If a simply run was performed one parallel loop in another parallel loop as shown in Algorithm 6 a prunning technique from section 3.4.2 could not be used. Therefore more than actually needed children leading from information set H' would be examined and the parallelization could be slower than the sequential search with the prunning technique described in section 3.4.2.

```
1 for $h' \in H'$ do in parallel
2     for $a \in A(h')$ do in parallel
3         search node $h'a$
4     end
5 end
```

Algorithm 6: An example of a parallel loop inside of another

There is a possibility to examine action leading from a node *h* belonging to H' in parallel. This is allowed by the fact that a searching player can determine an information set according to actions that he has performed. When he performs two different actions from *h* in parallel, it means that two different subtrees are examined. So, during the parallel computation no node is shared between these two examined subtrees; in other words, these subtrees do not

```
Data: $i$ - searching player, $h$ - current node, $\overline{r}'_{-i}$ - opponent's strategy, Min/MaxUtility -
      bounds on utility values, $\lambda$ lower bound for a node $h$
Result: expected value of strategy against the extended strategy of the opponent
1  if $h \in Z$ then
2      return $u_i(h) * \overline{r}'_{-i}(seq_{-i}(h)) * C(h)$
3  end
4  $w \leftarrow \sum_{a \in A(h)} \overline{r}_{-i}(seq_{-i}(ha)) * C(ha)$
5  $queue \leftarrow$ sort $A(h)$ according to probability
6  $v^h \leftarrow 0$
7  while $queue$ is not empty do
8      $a \leftarrow queue.peek()$
9      $w_a \leftarrow \overline{r}'_{-i}(seq_{-i}(ha)) * C(ha)$
10     $\lambda' \leftarrow \lambda - [v^h + (w - w_a)*\text{MaxUtility}]$
11     if $\lambda' > -w * MinUtility$ then
12         break
13     end
14     if $\lambda' \leq w_a * MaxUtility$ then
15         $v_a \leftarrow \text{BRS}_i(ha, \lambda')$
16         $v^h \leftarrow v^h + v_a$
17         $w \leftarrow w - w_a$
18     else
19         return MinUtility $*w$
20     end
21     $queue.delFront()$
22 end
23 for $a \in queue$ do in parallel
24     $v_a \leftarrow \text{BRS}_i(ha, -\text{MinUtility})$
25     lock
26     $v^h \leftarrow v^h + v_a$
27     unlock
28 end
29 return $v^h$
```

Algorithm 5: Parallel BRS$_i$ in opponent's nodes

share any resources. In contrast to two parallel loops from the previous paragraph, this is more promising since in the previous case there can be shared resources which slow down a parallel computation. In addition, the prunning technique for this node type can still be used. A pseudocode for this proposal is in Algorithm 7.

Firstly, an information set H' containing a requested node h is locked (line 4). The first thread that requested to compute information set H' is given control over this set. Then, each node in H' is examined in the descending probabilities order like in non-parallel version with the exception of a parallel examination of actions leading from a currently examined node (lines 14 - 29). The prunning technique described in section 3.4.2 is used as well (lines 17 - 25). Finally, computed values are stored (line 30) and the information set is unlocked (line 31). Meanwhile a thread is computing an information set, other threads that also requested to get a node from this information set are waiting till the first thread completes its work; thereafter others threads take store value (line 6).

```
Data: $i$ - searching player, $h$ - current node, $\overline{r}'_{-i}$ - opponent's strategy, Min/MaxUtility -
      bounds on utility values, $\lambda$ lower bound for a node $h$
Result: expected value of strategy against the extended strategy of the opponent
if $h \in Z$ then
    return $u_i(h) * \overline{r}'_{-i}(seq_{-i}(h)) * C(h)$
end
lock this information set
if h has been evaluated then
    return stored value
end
$H' \leftarrow \{h'; h' \in I_i^k\}$
sort $H'$ descending according to value $\overline{r}'_{-i}(seq_{-i}(h)) * C(h)$
$w \leftarrow \sum_{h' \in H'} \overline{r}'_{-i}(seq_{-i}(h)) * C(h)$
$maxVal \leftarrow -\infty$
$v_a \leftarrow 0 \; \forall a \in A(h')$
$w \leftarrow \sum_{a \in A(h)} \overline{r}_{-i}(seq_{-i}(ha)) * C(ha)$
for $h' \in H'$ do
    $w_{h'} \leftarrow \overline{r}'_{-i}(seq_{-i}(h')) * C(h')$
    for $a \in A(h')$ do in parallel
        if maxAction is empty then
            $\lambda \leftarrow w_{h'} *$ MinUtility
        else
            $\lambda \leftarrow (v_{maxAction} + w *$ MinUtility $) - [v_a + (w - w_{h'}) *$ MaxUtility $]$
        end
        if $\lambda \leq w_{h'} * MaxUtility$ then
            $v_a^{h'} \leftarrow \mathrm{BRS}_i(h'a)$
            $v_a \leftarrow v_a + v_a^{h'}$
        end
    end
    $maxAction \leftarrow \mathrm{argmax}_{a \in A(h')} v_a$
    $w \leftarrow w - w_{h'}$
end
store $v_{maxAction}^{h'} \; \forall h' \in H'$
unlock this information set
return $v_{maxAction}^{h}$
```

Algorithm 7: Parallel BRS$_i$ in playing player's nodes

5.2 Parallelization of independent loops

The second area of possible parallelization of the algorithm described in Chapter 3 is area of independent loops. By *independent* is meant such a loop that its execution does not depend on previous iterations of the loop.

In the existing code there are few independent loops, for example inside *traverseCompleteGameTree* method in *GeneralDoubleOracle* class. Since this particular loop is used only for computation the size of a game tree, it has no effect on Double-oracle sequence-form algorithm's speed. Thus its parallelization is useless and was not implemented.

In *GeneralDoubleOracle* class and *SQFBestResponseAlgorithm* class, which implements Best-response sequence algorithm, are some independent loops, but these loops work with a small set of seqeuences or with players. Let's imagine that we have two players and for each of them we want to do some work, for example copy their best-response sequences from one data structure to another. Would it be efficient to make new tasks, assign them to threads while the destination data structure is shared for both players? The destination data strucutre has to be thread-safe in order no data corruption can occure. In addition, would this parallelization of few players or sequences executed in parallel have a significant improvement while a few instructions are executed inside the loop?

In *DoubleOracleConfig* class there is a loop that in *initializeRG* method that constructs the restricted game in each iteration. The method uses depth-first search throught the game tree in order to construct the restricted game. This is a promising loop for parallelization but on the other hand, there are few shared resources that are used for all nodes in the restricted game. So, speed-up by parallelization of this loop and the parallelization efficiency are not ensured in this case.

5.3 Summary

In this chapter we proposed parallelization of Doube-oracle sequence-form algorithm. Our main goal was to propose such a parallelization that will not do more work than original non-parallel Best-response sequence algorithm. We were also focused on parallelization of independent loops but it occurs that this type of loops is not used in the algorithm. However, we try to parallelize a loop inside *initializeRG* method which creates the restricted game to see if there is any speed-up or wheather the loop si independently in the first place.

6 Implementation

6.1 Frameworks and libraries for parallelization

Before deciding which framework or library to use for parallelization a survey of these technologies was performed. The survey was focused on parallelization technologies and frameworks that use functional programming's approach because this approach allows executing a function over each element of a collection. Moreover, execution of a function over a collection can be done in parallel, thus it can be interpreted as a parallel foreach loop over a collection. Notice, that no comparsion between these frameworks was found. These frameworks were investigated:

- Functional Java
- Java 8
- Lambdaj
- Guava
- Ateji PX
- JPPF

Functional Java [10] is a library that is focused on some concepts that are not present in Java (at least until Java 1.7). For example, it allows to use *map, zip, fold* function over a collection. Also it has a parser for monads.

Java 8 [20, 21] brings to Java some functional's approaches. There are *functional interfaces* which are interfaces containing precisely one method. Also new to Java are *closures* also called *anonymous method*. Also the collection interface is enhacemented by *parallelStream* method that returns a parallel stream of a collection. On a parallel stream one can call *foreach* method that takes a closure. *parallelStream* is implemented in the way that the collection is divided to parts and each part is executed by a thread. A fork-join design pattern is used in parallelStream; the pattern works in the way that the main thread assigns work to other threads (it forks threads) and after all forked thread finish their work the main thread continues.

Lambdaj [16] is a library which was designed to make Java code more readable; in details, clean and readable work over collections is the main idea of it. It brings to Java foreach, filters and closures to Java. But no parallelism in foreach loops is used.

Guava [9] is a library from Google that brings to Java functional approach over collection using interfaces for predictors and functions. This library was written before release of Java 8 which enables some of funcional approach.

Ateji PX [1] is a framework that allows to use parallel for loops in Java. The framework is released in a form of a plugin to Eclipse IDE.

JPPF [12] is a framework for computation Java application on cloud. It is mentioned here as another possibility for parallelization. It works in the way that the task is separated to subproblems and each of the subproblem is computed in parallel.

6.2 Used technologies

The project was coded in NetBeans IDE using these technologies:

- Java 8

- org.apache.wicket.util.collections
- java.util.concurrency

Java 8 was used because of its functional and parallel approach to collections. In addition, while continuing in expansion of the framework in which Double-oracle sequence-form algorithm is implemented no additional framework is needed, only version of Java 8 is needed.

The two remaining libraries was used because they implements conccurent collections and locks that was used in the implementation, for example *ConccurentHasSet* was used.

6.3 Implementation

6.3.1 Best-response sequence algorithm

Double-oracle sequence-form algorithm is implemented in *GeneralDoubleOracle* class which uses *DoubleOracleBestResponse* class to calculate the best-response. *DoubleOracleBestResponse* class extends *SQFBestResponseAlgorithm* class which implements Best-response sequence algorithm.

In Chapter 7 three versions of parallelization of Best-response sequence algorithm are compared to non-parallel version; these are:

- Best-response sequence algorithm with parallelization of searching player's nodes implemented in *DoubleOracleBestResponseParallelSearchingPlayerOnly* class
- Best-response sequence algorithm with parallelization of opponent's nodes implemented in *DoubleOracleBestResponseParallelOpponentPlayerOnly* class
- Best-response sequence algorithm with parallelization of both opponent's and searching player's nodes implemented in *DoubleOracleBestResponseParallel* class

In order to test these three variants, they have to be implemented. They differs only in implementation of *selectAction* method that calls the best-response recursively call if cut-off has not occured. Since they differ only in implementation of one method they can be implemented as an extension of another class that represents thread-safe Best-response sequence algorithm; that thread-safe best-response is implemented in *DoubleOracleBestResponseConccurent* class. The *DoubleOracleBestResponseConccurent* class has to be an extension of *DoubleOracleBestResponse* class in order to be possible to use the class in *DoubleOracleBestResponse*.

The *DoubleOracleBestResponseConccurent* class was implemented in almost the same way as the *SQFBestResponseAlgorithm* class. The only difference is that conccurent collections are used for shared resources and *synchronized* was used for locking critical sections. Also this conccurent class is an abstract one because it has got two abstract method that have to be implemented according to what type of parallelization is wanted; each of the methods represents *selectAction* method for one type of player (searching player, opponent). In order to calculate a split point in opponent's nodes a new method was added to a new class *BROppSelectionConcurrent* which extends *BROppSelection* defined in *SQFBestResponseAlgorithm* class.

It is surely not the best design because some methods are overriden (*calculateBR*, *bestResponse*, *calculateEvaluation*) but it was implemented in this way because of complexity and size of the framework that Double-oracle sequence-form belongs to.

Unit tests were written for the implemented classes.

6.3.2 Implementation of independent loops

Implementation of independent loops is not covered in the output of this work. This is mainly because the the most promising independent loop in *initializeRG* method from *DoubleOracleConfig* class is not as independent as it seems to be. The order of expansion restricted game's nodes have some side effects on the whole Double-oracle sequence-form algorithm. Moreover, as shown in section 7.3.1 time consumed by constructing the restricted game is few times smaller than time of Best-response sequence algorithm.

7 Experiments

7.1 Methodology

Few types of games were selected for measuring speed-up and efficiency achieved by the proposed parallelization design. These games were submitted to Double-oracle sequence-form algorithm which called all of tested BRS versions during each iteration meanwhile collecting their execution time. Each test was performed serveral times, then arithmetic average and variace was computed from total of measured data for each BRS version. Smaller instance were tested ten times; bigger instances, such tests that run longer than 30 minutes, were performed five times.

There are two figures (figs. 7, 8 and 12), each representing execution time of tested BRS versions divided by time consumed by execution of sequential BRS in the same iteration. Each of these figures was computed from only one run of Double-oracle sequence-form algorithm. They are shown only for better imagination of how the game tree of the restricted game can differ in each iteration of Double-oracle sequence-form algorithm. With increasing number of iterations of the algorithm, the restricted game increases its size. By combining this knowledge with the assumption that parallel BRS is faster on large game trees in comparison to sequential BRS, meanwhile on small game trees parallel BRS does not have to show any improvements, one gets that with increasing number of iterations of the algorithm parallel BRS may increase its speed-up against sequential BRS.

Efficiency defined in section 4.3 was defined according to Fishburn's paper [8]. Since early 80's computers and processors have been evolved. Processors with Hyper-threading allow to act as they have double of cores than they have physicaly. Therefore, while computing efficiency one should remember Hyper-threading and use number of logical processors used as number of processors in (11) instead of number of used physical cores.

Experiments were performed on Intel(R) Core(TM) i7-2630QM CPU @ 2.00GHz with 4 cores with Hyper-threading, 8 GB RAM and 64 bit Windows 7.

7.2 Experiments with Best-response sequence algorithm

In this section, three possible combination of parallelization described in 5.1 are experimentally compared to the original algorithm described in section 3.4; these are:

- parallelization in searching player's nodes
- parallelization in opponent's nodes
- parallelization in nodes of both players

To make legends of graphs more readable, *parallel searching player*, *parallel opponent* and *parallel BRS* terms are used in this section to refer parallelization of searching player's nodes, opponent's nodes and parallelization in nodes of both players respectively. Notice that all BRS versions was with implemented with prunning described in section 5.1.

Graphs in figures are divided into subgraphs in order to be readable. This is caused by the fact that size of experiments grows exponentially with respect to depth of a game tree. While using a linear y axe (time), results of experiments with the smallest examined depth are hardly to see. While using log axe for time, all results are big enough to be seen but defferences between measurements of the same game tree depth are hardly to be recognized since their differnces are linear and the axe is logarithmic. Therefore results of experiments were divided to subgraphs according to the game tree depth.

In following experiments, such settings of games that have impacts on the size of game tree were set in the way, there would be smaller and larger game trees during the experiments; this is caused mainly by depth of a game tree and branching factor. Some other settings that have impacts on size of informations were tested, for example by allowing or fobridding slow moves in Border patrolling games.

7.2.1 Phantom Tic-Tac-Toe

Two experiments with Phantom Tic-Tac-Toe on 3×3 board was performed. Phantom Tic-Tac-Toe is a game that was used in most examples in this work; its description is in section 2.1.1.

In the first experiment, first two moves of the game was forced for both players; in other words, both players was set up to peform their first moves on one specific coordinate. This was used to narrow state space. The first player is forced to play his first move on coordinate $[2; 2]$ (rows and columns are numbered from 1); the second player is forced to do the same thing in his first move. In Figure 5 comparison of arithmetic averages of BRS variants is shown; in Table 2 variances of BRS versions are shown. This experiment shows almost double speed improvement of parallel BRS and parallel searching player over sequential BRS and no improvement of parallel opponent. Figure 7 shows each BRS version divided by time of sequential BRS with respect to iterations.

The second experiments differs from the first one only by the fact that only the frist player was forced to play to coordinate $[2; 2]$. The results of this experiments are in Figure 6 and Table 2. This experiment shows also almost double speed improvement of parallel BRS and parallel searching player over sequential BRS. In contrast to the previous experiment, in this case, parallel opponent slighly better than sequential BRS. Figure 8 shows each BRS version divided by time of sequential BRS with respect to iterations.

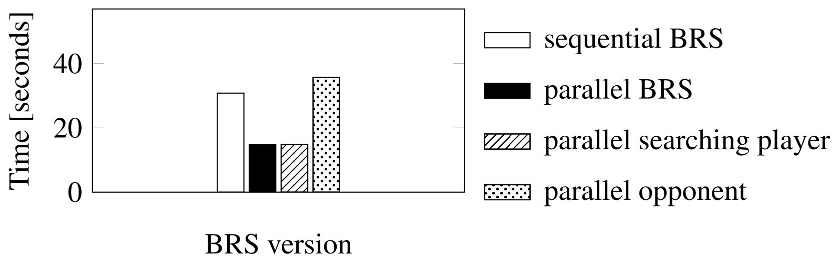

Figure 5 Comparison of arithmetical averages of BRS versions time on Phantom Tic-Tac-Toe on 3×3 board with forced first two moves to $[2; 2]$ coordiante for both players.

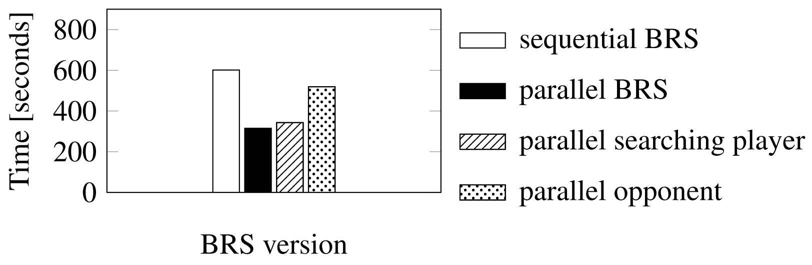

Figure 6 Comparison of arithmetical averages of BRS versions time on Phantom Tic-Tac-Toe game with forced the first move of the first player to coordinate $[2; 2]$.

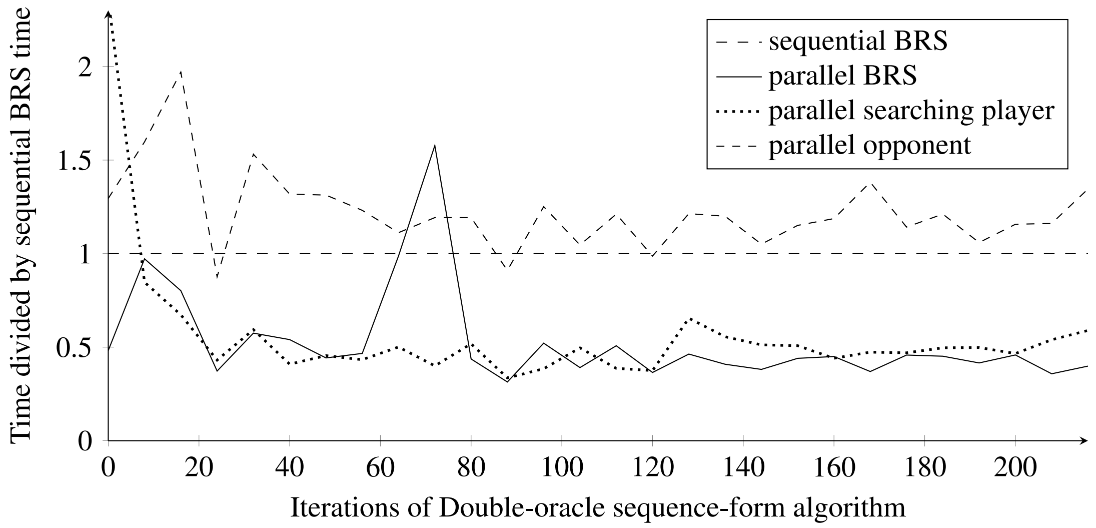

Figure 7 Comparison of BRS versions to sequential BRS in iterations of Double-oracle sequence-form algorithm on Phantom Tic-Tac-Toe 3×3 with forced first two moves on coordinate [2; 2] for both players.

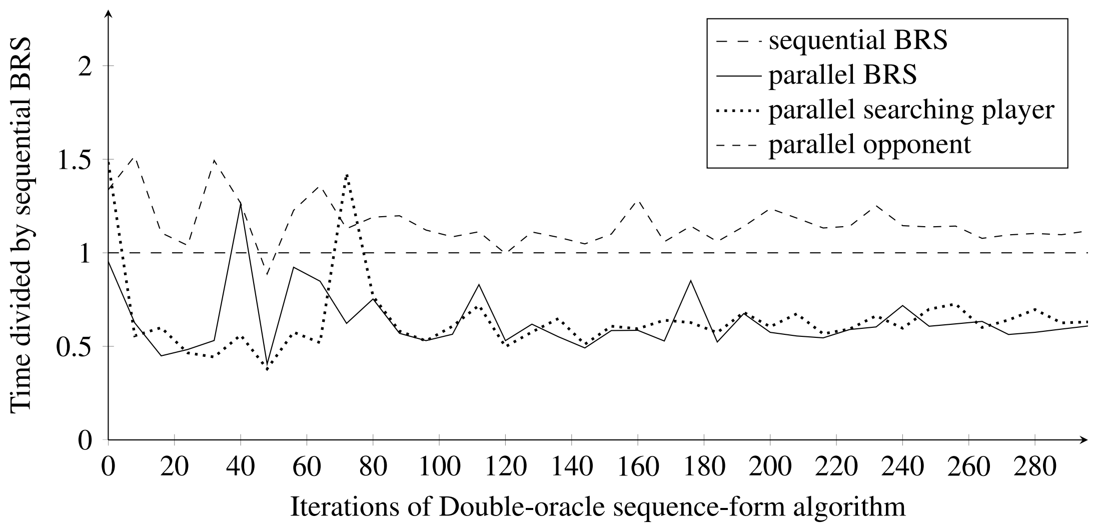

Figure 8 Comparison of BRS versions to sequential BRS in each iteration of Double-oracle sequence-form algorithm on Phantom Tic-Tac-Toe 3×3 with forced the first move of the first player.

Table 2 Variances of time of BRS versions on 3×3 board with forced move(s) on [2; 2] coordinate in $[s^2]$.

	sequential BRS	parallel BRS	parallel searching player	parallel opponent
first player forced	3401.5	128.019	302.216	138.554
both player forced	0.0430	0.3304	0.8968	1.7079

7.2.2 Random Game

Random game is a two-player zero-sum game with imperfect information which has got some settings that are described in this section. The depth of a Random game tree is double the size of *d* which represents how many decisions can a player perform. Number of possible actions is represented by *branching factor*. The following model for Random game was used for following experiments: a leaf utility is total of each action's utility on path from the root to the leaf; for each action is computed random number. Seed $s = 3$ for random number generator was used in all experiments. Each player can distinguish only a predefined number of opponent's actions; *max observations* expresses this pedefined number. During experiments with instances of Random game other setting as *max center modifications* was set to 5 and *keep observations probability* was set to 0.9.

Three experiments were performend on Random games with different settings; these are shown in figs. 9 to 11. In each experiment different values for *max observations* and *branching factor* are used. Each experiment was performed with two or three depths. Remaining settings of games are described in the previous paragraph. In these experiments parallel opponent always finished after sequential BRS except the case on Figure 9 with d = 3. however, it is a small case and such result has not been recorded on other cases. Parallel searching palyer has got unclear results in these experiments since in some cases it was faster than sequential BRS but in smaller cases (d = 2) it was even two times slower (see figs. 10 and 11). Parallel BRS was faster in all cases except the ones with small depth (Figrue 10 with d = 2 and Figure 11 with d = 2). In two cases parallel BRS was almost three times faster then sequential BRS (Figure 10 with d = 5 and Figure 11 with d =4).

In Figure 12 the first two thousand iterations of an instance of Random game with settings $max\ observations = 3$, $depth = 6$, $branching\ factor = 4$, are shown with divided time of each BRS version by sequential one.

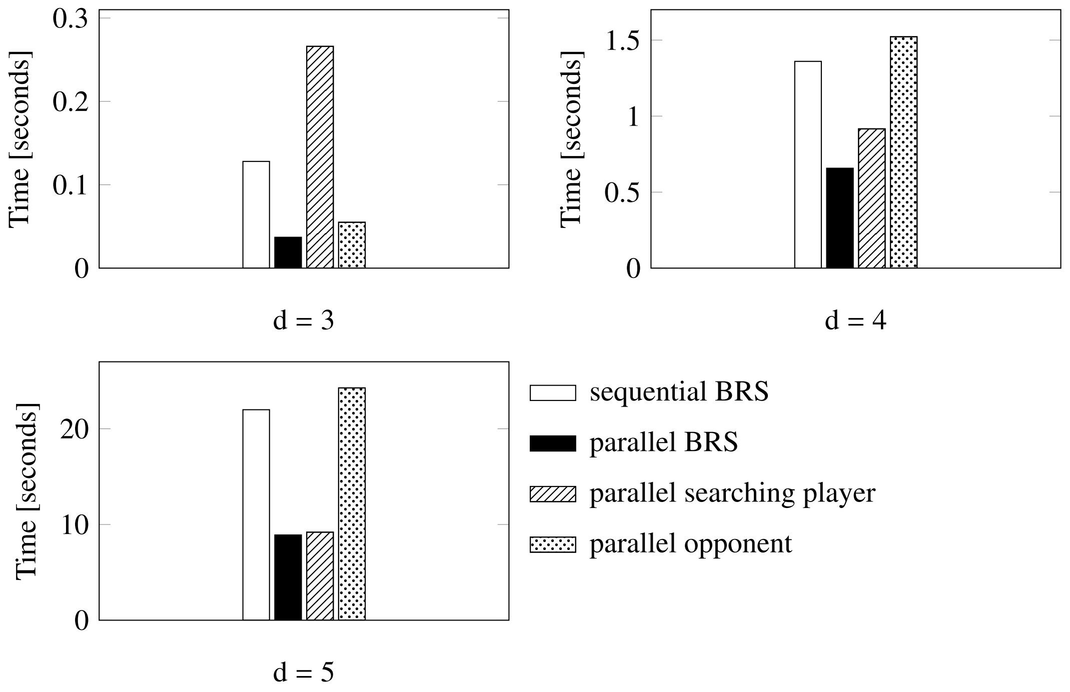

Figure 9 Comparison of arithmetical averages of BRS versions time on a Random game with respect to depth and settings: $max\ obeservations = 2$ and $branching\ factor = 3$.

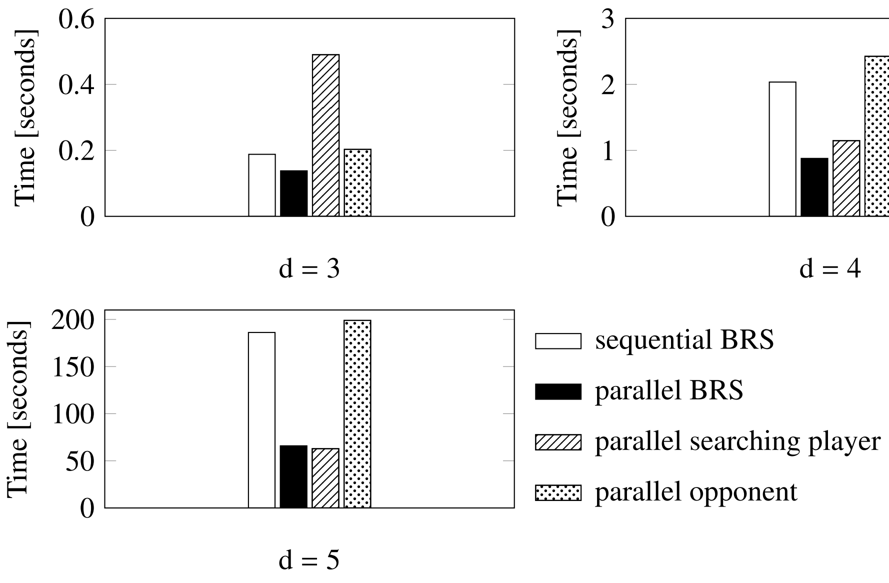

Figure 10 Comparison of arithmetical averages of BRS versions time on a Random game with respect to depth and settings: $max\ obeservations = 2$ and $branching\ factor = 5$.

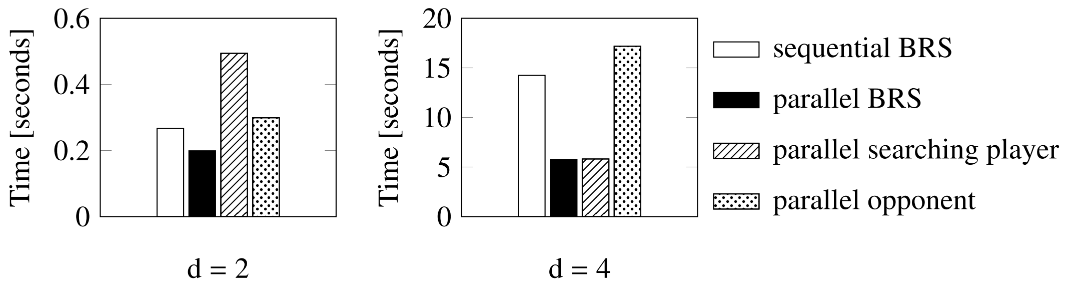

Figure 11 Comparison of arithmetical averages of BRS versions time on a Random game with $max\ obeservations = 3$ and $branching\ factor = 6$ with respect to depth.

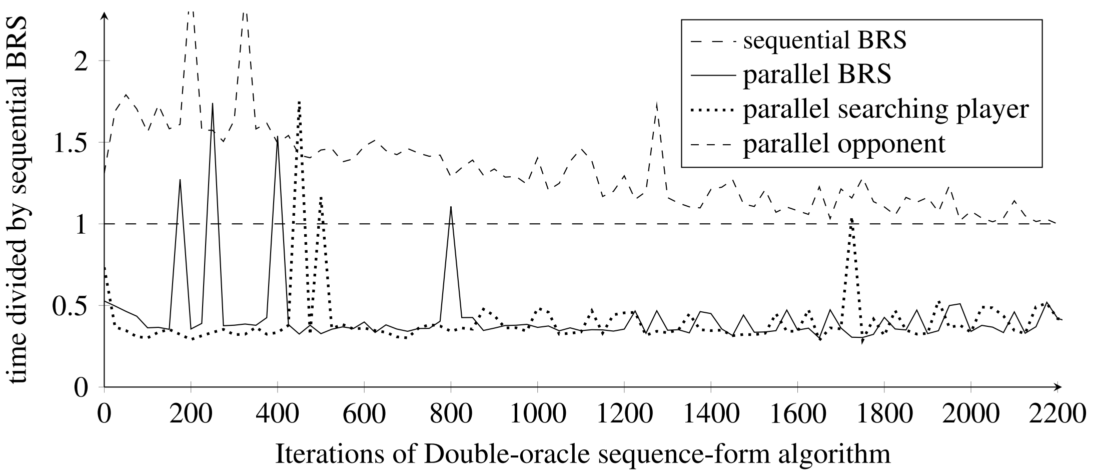

Figure 12 Comparison of BRS version to sequential BRS in the first two thousand iterations iterations of Double-oracle sequence-form algorithm on Random game with settings: $max\ observations = 3$, $depth = 6$, $branching\ factor = 4$.

7.2.3 Generic Poker

Generic poker is a version of poker in which number of antes and number of maximal raises in row can be set for one player. In experiments with Generic poker number of antes for one player was set to 2 and number of maximal raises in row was set to 2. Besides, max different bets and max different raises was set to 2. The main thing corresponding to the game size is number of maximum card types, which was set to 2, and max card of each type, which was set to 2. Experiment's results are shown in Figure 13 and Table 3. Parallel opponent was again slower than sequential BRS. Parallel BRS and parallel searching player achiev almost double speed of sequential BRS.

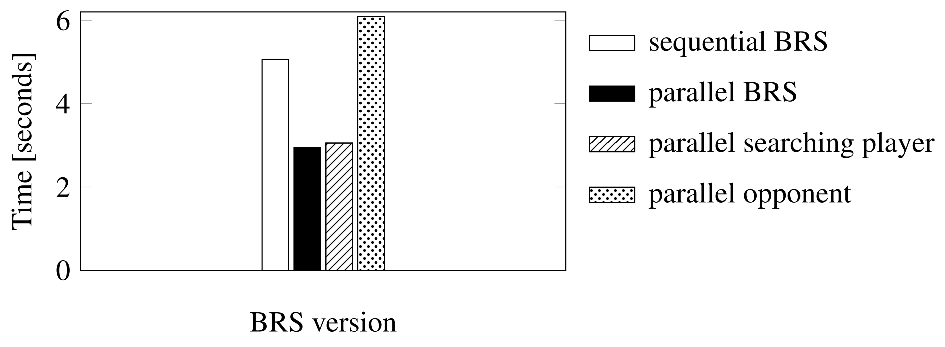

Figure 13 Comparison of arithmetical averages of BRS versions time on a Random game experiment.

Table 3 Variances of time of BRS versions on the Generic poker experiment in [s^2].

	sequential BRS	parallel BRS	parallel searching player	parallel opponent
variance	0.0189	0.0223	0.0118	0.0323

7.2.4 Border patrolling

Border patrolling game is a type of search games described in 2.1.3. Two players act simultaneously on a directed graph. One player, evader, wants to get from start destination to finish destination. He wins if he gets to goal destination within a predefined number of moves. The second player, patroller, aims to catch evader. If patroller catches evader within a predefined number of moves, he wins. If the predefined number of moves are exceeded and no player wins, a draw occurs. While evader is moving, he leaves track; this can be avoided by using *slow moves* by evader cause slow moves do not leave tracks. Evader also can use wait action to stay in a node as his move. Experiments with different graphs were pefromed with allowed or forbiden slow moves action (see figs. 14 and 15). Graphs used in experiments are in Appendix A tables 8 and 9. The graph from table 8 will be refered as G1 and the one from table 9 will be refered as G2. In experiments *depth* representing number of allowed moves for players and usage of slow moves were examined. In both cases destination node was the 17th node of a graph and evader starting node was the first one node of a graph. Also in both cases patrolling player starts in 18th and 19th node of the graph and from these nodes he starts his patrolling.

In the first experiment on G1 there is no improvement for any of parallel versions (see Figure 14). Despite of that, the second experiment on G2 has shown speed-up around 2.5 for parallel BRS and parallel searching player over sequential BRS. As in most of previous experiments, parallel opponent shown no improvement over sequential BRS. In the first experiments, no improvement may have been caused by the state space which contains large information sets.

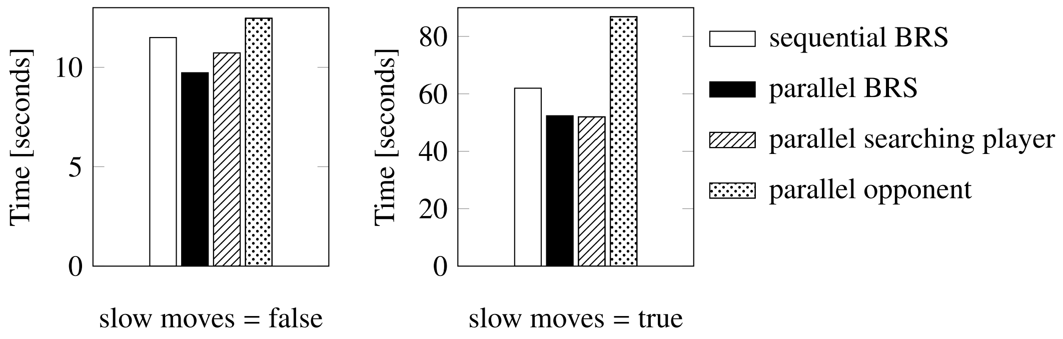

Figure 14 Comparison of arithmetic average of BRS versions time on a Border patrolling game with *depth* = 4 on G1.

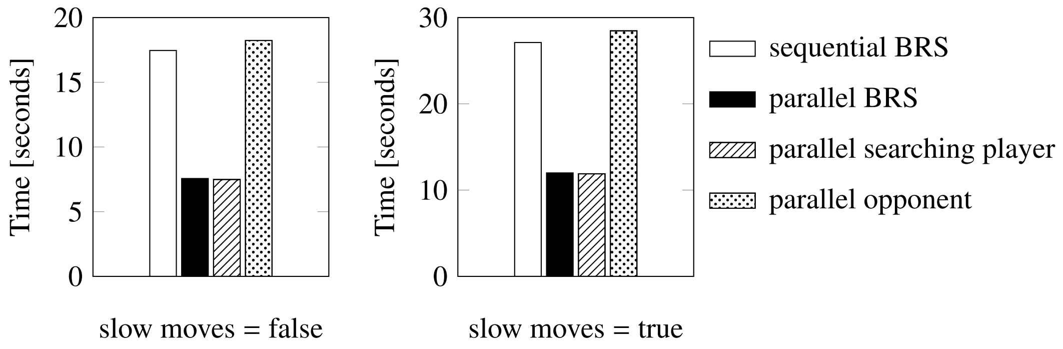

Figure 15 Comparison of arithmetic average of BRS versions time on a Border patrolling game *depth* = 5 on G2.

7.2.5 Conclusion

Data from these experiments were taken and speed-up was computed for each record. In Table 5 are corresponding results to all tested variants of BRS.

Table 4 Speed-ups computed from all measurements.

	parallel BRS	parallel searching player	parallel opponent
median	2.184	1.753	0.906
arithmetic average	2.095	1.587	0.992

Since it is evident that parallel computation takes some time because of usage of parallelism, results computed in the same way as above can be calculated with forgetting the experiments on small games; these are ment those game in which average of sequential BRS took less than 1 second (these are experiments shown in figs. 9 to 11 with the smallest *d* in each experiment). These results are in Table 5.

Table 5 Speed-ups computed from measurements without ones from small cases.

	parallel BRS	parallel searching player	parallel opponent
median	2.222	1.873	0.899
arithmetic average	2.105	1.866	0.894

Median is the best likelihood estimation for 1-D measurement, according to that parallel BRS has the biggest speed-up. But still, with a four-core processor with Hyper-threading its efficiency is only $27,7\%$. In Table 6 efficiency is computed according to median from Table 5.

Table 6 Efficiency computed using median from Table 5.

	parallel BRS	parallel searching player	parallel opponent
efficiency	0.2778	0.234	0.112

Figures 7, 8 and 12 were shown to imagine how long it takes to compute the best-response in each iteration to each BRS version. Since parallel BRS has been found to be the most efficient version, it was examined in Figure 12 case. It has shown that in some iterations speed-up of parallel BRS against sequential one was about 5 which means efficiency slightly over 60%. For computation of this number, the speed-up of 5, the graph shown in Figure 12 was traversed and its values representing *parallel BRS time / sequential BRS time* were examined; about ten of these ratios were around 0.2.

To summarize, there are speed-ups around 2 but also a pathological case of one Border patrolling game instance with no speed-up at all. Thus the proposed parallelization of parallel BRS does not have to be faster than sequential BRS. This is probably caused by the structure of the search space in this case.

7.3 Experiments with Double-oracle sequence-form algorithm

7.3.1 Restricted game building time

Measurements from the previous section also provided more data about the whole algorithm. In section 3.2 Double-oracle sequence-form algorithm is divided into three steps - Best-response sequence algorithm, LP computation and the restricted game building. Time required for these three part was recorded and then perctentage of each part was computed for each experiment. The results from this are in Table 7.

Table 7 Median and arithmetic average from percetage occurrence of LP, BRS and restricted game building.

	LP	BRS	restricted game building
median	0.288	0.534	0.054
arithmetic average	0.338	0.562	0.098

From Table 7 one can see which parts of the algorithm are the most time consuming. So, building of the restricted game contributes to the whole time of the algorithm by only a small amount of time in comparison to remaining parts. Therefore parallelization of this part of the algorithm would not changed much the overall time.

8 Conclusion

This work described basic background of Game theory, principles Double-oracle sequence-form algorithm uses and made a survey of parallalization search on game trees. Contribution of this thesis is a parallel design of Best-response sequence algoritm, its implementation and its experimental evaluation.

Chapter 2 desribes basic background of Game theory and imperfect-information games; also expamles of imperfect-information games are described. In chapter 3 an analysis of Double-oracle sequence-form algorithm from high level point of view was done. In chapter 4 a survey of parallel techniques for game tree search is presented. In this survey are described techniques relevant to Best-response sequence algorithm. Chapter 5 presents possible parallelization of the algorithm which was implemented (chapter 6) and experimentally evaluated in chapter 7. The analysis was mainly focused on Best-response algorithm but reamining parts of Double-oracle sequence form algorithm was examined as well.

The analysis of possible parallelization of Best-response sequence algorithm was designed under the assumption that cutt-off in Best-response sequence algorithm are very useful for narrowing search over a game tree. Also a proposal for parallelization of the algorithm without prunning techniques or with partial usage of cut-offs is proposed in future work.

Experiment, described in the previous chapter, tell us that speed-up of the proposed parallel design from Chapter 5 is statisticaly 2.22 with efficiency of 27.7% on card games, Phantom Tic-Tac-Toe and Border patrolling and Random games. But these results were achieved on a four-core processors with Hyper threading. Since speed-up and mainly efficiency may differ according to used proccesor, the experimental part of this thesis was not fully elaborated and may remain for future work.

Although, contribution of this thesis is a prallel version of Best-response sequence algorithm and the proposed parallel version may be used for comparing future paralleling approaches of the algorithm.

8.1 Future work

The lesson we learned from experiments is that sticking with a cut-off technique in parallel search is not always as time saving as it may seem. As seen on results of Best-response sequence version with opponent nodes parallelization; using only this approach is in most of cases slower than sequential algorithm. This is caused by speculative loss.

Since one instance of Border patrolling game has shown almost no improvement, some other knowledge about a game's state space may be helpful. More than some speculative work evaluation of actions leading from an information set in advance would be usefull in order to get results faster even at the cost of doing some more work than is needed. This could be used in games where the depth of search is know and thus from some predefined height from lowest leaves of the game tree parallel evaluation of all subtrees of this height could be performed.

This bring us back into deeply study of information sets since it is an element that has not been used in any of methods in the survey in this work. Still, this topic remains open.

Appendix A

Graphs for expriments

Here are incidence matrices used in experiments with Border patrolling games.

0	1	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	0	1	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	0

Table 8 The incidence matrix for graph G1.

0	1	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0	0
0	0	0	0	1	1	1	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	0	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	0	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	0	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	0	0	0	0

Table 9 The incidence matrix for graph G2.

Bibliography

[1] Ateji.com. Ateji PX. `http://www.ateji.com/px/`, 2014. Accessed: 2014-2-1.

[2] Branislav Bosansky, Christopher Kiekintveld, Viliam Lisy, Jiri Cermak, and Michal Pechoucek. Double-oracle algorithm for computing an exact nash equilibrium in zero-sum extensive-form games. In *Proceedings of the 2013 international conference on Autonomous agents and multi-agent systems*, pages 335–342. International Foundation for Autonomous Agents and Multiagent Systems, 2013.

[3] Branislav Bosansky, Christopher Kiekintveld, Viliam Lisy, and Michal Pechoucek. *Solving Zero-Sum Extensive-Form Games with Sequence-Form Double-Oracle Algorithm*, 2012. article under review.

[4] Rainer Feldmann, Burkhard Monien, and Peter Mysliwietz. *Game tree search on a massively parallel system.* 1994.

[5] Ranier Feldmann, Peter Mysliwiete, and Burkhard Monien. Studying overheads in massively parallel min/max-tree evaluation. In *Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures*, pages 94–103. ACM, 1994.

[6] Thomas S. Ferguson. Game theory. `publishhttp://dutraeconomicus.files.wordpress.com/2014/02/game-theory-text-thomas-s-ferguson.pdf`, 2005. Accesed: 2014-5-5.

[7] Raphael A Finkel and John P Fishburn. Improved speedup bounds for parallel alpha-beta search. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, (1):89–92, 1983.

[8] John P Fishburn. *Analysis of speedup in distributed algorithms.* University Microfilms International (UMI), 1984.

[9] Guava. Guava. `http://code.google.com/p/guava-libraries/`, 2014. Accessed: 2014-2-1.

[10] http://functionaljava.org/. Functional java. `http://functionaljava.org/`, 2014. Accessed: 2014-02-1.

[11] Michael Johanson, Kevin Waugh, Michael Bowling, and Martin Zinkevich. Accelerating best response calculation in large extensive games. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence-Volume Volume One*, pages 258–265. AAAI Press, 2011.

[12] JPPF.org. Jppf. `http://www.jppf.org/`, 2014. Accessed: 2014-02-1.

[13] Richard M Karp and Yanjun Zhang. On parallel evaluation of game trees. In *Proceedings of the first annual ACM symposium on Parallel algorithms and architectures*, pages 409–420. ACM, 1989.

[14] Donald E Knuth and Ronald W Moore. An analysis of alpha-beta pruning. *Artificial intelligence*, 6(4):293–326, 1976.

[15] Rigie G Lamanosa, Kheiffer C Lim, Ivan Dominic T Manarang, Ria A Sagum, and Maria-Eriela G Vitug. Expectimax enhancement through parallel search for non-deterministic games. 2013.

[16] Lambdaj. Lambdaj. `http://code.google.com/p/lambdaj/`, 2014. Accessed: 2014-2-1.

[17] Valavan Manohararajah. *Parallel alpha-beta search on shared memory multiprocessors*. PhD thesis, University of Toronto, 2001.

[18] T Anthony Marsland and Murray Campbell. Parallel search of strongly ordered game trees. *ACM Computing Surveys (CSUR)*, 14(4):533–551, 1982.

[19] T Anthony Marsland and Fred Popowich. Parallel game-tree search. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, (4):442–452, 1985.

[20] TED NEWARD. Java 8: Lambdas. `http://www.oracle.com/technetwork/issue-archive/2013/index.html`, 2013. Accessed: 2013-11-1.

[21] TED NEWARD. Java 8: Lambdas. `http://www.oracle.com/technetwork/issue-archive/2013/index.html`, 2013. Accessed: 2013-11-1.

[22] Monroe Newborn. Unsynchronized iteratively deepening parallel alpha-beta search. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 10(5):687–694, 1988.

[23] Jonathan Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 11(11):1203–1212, 1989.

[24] Martin Schmid. Game theory and poker, 2013.

[25] Yoav Shoham and Kevin Leyton-Brown. *Multiagent systems: Algorithmic, game-theoretic, and logical foundations*. Cambridge University Press, 2009.

[26] Igor Steinberg and Marvin Solomon. *Searching game trees in parallel*. University of Wisconsin-Madison, Computer Sciences Department, 1989.