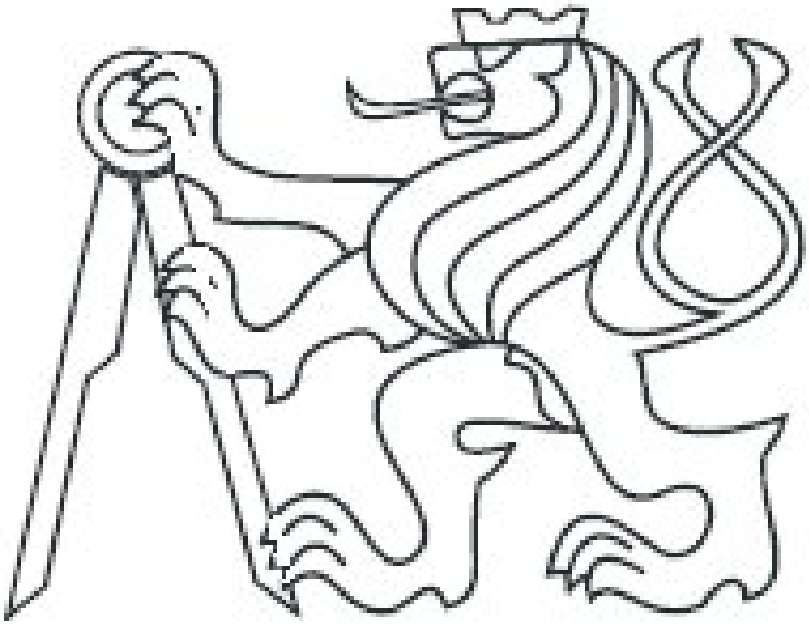
České vysoké učení technické

Fakulta elektrotechnická



DIPLOMOVÁ PRÁCE

Hraní obecných her s neúplnou informací

General Game Playing in Imperfect Information Games

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra kybernetiky

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: Bc. Tomáš M o t a l

Studijní program: Elektrotechnika a informatika (magisterský), strukturovaný

Obor: Kybernetika a měření , blok KM2 – Umělá inteligence

Název tématu: Hraní obecných her s neúplnou informací

Pokyny pro vypracování:

Hraní obecných her (GGP) se zabývá vytvářením agentů, kteří jsou na základě formální specifikace schopni hrát libovolnou, pro ně dopředu neznámou, hru.

1. Student se seznámí s algoritmy používanými pro hraní specifických her s neúplnou informací, přednostně her Poker a Kriegspiel (tj. Šachy s neúplnou informací).
2. Nastuduje hlavní principy, pravidla a požadavky soutěže v automatickém hraní obecných her (General Game Playing Competition), organizované při konferenci AAAI. Konkrétně nastuduje novou verzi jazyka na popis obecných her (GDL-II), která umožňuje definovat hry s neúplnou informací.
3. Shrne existující přístupy k hraní her s částečnou informací, popíše jejich požadavky a zanalyzuje jejich použitelnost při vytváření hráče obecných her s neúplnou informací.
4. Navrhne, implementuje a experimentálně ověří hráče pro hraní obecných her. Tento hráč nemusí plně podporovat GDL-II, ale měl by být schopen hrát více formálně specifikovaných her.

Seznam odborné literatury:

[1] Ciancarini, P., & Favini, G. P. (2010). Monte Carlo tree search in Kriegspiel. Artificial Intelligence, 174(11), 670-684. Elsevier

[2] Raboin, E., Nau, D., Kuter, U., Gupta, S. K., & Svec, P. (2010). Strategy Generation in Multi-Agent Imperfect-Information Pursuit Games Categories and Subject Descriptors. International Joint Conference on Autonomous Agents and Multiagent Systems (pp. 947-954).

[3] Zinkevich, M., Johanson, M., Bowling, M., & Piccione, C. (2008). Regret minimization in games with incomplete information. Advances in Neural Information Processing Systems, 20, 1729–1736.

[4] Genesereth, M., Love, N., & Pell, B. (2005). General game playing: Overview of the AAAI competition. AI Magazine, 26(2), 62. http://www.aaai.org/ojs/index.php/aimagazine/article/viewArticle/1813

[5] Thielscher, M. (2010). A General Game Description Language for Incomplete Information Games. AAAI 2010. http://cgi.cse.unsw.edu.au/~mit/Papers/AAAI10a.pdf

Vedoucí diplomové práce: Mgr. Viliam Lisý, M.Sc.

Platnost zadání: do konce letního semestru 2011/2012

L.S.

prof. Ing. Vladimír Mařík, DrSc.
vedoucí katedry

prof. Ing. Boris Šimák, CSc.
děkan

V Praze dne 9. 2. 2011

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

DIPLOMA THESIS ASSIGNMENT

Student: Bc. Tomáš Motal

Study programme: Electrical Engineering and Information Technology

Specialisation: Cybernetics and Measurement – Artificial Intelligence

Title of Diploma Thesis General Game Playing in Imperfect Information Games

Guidelines:

General Game Playing (GGP) deals with creating agents capable of playing any previously unknown game only based on its formal specification.

1. The student will study the state-of-the-art techniques for playing specific partial information games, mainly Poker and Kriegspiel (i.e. Chess with imperfect information).
2. He will get familiar with the rules, requirements and main principles of the General Game Playing competition organized at the AAAI conference. In particular, he will study the new variant of Game Description Language (GDL-II), which allows specifying imperfect information games.
3. He will review the existing approaches to playing imperfect information games, analyze their requirements and discuss their usability in GGP agent.
4. He will design, implement and experimentally evaluate a GGP agent. The agent does not have to fully support GDL-II, but it should be able to play multiple formally specified imperfect information games.

Bibliography/Sources:

[1] Ciancarini, P., & Favini, G. P. (2010). Monte Carlo tree search in Kriegspiel. Artificial Intelligence, 174(11), 670-684. Elsevier
[2] Raboin, E., Nau, D., Kuter, U., Gupta, S. K., & Svec, P. (2010). Strategy Generation in Multi-Agent Imperfect-Information Pursuit Games Categories and Subject Descriptors. International Joint Conference on Autonomous Agents and Multiagent Systems (pp. 947-954).
[3] Zinkevich, M., Johanson, M., Bowling, M., & Piccione, C. (2008). Regret minimization in games with incomplete information. Advances in Neural Information Processing Systems, 20, 1729–1736.
[4] Genesereth, M., Love, N., & Pell, B. (2005). General game playing: Overview of the AAAI competition. AI Magazine, 26(2), 62. http://www.aaai.org/ojs/index.php/aimagazine/article/viewArticle/1813
[5] Thielscher, M. (2010). A General Game Description Language for Incomplete Information Games. AAAI 2010. http://cgi.cse.unsw.edu.au/~mit/Papers/AAAI10a.pdf

Diploma Thesis Supervisor: Mgr. Viliam Lisý, M.Sc.

Valid until: the end of the summer semester of academic year 2011/2012

prof. Ing. Vladimír Mařík, DrSc.
Head of Department

ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE
Fakulta elektrotechnická
L.S.

prof. Ing. Boris Šimák, CSc.
Dean

Prague, February 9, 2011

Prohlášení
Prohlašuji, že jsem svou diplomovou práci vypracoval samostatně a použil jsem pouze podklady (literaturu, projekty, SW atd.) uvedené v přiloženém seznamu.

V Praze, dne 7.5.2011 …….

podpis

Acknowledgements

Here I would like to thank my advisor Mgr. Viliam Lisý, MSc. for his time and valuable advice which was a great help in completing this work.
I would also like to thank my family for their unflinching support during my studies.

Tomáš Motal

Abstrakt

Název: Hraní obecných her s neúplnou informací

Autor: Tomáš Motal
email: tomasmotal@yahoo.com

Oddělení: Katedra kybernetiky
Fakulta elektrotechnická, České vysoké učení technické v Praze
Technická 2
166 27 Praha 6
Česká Republika

Vedoucí práce: Mgr. Viliam Lisý, MSc.
email: lisy@agents.felk.cvut.cz

Oponent: RNDr. Jan Hric
email: Jan.Hric@mff.cuni.cz

Abstrakt

Cílem hraní obecných her je vytvořit inteligentní agenty, kteří budou schopni hrát konceptuálně odlišné hry pouze na základě zadaných pravidel. V této práci jsme se zaměřili na hraní obecných her s neúplnou informací. Neúplná informace s sebou přináší nové výzvy, které je nutno řešit. Například hráč musí být schopen se vypořádat s prvkem náhody ve hře či s tím, že neví přesně, ve kterém stavu světa se právě nachází.
Hlavním přínosem této práce je TIIGR, jeden z prvních obecných hráčů her s neúplnou informací který plně podporuje jazyk pro psaní her s neúplnou informací GDL-II. Pro usuzování o hře tento hráč využívá metodu založenou na simulacích. Přesněji, využívá metodu Monte Carlo se statistickým vzorkováním.
Dále zde popíšeme jazyk GDL-II a na námi navržené hře piškvorek s neúplnou informací ukážeme, jak se v tomto jazyce dají tvořit hry.
Schopnost našeho hráče hrát konceptuálně odlišné hry i jeho výkonnost je experimentálně ověřena při hraní několika různých her (karty, piškvorky s neúplnou informací, Macháček).

Klíčová slova

Hraní obecných her, Hry s neúplnou informací, Monte Carlo

Abstract

Title:	General Game Playing in Imperfect Information Games
Author:	Tomáš Motal email: tomasmotal@yahoo.com
Department:	Department of Cybernetics Faculty of Electrical Engineering, Czech Technical University in Prague Technická 2 166 27 Prague 6 Czech Republic
Advisor:	Mgr. Viliam Lisý, MSc. email: lisy@agents.felk.cvut.cz
Opponent:	RNDr. Jan Hric email: Jan.Hric@mff.cuni.cz

Abstract

The goal of General Game Playing (GGP) is to create intelligent agents that are able to play any game based only on the description of the games rules. In this thesis we focus on GGP for games with imperfect information. Compared with perfect information games the imperfect information games bring with them numerous new challenges that must be tackled, for example: Player must find a way how to work with the uncertainty about her current state of the world, or cope with randomness in the game.

The main outcome of this thesis is TIIGR, one of the first GGP agents for games with imperfect information. Our agent uses a simulation-based approach to action selection. Specifically it uses perfect information sampling Monte Carlo with Upper Confidence Bound applied to Trees as its main reasoning system.

Further we explain the Game Description Language-II (GDL-II) and on a game we created we explain how to create games in this language.

The ability of our player to play conceptually different games and her performance was verified on several games including Latent Tic-Tac-Toe, Liar's dice and cards.

Keywords

General Game Playing, Imperfect Information Games, Monte Carlo

Contents

List of acronyms

CFR – counterfactual regret

EFG – extensive form game

GDL – game description language

GGP – general game playing

GS – game server

IIG – imperfect information game

PIG – perfect information game

PIMC – perfect information sampling Monte Carlo

MC – Monte Carlo

MCTS – Monte Carlo tree search

TIIGR – name of our imperfect information game player

UCT – upper Confidence Bound applied to Trees

1 Introduction

Games accompanied men since the beginning of time. We all played some games during our lives, be it card, board or computer games. A lot of researchers focus on the problems connected with game playing one of them being the creation of an intelligent agent that would be able to play against humans and best them in their own games.

At first, researchers focused on games with perfect information. Probably the most discussed games at that time were Chess and Go. After several decades of research a computer called Deep Blue emerged. It was the first computer that was able to defeat human world champion Garry Kasparov. With some exaggeration we can say that overcoming this challenge allowed people to start focusing on other areas. After Deep Blue came Chinook. Another program that was able to beat humans this time in the game of Checkers. And so on. It is only logical that our attention has shifted towards imperfect information games.

Games with imperfect information can offer more than their predecessors could. They can conveniently model real-life strategic interactions among multiple agents (including uncertainty and imperfect information), be it in economy, military, business process management, etc. En excellent example of a game that was used in reality to model military behavior and train military officers is the game Kriegspiel (originally named *Instructions for the Representation of Tactical Maneuvers under the Guise of a Wargame)* first invented by a Prussian officer Georg von Rassewitz in the first years of the 19th century. It was used to train officers in tactical maneuvers in Prussian army, and later adopted by many other countries. Kriegspiel was applied during the Russo-Japanese war by Japanese navy which resulted in Japan's unexpected victory. Other examples of games with imperfect information are modern computer games used for combat simulation.

As we can see there are numerous types of games used today for simulating reality. Unfortunately, there are not many solvers that would be able to cope with the extremely large state spaces typical for these games. And there are even less domain independent solvers that would be able to play these games without domain-specific knowledge hard coded in advance. In this work we aim to create a program that is able to solve conceptually different games only with the most essential knowledge – the rules of the game.

Our contributions include:

- Creating an overview of approaches used for solving imperfect information games
- Analyzing counterfactual regret, perfect information sampling Monte Carlo and Information set search
- Creating a Latent Tic-Tac-Toe game in GDL-II

- Implementation of one of the first general game players fully compatible with GDL-II which is able to play any altering move game defined in GDL-II.

1.1. Thesis outline

In this Section we present the outline of our thesis which should give you the basic idea what to expect.

In Chapter 2 we start with introducing basic concepts of game theory, such as what is a game, how does the extensive form of a game look like, etc. and some basic solution concepts for games.

In the following Chapter 3 we go through several algorithms that can be used for creating a game player such as Minimax, Monte Carlo methods, Counterfactual regret, etc.

Then in Chapter 4 we discuss general game players and the general game playing competition that has been started to promote research in this area. After that we discuss the syntax of game description language (GDL) that is now the standard for describing perfect information games (PIG), and GDL-II that is used for describing imperfect information games (IIG). In the last section of this chapter we show how to create a game in GDL-II on a game of Latent Tic-Tac-Toe that we created.

In Chapter 5 we set our requirements for our general game player and discuss the different approaches and their advantages and disadvantages. Based on our discussion we selected perfect information sampling Monte Carlo for our general game player TIIGR.

Following the discussion of existing methods and their advantages and disadvantages we discuss the implementation details of our TIIGR player in Chapter 6.

Chapter 7 presents several experiments that we performed with our imperfect information player. With these experiments we prove that our player is capable of playing conceptually different games. From these experiments we deduce which parameters influence our player's performance.

Last but not least, in Chapter 8 we evaluate our work, revisit our goals and we offer several ways in which our player can be improved thus setting new goals for future work.

2 Extensive form games

This Chapter introduces the concept of Extensive form game. We begin in Section 2.1 with defining what a game means. In Section 2.2 we explain all key concepts in games (such as extensive form game, information sets, etc.) and we define the terminology that we use further in the text. Later we discuss several important types of games – zero-sum game, game of perfect recall, etc. In Section 2.3 we introduce one of the best known solution concepts for extensive form games - Nash equilibrium and Ɛ-Nash equilibrium.

2.1. Game

Every one of us has some idea of what a game is. But let's specify what every game consists of:

1. **Player** – A person or an agent who participates in the game and determines the actions in the game. In imperfect information games chance (dice rolling, etc.) is considered one of the players.
2. **Action** – A move that a player can make during the game.
3. **Utility (payoff)** – A reward that a player obtains after the game has ended. Utility depends on how all the players played during the game (on all players' actions).

It is important to state that in this thesis all games we discuss are implicitly considered sequential games (if not stated otherwise). A sequential game is a game where a player takes an action only in a specified moment that is defined by the order of players (e.g.: player 2 takes an action only after player 1 has taken hers). We call this moment a **turn**. This is opposed to simultaneous move games where all players choose their actions without first seeing what the other players have played.

Games can be divided into 2 classes based on the information players have available: games with perfect information and games with imperfect information. In perfect information games every player knows the whole state of the game world (e.g.: position of all pieces, cards dealt to other players, etc.). In imperfect information games player has only a limited knowledge concerning the world (e.g.: poker – player only knows cards dealt to her but not to the other players, etc.). Now we need to somehow represent the game. There are several different ways how this can be done. The game representation we use throughout this thesis is Extensive Game Form that we will now describe.

2.2. Extensive form Game

Extensive form is one of the possibilities how to represent a game. There are other forms that can be used, such as normal form (also called the strategic form), etc., but we will not discuss them in this thesis since we only use the extensive form. An extensive form game is

represented in a tree form (there are no cycles in a tree thus there are no cycles in an extensive form game). A game tree consists of nodes and edges. Nodes represent game states and edges represent actions available at the current state. Connected with every edge is a label with the action's name. The formal definition of an extensive form game for a game of imperfect information is as follows (Osborne & Rubinstein, 1994):

Definition 1 (Extensive Form): *A finite extensive form game with imperfect information has the following components:*

- *A finite set N of players.*
- *A finite set H of sequences, the possible **histories** of actions, such that the empty sequence is in H and every prefix of a sequence in H is also in H. $Z \subseteq H$ are the **terminal histories**. No sequence in Z is a strict prefix of any sequence in H. $A(h) = \{a: (h, a) \in H\}$ are the actions available after a non-terminal history $h \in H \backslash Z$.*
- *A **player function** P that assigns to each non-terminal history a member of $N \cup \{c\}$, where c represents chance. $P(h)$ is the player who takes an action after the history h. If $P(h) = c$, then chance determines the action taken after history h. Let H_i be the set of histories where player i chooses the next action.*
- *A function f_c that associates with every history h for which $P(h) = c$ a probability measure $f_c(\cdot|h)$ on $A(h)$. $f_c(a|h)$ is the probability that a occurs given h, where each such probability measure is independent of every other such measure.*
- *For each player $i \in N$, a partition $\mathcal{I}_i$ of H_i with the property that $A(h) = A(h')$ whenever h and h' are in the same member of the partition. $\mathcal{I}_i$ is the **information partition** of player i; a set $I_i \in \mathcal{I}_i$ is an **information set** of player i.*
- *For each player $i \in N$, a **utility function** u_i that assigns each terminal history a real value. $u_i(z)$ is rewarded to player i for reaching terminal history z.*

Let us explain the extensive form on a slightly modified Tic-Tac-Toe game. We consider a Tic-Tac-Toe game with imperfect information (Latent Tic-Tac-Toe). Compared to classic Tic-Tac-Toe game there are several changes. First, each player sees only her own marks on the board. Second, when a player takes an action (tries to mark an empty space) there are 2 possible outcomes:

1. The mark was made (the same as in perfect information game)
2. The mark could not be made (in the case there already is a different mark on the same tile).

In our Latent Tic-Tac-Toe example the game begins with an empty board. The empty board is the root (initial state) of the game tree. Because players take turns each ply of the game tree consists of nodes where only 1 player can select an action.

Now we need to define several concepts (most of them have been defined in the definition of extensive form game but we will try to use a less formal description for some of the

concepts, and introduce some observations which can be made from these definitions). For that we will use Figure 1 which shows the first 3 plies of an imperfect information Tic-Tac-Toe game in extensive game form.

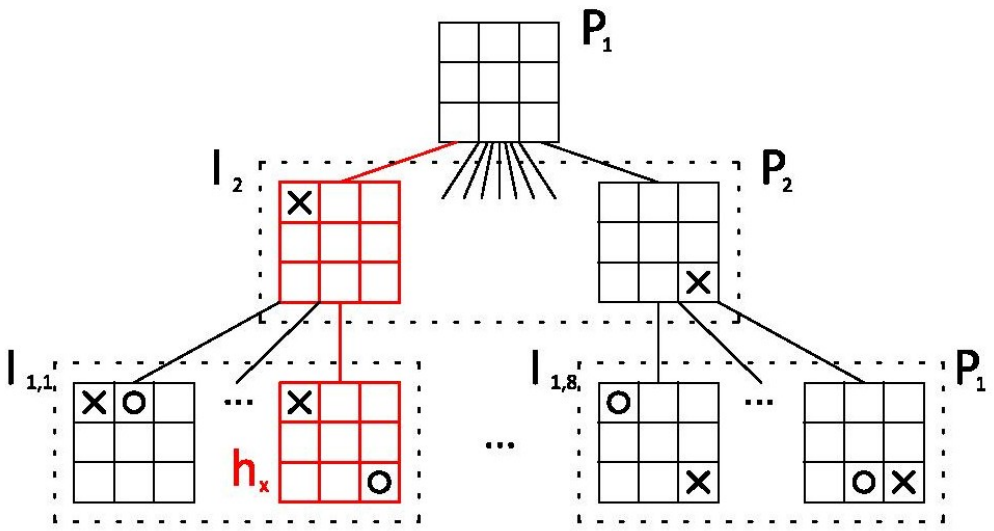


Figure 1: 3 plies of Tic-Tac-Toe game with several different concepts. Each ply has its own player (P_1, P_2). Each of these players has several information sets (all the nodes encircled by one dotted line represent 1 information set). The red line throughout the nodes is a history h_x.

The idea of history was already formally defined in Definition 1 but because it is a key term which we use throughout the thesis let's try to make the definition a little less formal. **History *h*** is a sequence of all player's actions. It is used for representing a node in the game tree as well. Because history is a sequence of actions of all players, it provides a path starting from root through the game tree which ends in one specific node. On Figure 1 the history h_x (red line) represents both the sequence of actions taken from the root node of the game to the last node as well as it represents the last node with one X and one O.

Definition 2 (Information set): "*Player P_i's **information set** I_i at any particular point of the game is a set of different nodes in the game tree that she knows might be the actual node, but between which she cannot distinguish by direct observation.*" (Rasmusen, 2006)

Simply said, information set I is a set of states between which a player cannot distinguish. To show an example let's consider the 2nd ply where it is player 2's turn. During the previous turn player 1 placed her cross somewhere on the game board but player 2 does not know where. Therefore all the possible boards with only 1 cross placed are in single player 2's information set because she cannot distinguish in which state she is. There are several observations we can make from the definition:

a) All the nodes in I_i are nodes where player P_i makes a decision.
b) All nodes in 1 information set must have the same actions available. If they have different actions available then the player would be able to distinguish between

them. Examples of sets that can be incorrectly considered an information set are shown on Figure 2.

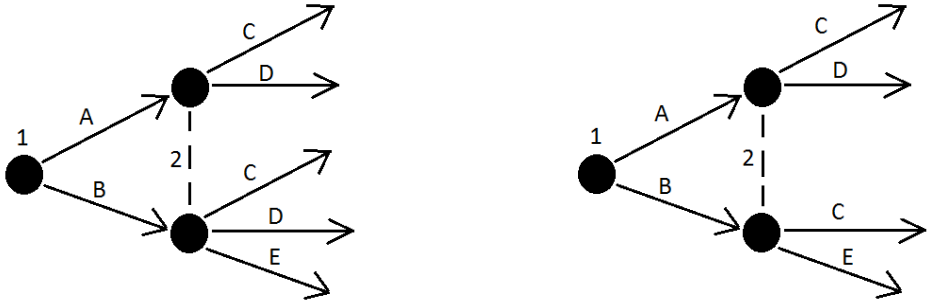


Figure 2: Example of 2 incorrectly defined information sets. Information set is represented by the dotted line. The number next to the dotted line is the number of the player whose information set that is. Names of actions are written above the transition arrows. Player 2 cannot observe the action that player 1 did thus she cannot distinguish between the 2 states. (Left) In this case player 2 can distinguish between the states because from the states lead different number of actions. (Right) In this situation player 2 can distinguish between the 2 states because the actions leading from the states are different.

It is easy to perceive that if all information sets are singletons (each information set contains only one node) then we have a game of perfect information.

A **zero-sum game** is a game where the sum of utilities for all players in every terminal node equals zero.

It is easy to show that any game can be transformed into a zero-sum game. Let's take a general game with n players where we have a terminal node with utilities $u_1 = a, u_2 = b, \dots, u_n = z$ where $\sum_{i=1}^{n} u_i \neq 0$. Then by a simple trick of adding an imaginary $n + 1^{\text{th}}$ player with the utility $u_{n+1} = -\sum_{i=1}^{n} u_i$ we have created a zero-sum game.

In this thesis we are going to focus on games of perfect recall. The following definition is taken from (Shoham & Leyton-Brown, 2010).

Definition 3 (Perfect recall): *Player i has **perfect recall** in an imperfect-information game G if for any two nodes h, h' that are in the same information set for player i, for any path h_0 , a_0 , h_1, a_1, h_2 , . . . , h_m , a_m , h from the root of the game to h (where the h_j are decision nodes and the a_j are actions) and for any path h_0, a_0', h_1', a_1', h_2', . . . , h_m', a_m', h' from the root to h' it must be the case that:*

1. $m = m'$;
2. *for all* $0 \leq j \leq m$, *if* $\rho(h_j) = i$ *(i.e., h_j is a decision node of player i), then h_j and h_j' are in the same equivalence class for i; and*
3. *for all* $0 \leq j \leq m$, *if* $\rho(h_j) = i$ *(i.e., h_j is a decision node of player i), then $a_j = a_j'$.*

G is a game of perfect recall if every player has perfect recall in it.

From the above definition a game of **perfect recall** is a game where all players remember the whole history that has happened (that means all the actions taken by her and by the

opponent before ending in the current state). This means that even though the current state might be identical, in perfect recall games the path through which a player reached the state also defines the current state. Therefore a state that looks exactly the same might not be considered the same state in games of perfect recall. On Figure 3 there is an example of such a case. The final state is the same in both cases but the path leading to the state is different, thus it is not the same state.

A game is of **imperfect recall** if the above definition does not hold (if it is not of perfect recall).

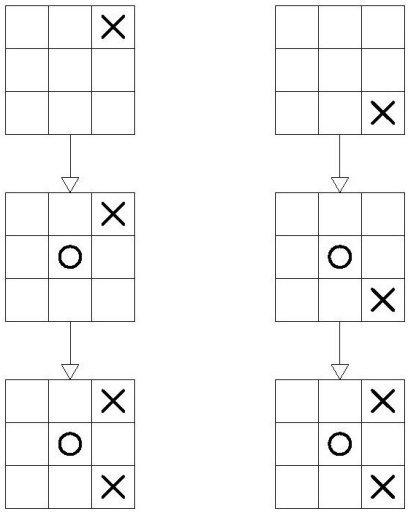


Figure 3: Example of a different state in perfect recall games.

A **strategy** σ_i of player P_i in perfect information game can be viewed as an instruction sheet that tells the player $\mathrm{P_i}$ which action to take in each state of the game. In an imperfect information game it tells player $\mathrm{P_i}$ what to do in each information set. Thus a strategy is a function of information set and we denote it $\sigma_\mathrm{i}(\mathrm{I})$, where I is the current information set the player $\mathrm{P_i}$ is in (in PIG information set I is a singleton and thus equals only to 1 game state). To keep the equations simple, whenever we write σ_i further in the text we mean $\sigma_\mathrm{i}(\mathrm{I})$. We have 2 types of strategies:

1. **Pure strategy** – is a deterministic strategy that specifies for each state exactly 1 action that a player should take.
2. **Mixed strategy** – is a probability distribution over all pure strategies

Below are the formal definitions of strategy, strategy set and strategy profile as defined in (Rasmusen, 2006).

Definition 4 (Strategy): *Player i's **strategy** $\sigma_i(I)$ is a rule that tells her which action to choose at each instant of the game, given hers information set.*

Definition 5 (Strategy set): *Player i's **strategy set** or **strategy space** $S_i = \{\sigma_i\}$ is the set of strategies available to her.*

Definition 6 (Strategy profile): *A **strategy profile** $\sigma = (\sigma_1, \sigma_2, \dots, \sigma_n)$ is an ordered set consisting of one strategy for each of the n players in the game.*

Further in the text $\sigma_{-i} = (\sigma_1, \dots, \sigma_{i-1}, \sigma_{i+1}, \dots, \sigma_n)$ refers to a strategy profile without player's i strategy σ_i.

2.3. Solution concepts

In the previous section we have defined all important concepts and ideas centered on extensive form game. Let us now focus on how we can solve games. Algorithms and specific approaches are covered in Chapter 3, here we define and discuss the general idea.

Let's have a look on probably the best known and one of the fundamental solution concepts in game theory: Nash Equilibrium. But first, we need to define the idea of best response that will be later used in the definition of Nash equilibrium.

Definition 7 (Best response): *Player i's **best response** to the strategy profile σ_{-i} is a mixed strategy $\sigma_i^* \in S_i$ such that $u_i(\sigma_i^*, \sigma_{-i}) \geq u_i(\sigma_i, \sigma_{-i})$ for all strategies $\sigma_i \in S_i$.* (Shoham & Leyton-Brown, 2010)

Now we can finally define the Nash Equilibrium.

Definition 8 (Nash Equilibrium): *A strategy profile $\sigma_{NE} = (\sigma_1, \sigma_2, \dots, \sigma_n)$ is a **Nash equilibrium** if, for all agents i, σ_i is a best response to σ_{-i}.* (Shoham & Leyton-Brown, 2010)

From **Definition 7** and **Definition 8** we can see that Nash equilibrium is a strategy profile σ_{NE} from which none of the players has a reason to deviate. At this point we can provide one more definition of Nash equilibrium with the use of regret. Regret is a concept that tells us how much a player loses when he plays a specific move m_i in response to opponent's move m_{-i}. In other words, how much she regrets playing move m_i instead of playing the best response to opponent's move m_{-i}. A more detailed description of regret and an algorithm based on regret can be found in Section 3.2. This is done because later we discuss regret and counterfactual regret so that we have a better understanding of the connection between Nash Equilibrium and regret.

Definition 9 (Nash Equilibrium): *A strategy profile $\sigma_{NE} = (\sigma_1, \sigma_2, \dots, \sigma_n)$ is a **Nash equilibrium** if for all players the value of regret is zero.*

Definition 8 and Definition 9 state the exact same thing – that players have no reason to deviate from Nash equilibrium. If a player does not want to deviate from some plan of actions then the player does not regret taking those actions. Thus each player's regret must equal to 0.

Any one player does not deviate because no player can increase their utility by abandoning σ_{NE} , while holding the strategies of other players fixed. Let's illustrate this on an example shown on Figure 4. This is a Prisoner's Dilemma game (Russell & Norvig, 2003). Each player has 2 possible actions: Testify (T) and Defect (D). Rewards of the game are defined in Figure 4. We can see that the Nash equilibrium in this game is (D, D). Why is it so? Well, if player 1 selects to Defect from this strategy while we fix player 2's choice, she would move to the history (T, D) with the utility $(0{,}10)$. Thus she would not gain anything, but quite contrary she would lose her reward of 2. The same goes for player 2. Thus no player has the tendency to deviate from their decision. If we look on all the other histories, none of them has this property.

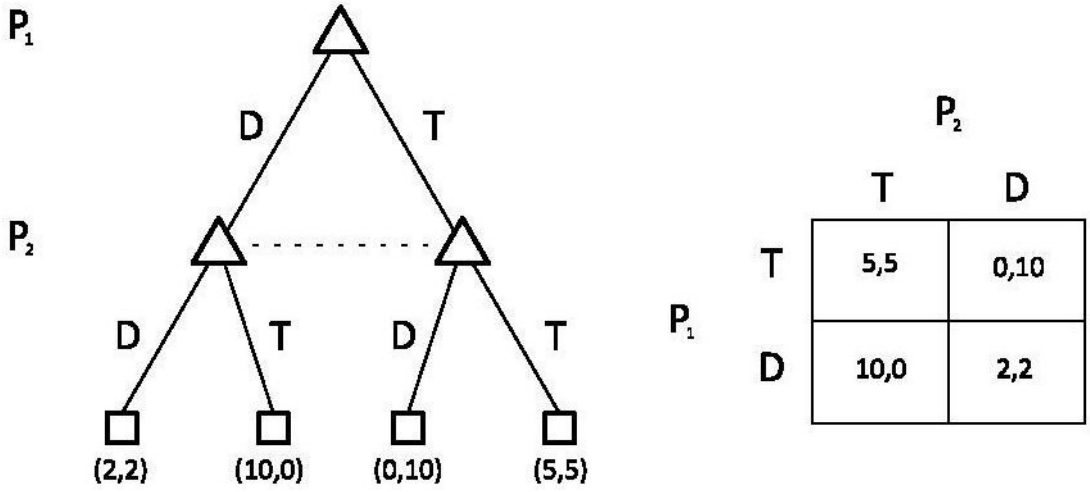


Figure 4: (Left) Extensive form and (Right) and normal form of Prisoner's dilemma. We have not introduced the normal game form but for our purposes it is enough to say that inside of the matrix are the utilities of players, each row is an action available to player 1 and each column is an action available to player 2.

A Player using Nash Equilibrium strategy plays the best response against their opponent. There are situations when players might not want to change their strategies if the utility they gain from switching to Nash Equilibrium is smaller than some value Ɛ. This solution concept is called Ɛ-Nash Equilibrium and is defined below:

Definition 10 (Ɛ-Nash Equilibrium): *Fix* $\varepsilon > 0$. *A strategy profile* $\sigma = (\sigma_1, \sigma_2, \dots, \sigma_n)$ *is an* Ɛ-*Nash equilibrium if, for all agents* i *and for all strategies* $s_i' \neq s_i$, $u_i(s_i, s_{-i}) \geq u_i(s_i', s_{-i}) - \varepsilon$ (Shoham & Leyton-Brown, 2010).

This Chapter covered the concept of a game and one of its formal models – extensive game form. Later in this thesis we consider our games to be always in extensive form because some algorithms can be easily explained on this form. Then we have defined several concepts that are closely bound with games – information sets, zero-sum games, perfect recall, strategy, history, etc. Later in this work we use these concepts (especially information sets, strategy, history, etc.) to define algorithms for solving imperfect information games. In the end we introduced one of the most widely known solution concepts – Nash Equilibrium

and Ɛ-Nash Equilibrium. Nash Equilibrium is especially important because if we are able to prove that our algorithm converges to a Nash Equilibrium, it means that our player plays optimally against a rational opponent. In the following Chapter we explain several general game playing algorithms and concepts that can be used to implement a general game player.

3 General game playing algorithms

This Chapter lists some of the existing algorithms that can be used for implementing general game players – agents that can play conceptually different games without having any game-specific knowledge hard coded in advance. We will make a brief stop at each algorithm, explain it and point out its advantages and disadvantages. In Section 3.1 we discuss the Minimax approach and in the following Section 3.2 we cover the idea of regret. Section 3.3 provides an introduction to Monte Carlo methods. In the following Section 3.4 we extend the Monte Carlo methods and explain, how Monte Carlo tree search can be applied on games with imperfect information. In Section 3.5 we look into a new idea called counterfactual regret minimization. Last but not least, we explain the Information Set search technique in Section 3.6.

Throughout this Chapter all games we discuss are altering moves games – each player plays in a defined order (as opposed to simultaneous games where players play without the immediate knowledge of their opponents moves – this simulates that the players are making their moves at the same time).

3.1. Minimax

Minimax concept is based on the assumption that your opponent is going to try and minimize your gain as much as possible. A logical idea is to try and maximize your gain in the worst-case scenario and that is what Minimax algorithm does. The algorithm uses values to estimate each state in the game tree. These values are called **Minimax values** and are defined as (Russell & Norvig, 2003):

$$Minimax - value(node) =$$

$$\begin{cases} utility(node) & if\ n\ is\ a\ terminal\ state \\ max_{s\ \in\ Successors(node)}\ Minimax - value(s) & if\ n\ is\ a\ MAX\ node \\ min_{s\ \in\ Successors(node)}\ Minimax - value(s) & if\ n\ is\ a\ MIN\ node \end{cases} \quad (1)$$

The algorithm traverses the game tree depth-first and searches for leaves. From the leaves we obtain the utility for all players (in this case they are equal to Minimax value) and we calculate the Minimax values of their parent nodes. Minimax, how it is described here, is applicable on a 2-player, zero-sum game. In its paranoid version (all players try to minimize Max's utility) Minimax can be applied even to n-player games. It is a custom to call one of the players *Max* (this one tries to maximize her utility) and the other player *Min* (she tries to minimize *Max*'s utility). Let's show the Minimax algorithm on an example. On Figure 5 we present Minimax algorithm on 3 plies of a 2 player game. States where player *max* makes a choice ("MAX nodes") are represented by upward pointing triangle, and *Min* player's nodes ("MIN nodes") are the downward pointing triangle.

With the Minimax value definition and the example on Figure 5 it should be pretty straightforward to see how does the Minimax algorithm work. A nice description of Minimax algorithm can be found in (Russell & Norvig, 2003) or a more formal description in (Shoham & Leyton-Brown, 2010).

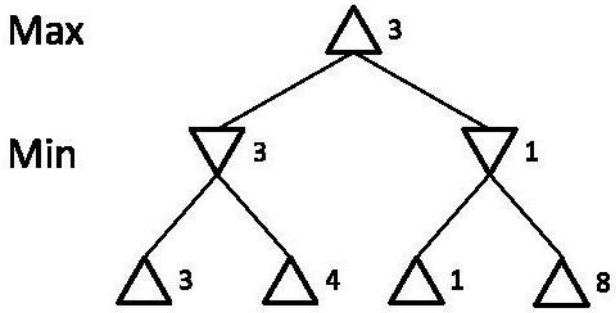


Figure 5: Minimax game tree. Upward pointing triangles represent states where Max makes her decision, downward pointing triangle states where Min makes her decision.

Just from the basic description above it is clear that such approach cannot be plausible for large games and of course it is not (because of the huge time required to traverse the whole tree and calculate Minimax values in the whole game tree). There are ways how to decrease the time requirements. One of them is alpha-beta pruning which is described in (Russell & Norvig, 2003).

Another approach is to limit the depth to which we traverse the tree in search of leaves. We prune all the levels of the tree below the depth. Now we have a smaller tree that we can traverse completely. However, the leaf nodes of this new tree might not be terminal states and thus there is no utility that we can use to compute the Minimax values for the whole tree. To cope with this problem we need to apply a heuristic evaluation function that tells us how good/bad the states are.

However, even with the use of the above mentioned approaches the use of Minimax is fairly limited for large games.

Minimax can be easily extended to games with more than 2 players. The extension was first made by (Luckhardt & Irani, 1986) and is called $\boldsymbol{Max^n}$. In Max^n all players try to maximize their own utility. Compared to Minimax utility that was represented just by one number, in an n-player game it is represented by an n-tuple $(u_1, \dots, u_n)$. Therefore, instead of propagating just one number from the leaves we will now propagate an n-tuple as shown on Figure 6.

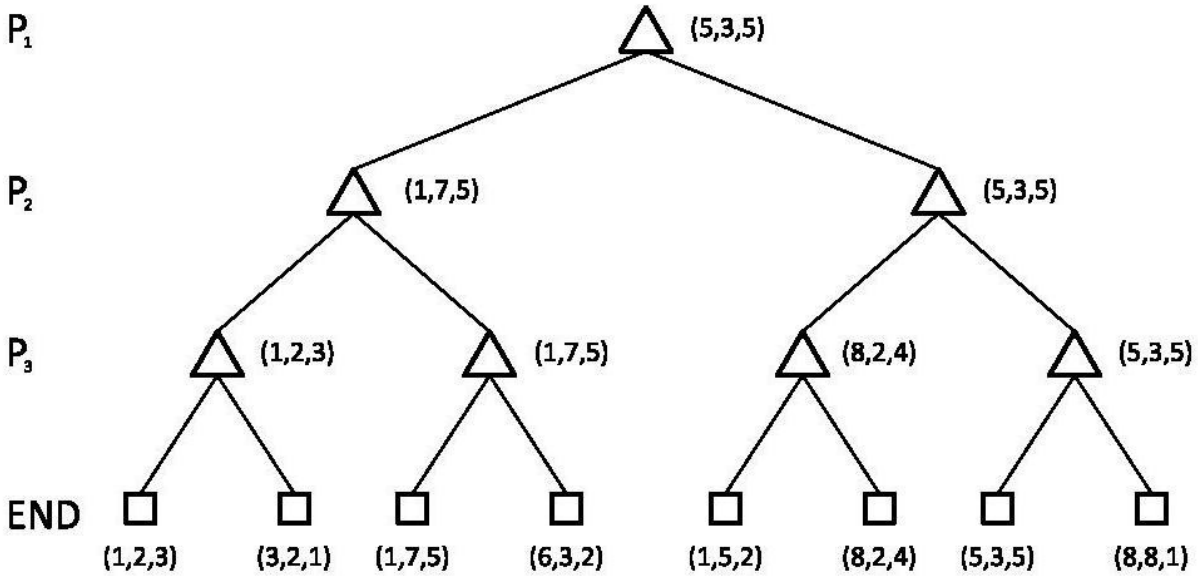


Figure 6: Max^n game tree for a 3 player game.

3.2. Regret minimization

In the section 3.1 we have introduced the Minimax concept. But there are situations when we are not playing against an opponent who wants always to minimize our gain or the opponent is not able to play to minimize our gain (due to lack of skills or knowledge). In those situations Minimax does not always give us optimal results. In this Section, all the definitions are taken from (Shoham & Leyton-Brown, 2010). Let us introduce the idea of regret (in the definitions below action profile is the same thing as strategy profile).

Definition 11 (Regret): *Player i's **regret** for playing an action a_i if the other players adopt action profile a_{-i} is defined as*

$$regret_i = \left[max_{a'_i \in A_i} u_i(a'_i, a_{-i})\right] - u_i(a_i, a_{-i}), \tag{2}$$

Where u_i is the utility of player i, a'_i is the action player i could have taken and A_i is the set of all actions available to player i. a_{-i} is the action profile containing all actions except player i's action.

The idea of regret is how much we regret not taking action a'_i instead of taking action a_i.

Definition 12 (Minimax regret): ***Minimax regret** actions for player i are defined as*

$$argmin_{a_i \in A_i}\left[max_{a_{-i} \in A_{-i}}(regret_i)\right] = \\ argmin_{a_i \in A_i}\left[max_{a_{-i} \in A_{-i}}\left(\left[max_{a'_i \in A_i} u_i(a'_i, a_{-i})\right] - u_i(a_i, a_{-i})\right)\right] \tag{3}$$

We can compare the results of Minimax and Minimax regret on an example. Let us consider a game with the following game tree shown on Figure 7.

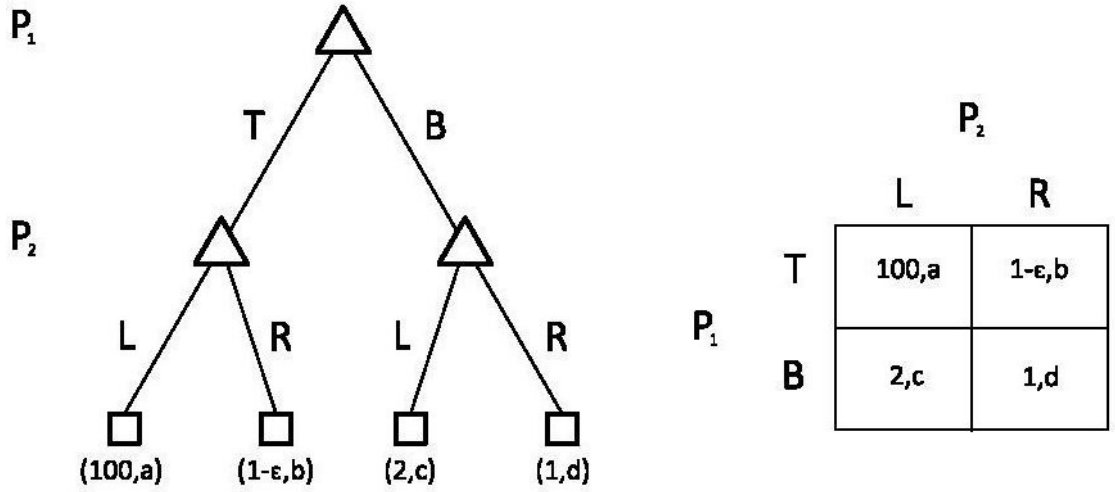


Figure 7: (Left) Extensive and (Right) normal form of game for comparing Minimax and Minimax regret. a, b, c, d are random numbers and ε is a small positive number. We have not introduced the normal game form but for our purposes it is enough to say that inside of the matrix are the utilities of players, each row is an action available to player 1 and each column is an action available to player 2

Player 1 can select between actions T or B. If she plays by the Minimax strategy, then she will select action B. This is easily deduced if we look on the rows of the normal form representation on Figure 7 and on the utility for player 1. If player 1 selects action T and player 2 selects action L, then player 1 receives utility of 100. But if player 2 selects action R, then player 1 receives utility $1-\varepsilon$. We do the same for the second row and for actions B and R we obtain the utility 1. If player 1 plays the Minimax strategy, then in the worst-case scenario she receives 1 which is greater than $1-\varepsilon$.

		2	
		L	R
1	T	$100-100 = 0$	$1-\epsilon+1 = \epsilon$
	B	$100-2 = 98$	$1-1 = 0$

Figure 8: Regret for player 1

But what if player 2 plays action L? Then by playing the Minimax strategy player 1 receives utility 2 instead of 100. Let's apply the Minimax regret concept we defined earlier. On Figure 8 we calculated the regret for player 1. We subtracted from the current value the best value in each column (e.g.: For actions B,L the best value in the column is 100 therefore the regret of playing action B is 100 - 2 = 98. We regret not playing T by 98).

We can see that in the case where we are not playing an adversarial opponent or an opponent that is unable to play to minimize our gain the concept of regret minimization gives us more optimal results than the Minimax concept.

3.3. Monte Carlo Methods

Monte Carlo methods were first introduced by von Neumann and Ulam during the World War II. Generally, Monte Carlo method is not a specific method but more a technique. This technique depends on a large number of simulations and statistical analysis from which it aims to infer the correct answer.

We focus on Monte Carlo tree search (MCTS) method (that can be nicely applied to extensive form games). It is a best-first search method that can be divided into 4 parts as shown on Figure 9. These parts are:

I. Selection
II. Expansion
III. Simulation / Playout
IV. Backpropagation

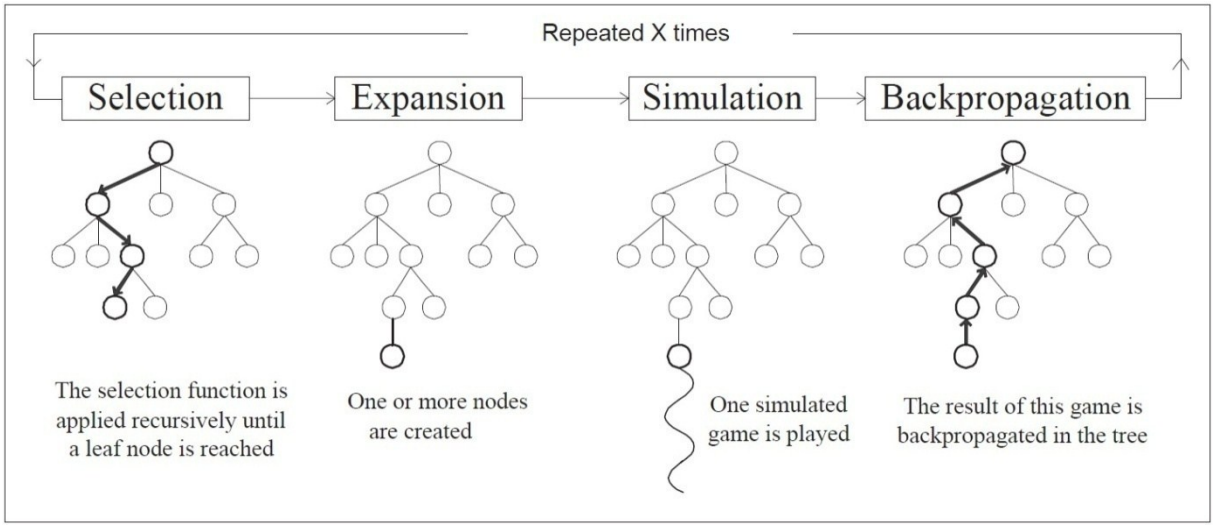


Figure 9 (Chaslot, Bakkes, Szita, & Spronck, 2008): Monte Carlo Tree search control loop

During the MCTS we build a tree that we are going to call simulation tree to distinguish it from extensive form game tree. Let's look on each section of MCTS and discuss them more in-depth.

Selection – At the beginning we have to select nodes in our simulation tree starting from the root node. We do the selection section of MCTS until we reach a leaf node. The selection of nodes is done according to how much we want to explore the game tree or how much we want to exploit the information we have obtained so far. On the one hand, if we always decide to select nodes with the best results so far (exploit them), we will probably get stuck in a local maximum. On the other hand, it makes sense to explore the unexplored parts of the tree, or occasionally explore a direction that led to a bad result to either verify that direction as a bad one, or discover that there are also good results to be obtained. One possible approach to selection is the Upper Confidence Bound applied to Trees (UCT) (Kocsis & Szepesvári, 2006) where we select the move i that maximizes

$$v_i + C * \sqrt{\frac{\ln N}{n_i}} \quad (4)$$

Where v_i is the value of the node i (usually it is the averaged value of previous games that have visited node i), n_i is the number of times node i was visited, N is the number of times the parent node of node i was visited. In MCTS a node usually contains the values n_i, v_i and others, depending on implementation. The last parameter C has to be tuned experimentally. Parameter C defines how much we want to prefer exploration over exploitation. The larger C the more emphasis we put on exploration.

Expansion – In this step we expand our simulation tree by adding one or more nodes that have not been previously part of the simulation tree. There are two main possibilities how this can be done. We can either add one node per game simulation or add a node only when it has passed some predefined condition (e.g.: the expanded node was visited certain amount of times, etc.)

Simulation / Playout – In this section we simulate the rest of the game from the simulation tree's leaf node till the end (or to some preset depth). The moves here can be chosen randomly but better results can be achieved with pseudo-random moves (for this we require a domain dependent heuristic). Opponent modeling is also an option to better estimate her moves.

Backpropagation – After finishing one simulation of the game we propagate the result (win/lose/draw) of that particular game back through the simulation tree, updating each node in the simulation tree that was part of the path that lead to the terminal state.

How do we create an algorithm from the above 4 steps? We simply put all 4 parts in a control loop as show in Figure 9, where X is the number of times we want the control loop to run. X can be set beforehand or it can be adjusted online depending ,for example, on the time we have before we need to decide on our action. In general game playing we are limited by time, thus X changes with every game. After we are finished running the control loop we select the best action which is the action that was selected the most times at root node. We can see that this approach results in building an asymmetric tree (the promising branches are more expanded than the others).

In the next Section we describe how to apply MCTS on games with imperfect information.

3.4. Perfect Information Sampling

In this Section we describe how we can apply a full information game playing algorithm on games with imperfect information. We use Monte Carlo as an example of the full information game playing algorithm but keep in mind that any other algorithm can be used (e.g.: Minimax, etc.). First, let's discuss what is different between the PIG and IIG that

influences full information game playing algorithms. In PIG the player always knows in which state she currently is. In IIG this is often not the case. Instead the player knows all the possible states in which she can be in. This is caused by the fact that every time opponent performs an action that the player does not see she has to consider all of her possible actions and the states that they lead to. These states are the states player can be in.

However MCTS can be applied only on 1 state. We have 2 options:

- We apply MCTS on all of the states but that can be time consuming since the number of states can grow rapidly or
- We generate samples from all of the states and apply the MCTS only on these selected states. By generating samples we mean choosing a subset of all the states based on some criterion (states with the highest utility for player, random states, etc.)

The second approach we described is the perfect information sampling Monte Carlo (PIMC). From each state we receive an action that MC considers to be best. From these actions we have to select one that we want to play. We discuss this in detail in Section 6.2.3. This way of applying MCTS on IIG has 2 main errors:

1) **Strategy fusion** – This type of error is caused by the fact that perfect information sampling MC (PIMC) incorrectly assumes that in every node it can make the right decision based on the full information about the game. However, in IIG in an information set it cannot make the right decision because it does not know the real current state. This is shown on Figure 10. Here we have a chance node C that randomly chooses if it goes to the left (World 1) or to the right (World 2). In reality, player 1 cannot distinguish between her states because they are in one information set but PIMC assumes that it knows where it is and that in states z and z' it can make the right choice to receive the maximum reward. As we can see this presumption is incorrect. In the example it will happen that instead of taking action R at the beginning and gain the guaranteed reward of 1 in both worlds the player traverses the tree to either state z or z' where there is no guarantee that she will obtain the reward of 1.

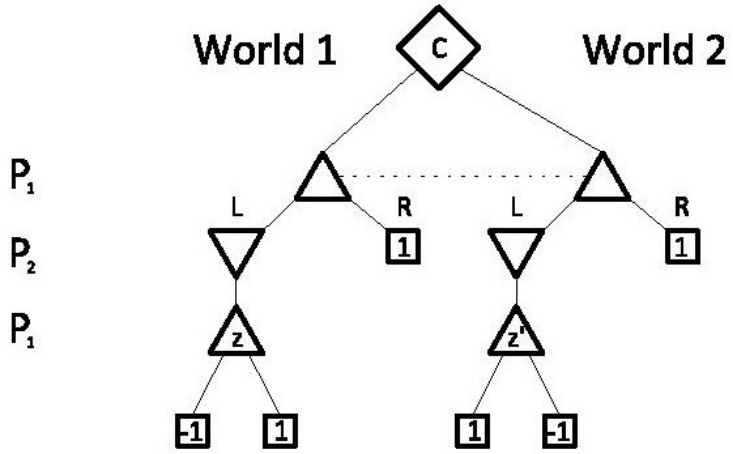


Figure 10 (Long, Sturtevant, Buro, & Furtak, 2010): Example of strategy fusion. The dotted line connects states in one information set and C represents a chance node.

2) **Non-locality** – this type of error is caused by the fact that in imperfect information games the value of any game node is not only dependent on its subtree (this is the case of perfect information game) but it can also depend on other parts of the tree not contained in its subtree. This is because opponent has different information than player and thus will try to direct the game into areas more favorable for her. Let's explain this on an example shown on Figure 11. In perfect information games the value of node n would only depend on its 2 children nodes with rewards -1 and 1. But in imperfect information the value of game node n also depends on node A. This is because player P_1 knows in which state she is after the chance node. Thus if the chance node took the left action and P_1 is able to distinguish between her states, she would not choose to go to the terminal state and gain the utility of -1 and instead she would select the left action and let P_2 play. P_2 cannot distinguish between her states but because of the reasoning we did above she knows exactly in which state she is and thus she is able to select the correct move that leads her to the -1 reward (-1 because P_2 is a minimizing player). But PIMC will perform a random move instead of the best one. This error is, as we can see, caused by the opponents influence on the game and her knowledge which is different from the other players.

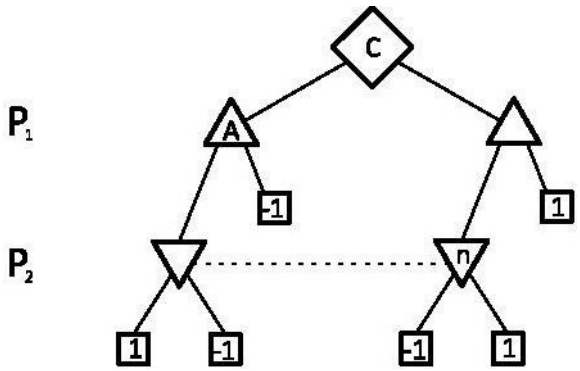


Figure 11 (Long, Sturtevant, Buro, & Furtak, 2010): Example of non-locality. The dotted line connects states in one information set and C represents a chance node.

These 2 types of errors might cause the PIMC to perform poorly on some imperfect information games. However, some games suffer less from these errors than others. An interesting way how to detect how much a given game suffers from these errors is described in (Long, Sturtevant, Buro, & Furtak, 2010).

3.5. Counterfactual regret

Counterfactual regret (CFR) is a new and interesting extension of regret minimization concept and it is lately being used for solving imperfect information games. The theory behind CFR guarantees it to work on 2-player, zero-sum games. However, the Department of Computing Science at the University of Alberta (creators of CFR) achieved good results by using CFR even in poker domain which is a non 2-player game. A more complex and thorough explanation of CFR can be found in (Johanson, 2007) and (Zinkevich, Johanson, Piccione, & Bowling, 2008).

The main idea behind counterfactual regret minimization is that instead of minimizing one regret value, as done in standard regret minimization, we split the regret value into additive terms where each term is dependent on one information set and we minimize those terms. The advantage is that counterfactual regret can then be minimized independently at each information set. Counterfactual regret is defined in (Zinkevich, Johanson, Piccione, & Bowling, 2008) as:

Definition 13 (Immediate counterfactual regret): ***Immediate counterfactual regret*** *is a player's average regret for their actions at I, if they had tried to reach it:*

$$R^T_{i,imm}(I) = \frac{1}{T} max_{a \in A(I)} \sum_{t=1}^{T} \pi^{\sigma^t}_{-i}(I)\left(u_i(\sigma^t|_{I \to a}, I) - u_i(\sigma^t, I)\right) \quad (5)$$

, where

- *$A(I)$ is the set of all applicable actions in information set I.*
- *$\pi^{\sigma}(I)$ is a probability of information set I occurring if players choose actions according to σ. Thus $\pi^{\sigma}_{-i}(I)$is a product of all player's contribution (including chance) except player i.*
- *For all $a \in A(I)$, $\sigma|_{I \to a}$ is a strategy profile identical to σ except that player i will take action a whenever she is in I.*
- *$u_i(\sigma, I)$ is counterfactual utility – the expected utility given that information set I is reached and all players play using strategy except that player i plays to reach I.*
- *T is the number of repetitions of the game*

(Zinkevich, Johanson, Piccione, & Bowling, 2008) proved that the average overall regret:

$$R_{avg,i}^{T} = \frac{1}{T} max_{\sigma_i^* \in \Sigma_t} \sum_{t=1}^{T} \left(u_i(\sigma_i^*, \sigma_{-i}^t) - u_i(\sigma^t) \right) \tag{6}$$

is bounded by the positive portion of immediate counterfactual regret:

$$R_{avg,i}^{T} \leq \sum_{I \in IS_t} R_{i,imm}^{T,+}(I) \tag{7}$$

where $R_{i,imm}^{T,+}(I) = max\left(R_{i,imm}^{T,+}(I), 0\right)$ is the positive part of immediate counterfactual regret. This is important because there is a connection between Nash Equilibrium and average overall regret saying that in a 2 player zero-sum game, if average overall regret is less than ε, then average strategy is a 2ε Nash equilibrium. Based on this, it is easy to see that we need an algorithm that will update $\sigma_i(I)$ in a way that will decrease the counterfactual regret. We will update the values of $\sigma_i(I)$ in the following way:

$$\sigma_i^{T+1}(I)(a) = \begin{cases} \dfrac{R_i^{T,+}(I,a)}{\sum_{a \in A(I)} R_i^{T,+}(I,a)} & if \sum_{a \in A(I)} R_i^{T,+}(I,a) > 0 \\ \dfrac{1}{|A(I)|} & otherwise \end{cases} \tag{8}$$

This way of updating the strategy leads to Nash equilibrium as proved in (Zinkevich, Johanson, Piccione, & Bowling, 2008).

With the counterfactual regret defined we can use it to compute a strategy for any 2-player, zero-sum game. To do so we need to have 2 players play the game repeatedly. In new match T the player i uses strategy σ_i^T. During the match the player traverses the information set tree and updates values $R_i^{T,+}(I,a)$. After each match we update the strategy using the equation (8). It is clear that to be able to do so we need to store the $R_i^{T,+}(I,a)$ for every information set I and for every action a. Unfortunately, to receive a good strategy we need to run large number of games.

This was a short introduction to the concept of counterfactual regret. As stated above, it is a useful approach to solving imperfect information games but one of its disadvantages is that it works well mostly with domain specific knowledge. For example, the reason why CFR works well in the poker domain is that CFR uses card abstraction (called buckets) which is specific for poker alone. Also, they use the fact that information set tree for poker domain is specific and with each move the number of information sets is rapidly decreasing. Again, this is a domain specific thing. Another setback is that achieving 2ε Nash equilibrium is theoretically guaranteed only for 2-player, zero-sum games.

3.6. Information Set Search

This approach was introduced by (Parker, Nau, & Subrahmanian, Paranoia versus Overconfidence in Imperfect Information Games, 2010). It is a game-tree search technique that uses opponent modeling to achieve optimal results. The definition of the Information set search is done on a 2-player, zero-sum game. To explain this approach, we need to define several new things. All the definitions in this Section are taken from (Parker, Nau, & Subrahmanian, Paranoia versus Overconfidence in Imperfect Information Games, 2010).

We have already defined some of the needed concepts in Chapter 2.2. In this section we say that strategy $\sigma_i(m|I)$ is a function that returns the probability of player i making move m in information set I. We can calculate the conditional probability of reaching a history $h \in I$ given that players play according to strategies σ_1, σ_2 as:

$$P(h|I,\sigma_1,\sigma_2) = \frac{\prod_{j=0}^{n-1} \sigma_{a(h_j)}(m_{j+1}|h_j)}{\sum_{h' \in I} P(h'|\sigma_1,\sigma_2)} \tag{9}$$

, where $h =< m_1, \dots, m_j >$ is a history and m_i are actions.

Before we can define expected utility for an information set we first need to define the expected utility of any non-terminal history based on the players' strategies. And for a node where it is player i's turn to move we define it as

$$EU(h|\sigma_1,\sigma_2) = \sum_{m \in M(h)} \sigma_i(m|h) * EU(h \circ m|\sigma_1,\sigma_2) \tag{10}$$

, where m is a move from all possible moves in history h $M(h)$, $\circ$ denotes concatenation and expected utility for a terminal history is the reward of player a_1 for that history $U(h_t)$. Since the game is a zero-sum game the reward of player a_2 is $-U(h_t)$. Here is the same problem as in Minimax. To traverse the whole tree can be time consuming for large games. We can limit the depth to which we traverse the tree in search of leaves and then we prune all the levels of the tree below the specified depth. The smaller tree that we created can be traversed completely. However, the leaf nodes of this new tree might not be terminal states and thus there is no utility that we need for further computations. To cope with this problem we need to apply a heuristic evaluation function that tells us how good/bad the states are.

Now we can define the expected utility of an information set I as a weighted sum of expected utilities of its histories

$$EU(I|\sigma_1,\sigma_2) = \sum_{h \in I} P(h|I,\sigma_1,\sigma_2) EU(h|\sigma_1,\sigma_2) \tag{11}$$

The last thing we need to define is a set of moves in an information set that maximize player a_1's expected utility.

$$M^*(I|\sigma_1,\sigma_2) = argmax_{m\in M(I)} EU(I \circ m|\sigma_1,\sigma_2) \quad (12)$$

We take such actions that maximize the expected utility in the current information set I.

Now we can find an optimal strategy by starting in the terminal histories and traversing the tree upwards applying the equations (10) and (11). The optimal strategy will estimate the probability of moves in the following way. In each information set if the move maximizes player 1's expected utility then it will have the probability of $1/|M^*|$, otherwise the move's probability will be 0. Formally:

$$\sigma_1^*(m|I) = \begin{cases} \frac{1}{|M^*(I|\sigma_1^*,\sigma_2)|}, if\ m \in M^*(I|\sigma_1^*,\sigma_2) \\ 0 \qquad\qquad , otherwise \end{cases} \quad (13)$$

In (Parker, Nau, & Subrahmanian, Paranoia versus Overconfidence in Imperfect Information Games, 2010) is the following theorem that claims that this way of computing strategy σ_1^* results in a strategy that is optimal against a given opponent model (for the proof of this theorem see the above mentioned paper):

Let σ_2 be a strategy for player a_2 and $\sigma_1^*(m|I) = \begin{cases} \frac{1}{|M^*(I|\sigma_1^*,\sigma_2)|}, if\ m \in M^*(I|\sigma_1^*,\sigma_2) \\ 0 \qquad\qquad , otherwise \end{cases}$. Then σ_1^* is a σ_2-optimal.

In most cases the computation of σ_1^* is an intractable issue which can be solved by approximating the utility by a search to some limited depth. Then we use these approximated values as if they are the actual expected utility values. And at this point the information set search technique uses the opponent modeling. There are 2 main opponent models:

1. **Paranoid** – this model expects that opponent will always make the best possible move for her (thus it is a worst possible move for player P_1) and she will minimize player P_1's utility. In IIG this approach might not yield such good results as in PIG because it builds on the assumption that player P_2 knows the exact pure strategy of player P_1 and also has the knowledge that player P_1 has about the game.
2. **Overconfident** – this model assumes that opponent does not consider the information available to her and thus makes her moves randomly (opponent will consider that all her actions have the same utility for her and thus the probability of every action is equal).

There is one more problem that needs to be resolved. Information sets of games such as kriegspiel can be extremely large. To be able to handle such large information sets we can use statistical sampling. We select a subset of the original information set and evaluate the expected utility of the information set based on computing the expected utility of the selected subset.

This is the basic theory behind information set search technique.

In this Chapter we have introduced several algorithms useful for implementing general game players including counterfactual regret minimization, information set search, Monte Carlo and we described their basic idea. Special attention was given to Monte Carlo methods because it is the method that we have implemented in our general game player. Details of our implementation are in Chapter 6.

4 General Game Playing

In this Chapter we introduce the concept of general game playing (GGP). First, in Section 4.1 we describe what a general game player is and how it is different from a specific game player. Section 4.2 provides a short introduction into GGP background; the motivation why the GGP competition was started and how did it evolve. Section 4.2 introduces the Game Description Language (GDL) which is the main language used for describing games in GGP competition, and we explain its syntax on a 2-player Bomberman game example. Then in Section 4.4 we show how easily we can upgrade the GDL into a GDL-II which can describe even games with imperfect information. In Section 4.5 we discuss in depth the GGP server, its role in GGP and the communication between it and players during a match. And in the last Section 4.6 we explain how to create your own imperfect information game with GDL-II and explain everything on a game of Latent Tic-Tac-Toe that we created.

4.1. General Game Player

First, let us focus on a specialized game player (SGP). A specialized game player is an agent specifically designed to play one game (e.g.: a chess computer Deep Blue, a checkers player Chinook at University of Alberta, etc.). Thus she can use the intricacies of the specific game to her advantage. But who is actually doing the thinking of such an agent? It is the agent's programmer who has to analyze the game and design the agent beforehand. Such agents are useful however their value is limited. On the other hand, the value of a player that is able to play several conceptually different games and thus is able to adapt to new scenarios is huge. This brings us to the concept of general game player.

A General game player is a concept that opposes the previously mentioned specialized game player. The general game player is an agent that is able to play a wide variety of conceptually different games without human intervention. Thus it cannot rely on algorithms and approaches specific for one type of game (that are coded in SGP in advance). Simply said, the general game player must be able to figure out how best to play a game given only the game description (we discuss the possible approaches in Section 3.3).

4.2. General Game Playing Competition

General Game playing competition is an annual competition that was introduced and began in 2005 (Genesereth, Love, & Pell, 2005) and it is a project of Stanford Logic Group of Stanford University in California. This competition (sponsored by Association for the Advancement of Artificial Intelligence - AAAI) was started to promote work in the area of general game playing which means moving more of the intellectual work to computers. At the beginning, the competition was focusing only on the general game players for perfect information games (PIG). From this year (2011), the competition should also start supporting

general game players for games with imperfect information. To mention some of the successful general game players for PIG developed during the past years:

- **Flux Player** – winner of AAAI GGP competition 2006. This player uses a Prolog-based implementation of Fluent Calculus for reasoning and non-uniform depth-first search with iterative deepening and general pruning techniques for searching the game tree (Schiffel & Thielscher, 2007).
- **Cadia Player** – winner of AAAI GGP competition 2007, 2008. This player uses UCT/Monte Carlo approach (Finnsson, 2007).
- **Ary Player** - winner of AAAI GGP competition 2009, 2010. This player uses Prolog for reasoning and MC-UCT and was created by Jean Méhat (Méhat & Cazenave, 2010).

4.3. Game Description Language

Game Description Language (GDL) is a language used to describe discrete games with perfect information and their rules in GGP. GDL can describe a wide variety of games: zero-sum games, non zero-sum games, single and multiplayer games, cooperative or adversary, etc. There are few restrictions on the games that can be described by GDL which we already mentioned, but we want to stress them. First, the games have to be of perfect information. Second, the games have to be deterministic (there is no 'chance' player in the game).

4.3.1. Syntax

We will present the syntax of GDL on an example. GDL is a language built on a relational logic whose syntax is close to LISP programming language. There is a universal format for writing relational logic rules called Knowledge Interchange Format (KIF) that GDL uses.

Surely, most of us have heard about the game Bomberman. It is a simple multiplayer game where players move in a maze-like map and their goal is to burn their opponent(s) by using bombs (and, of course, avoid being burnt themselves). The bombs can be placed only on a place which does not contain any other bomb. When a bomb is placed its timer will start and after a specified time the bomb explodes. The explosion has a limited range and does not destroy walls. However, it does burn any player who is in the bomb's range thus eliminating her from the game.

We will now explain the GDL syntax on several lines of GDL description of this game simplified for only 2 players. The whole GDL description of a 2-player Bomberman game can be found in Appendix A. Syntax of all keywords used in the following explanation can be found in Table 1.

At the beginning of most game descriptions you will find a declaration of players. This is done by the relation *role* followed by a name. In our case we have 2 players/roles: bomberman and bomberwoman.

```
(role bomberman)
(role bomberwoman)
```

Now that we have our players specified we need to declare the game board and our initial state. This is done by the *init* relation. The *init* predicate is used only at the beginning of a KIF file and it defines the initial state of the game. E.g.: `(init (location bomberman 1 1))` implies that bomberman starts at location `(x=1,y=1)`. `(cell 1 8)` informs us that there exists a cell with the coordinates `(x=1,y=8)`. If we look at all the cell statements below, we can see that the size of our board is 8x8. The `blockednorth` and `blockeast` statements define where a wall is located. We can see that `blockednorth`/`blockedeast` statement informs us that a player cannot move from the position defined in the statement north/east, because there is an obstacle there (e.g. a wall). The reconstructed game board from the game's GDL description (Dresden GGP server, 2010) is shown on Figure 12.

```
(cell 1 1)
(cell 2 1)
   ⋮
(cell 7 8)
(cell 8 8)
(init (location bomberman 1 1))
(init (location bomberwoman 8 8))
(init (blockednorth 2 1))
(init (blockedeast 1 2))
```

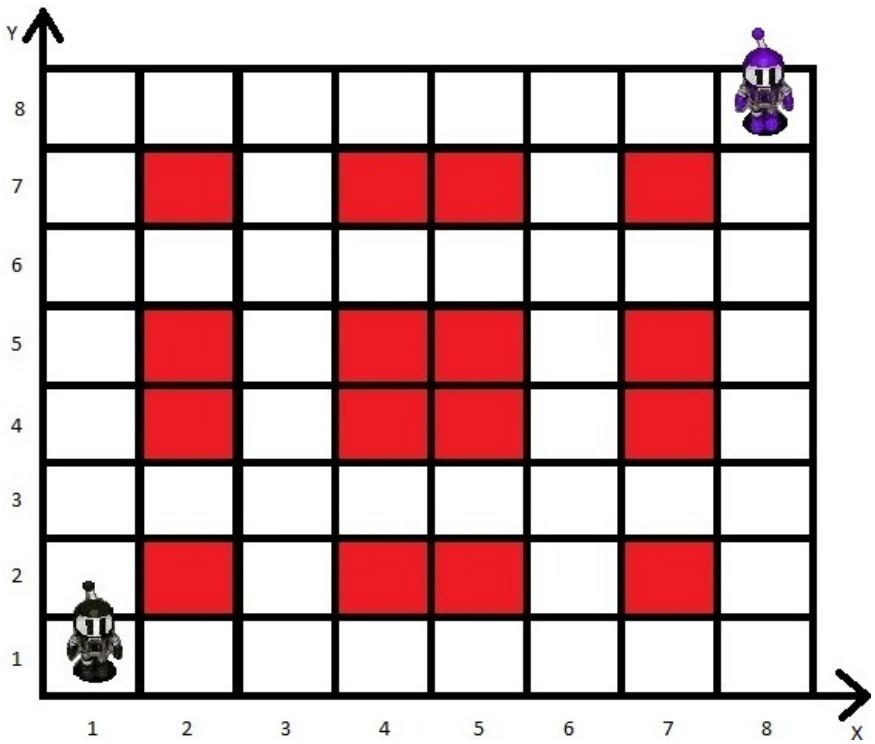


Figure 12: Reconstructed Bomberman gameboard from GDL description with initial starting position of bomberman (black figure) and bomberwoman (purple figure). Figures can move on the white tiles, red tiles are inaccessible (e.g.: wall).

Next we need to define legal actions for players and thus restrict the set of actions. This can be done by using the *legal* relation. In the following code snippet we show one such action. The meaning of the snippet is: if *role* of current player is ?char, then it is legal for her to drop a bomb.

```
(<= (legal ?char dropbomb)
      (role ?char))
```

In the above *legal* relation we encountered the ? sign for the first time. In GDL by ? we represent a variable. Thus ?char is a variable.

In the following code snippet there are 2 important relations. First, we will explain the relation *does*. In general, the *does* relation indicates the moves done by players in some specific state. In our case the do*es* indicates that player ?char does action dropbomb). This relation is mostly used for updating states. The *next* relation defines what holds in the next state of the game. In this case: IF it is player's ?char turn, the player is at location (?x, ?y) and she performs action dropbomb then in the next state (in the Bomberman game the next state is the state in the next time-step) player ?char will be at the same location (?x, ?y).

```
(<= (next (location ?char ?x ?y))
          (role ?char)
    (true (location ?char ?x ?y))
       (does ?char dropbomb))
```

For a game to ever end, we need to define its terminal states. In our game there are 3 distinct ends of the game: bomberman is burned, bomberwoman is burned or time runs out. This is done by the *terminal* relation that defines which conditions must hold for a state to be terminal.

```
(<= terminal bombermanburned)
(<= terminal bomberwomanburned)
     (<= terminal timeout)
```

We can also define our own relations. This is necessary for more complex games to make the code more readable and easier to maintain. One such relation is defined below. *bombermanburned* is a relation that tells us that bomberman is burned when there is fire at location (?x, ?y) and also bomberman stands on the same tile (?x, ?y). Only if both conditions hold, then bomberman is burned.

```
      (<= bombermanburned
(true (location bomberman ?x ?y))
  (true (location fire ?x ?y)))
```

Finally, the last thing needed for defining a game are utilities for players at the terminal state. This is done by the relation *goal*. Utility can vary from 0 to 100. For a Bomberman game we

need to define 4 ending scenarios. The first block in the following code snippet says that if a terminal state is reached and bomberman is burned and bomberwoman is not, then bomberman receives utility 0. If they are both burned then each of them receives the utility of 50 and of course if bomberman manages to burn bomberwoman without burning himself, he gains utility of 100. And the last ending scenario is that time runs out and nobody is burned.

```
(<= (goal bomberman 0)
Bombermanburned (not bomberwomanburned) )

(<= (goal bomberman 50)
Bombermanburned bomberwomanburned )

(<= (goal bomberman 100)
(not bombermanburned) bomberwomanburned )

(<= (goal bomberwoman 50)
          timeout
    (not bombermanburned)
  (not bomberwomanburned))
```

The summary of the main GDL keywords can be found in Table 1.

Keyword	Meaning
role(R)	R is a player
init(F)	F holds in the initial state
legal(R,M)	R can do move M in the current state
true(F)	F holds in the current state
next(F)	F holds in the next state
does(R,M)	player R does move M
Terminal	the current state is terminal
goal(R,N)	R gets N points in the current state

Table 1 (Thielscher, 2010): Main GDL keywords and their meaning.

4.3.2. Restrictions

GDL, as described above, cannot be used as is to create games. There are several restrictions that must be met. In GDL all recursions are restricted to keep the states of the game described by GDL finite. Except for restricting recursion we also require the game to be well-formed. This sets other restrictions for the game definition. According to (Love, Hinrichs, Haley, Schkufza, & Genesereth, 2008) the definition of well-formed game (and thus the restrictions as well) is:

Definition 14 (Termination): *A game described by GDL terminates if all infinite sequences of legal moves from the initial state of the game reach a terminal state after a finite number of steps.*

Definition 15 (Playability): *A game described by GDL is playable if and only if every role has at least one legal move in every non-terminal state reachable from the initial state.*

Definition 16 (Monotonicity): *A game described by GDL is monotonic if and only if every role has exactly one goal value in every state reachable from the initial state, and goal values never decrease.*

Definition 17 (Winnability): *A game described by GDL is strongly winnable if and only if, for some role, there is a sequence of individual moves of that role that leads to a terminal state of the game where that role's goal value is maximal. A game described by GDL is weakly winnable if and only if, for every role, there is a sequence of joint moves of all roles that leads to a terminal state where that role's goal value is maximal.*

Definition 18 (Well-formed Games): *A game description in GDL is well-formed if it terminates, is monotonic, and is both playable and weakly winnable.*

If we adhere to all of the above mentioned restrictions, then the resulting game can be played by any general game playing system that supports GDL. Let's take a look now on the extension of GDL, GDL-II.

4.4. GDL-II

Unfortunately, GDL cannot describe all games. Let's consider the game of kriegspiel – a variant of chess with imperfect information. In this particular game when we lose a pawn we do not know, what type of figure captured our pawn. Only that our pawn was captured on a specified position. However, there is no possibility of sending this information in GDL without extending the language. Another example is that in the game of poker we need to simulate the dealer (a chance player). Again, there is no possibility to do so in the GDL without any extensions. Therefore the GDL was extended to a new GDL-II language that accommodates functions required for defining imperfect information games. This is done by adding 2 new keywords:

- *random* – Defines a special player (chance player) that makes moves randomly. By this player we can model effects such as dice rolling, dealing cards from a deck, etc.
- *sees(R,P)* – Defines that player R perceives P in the next state.

These 2 new keywords are enough to define everything needed for describing games with imperfect information. We will now explain how to create a GDL-II game and what the possible pitfalls in the process are. Another source of information concerning GDL-II with some examples is (Thielscher, 2010).

4.5. Dresden GGP Server

In this Section we explain how different players are able to play against each other in the GGP competition. In perfect information game players could simply connect to one another

and play. But then, each player would also have to be able to deal with incorrect moves sent by opponent and other technical issues. Also, it would be hard to compare performance of multiple players. And imperfect information game players? They should have no possibility of communicating straight with their opponent. They have to communicate only with an umpire that will inform them about what they see during the game.

At first, to deal with the perfect information games a server was created at Dresden University. The GGP server works as an umpire for every game. It starts a match, sends information to players, checks if moves sent by players are correct, holds the state of the game and stops the match when it finishes. It was used in previous GGP competitions for perfect information games.

In imperfect information games the server works in the same way as in PIG but there is one important difference. Instead of sending joint moves (moves done by both players) after every turn the server sends sees terms to each player. There are 3 types of messages server can send to players and they are listed in Table 2.

Message type	Parameters
START	MATCHID ROLE GAMEDESCRIPTION STARTCLOCK PLAYCLOCK
PLAY	MATCHID SEESTERMS
STOP	MATCHID SEESTERMS

Table 2: Messages that can be send from server to players in GDL-II game. Parameters are explained in Table 3

Parameters	Explanation
MATCHID	Match identifier
ROLE	Role a player will play in this game (e.g. in Tic-Tac-Toe if player plays x or o)
GAMEDESCRIPTION	Description of the game that server decided to play
STARTCLOCK	How much time player has before the match starts (in seconds)
PLAYCLOCK	How much time player has for each move (in seconds)
SEESTERMS	What player sees after each turn

Table 3: Message parameters and their meaning

The communication between player and server is done via HTTP. From this point of view each player is a HTTP server that waits for messages from the game server and after each message the player returns an action that she wants to take. Let's explain the communication between game server and a player on an example (see Figure 13).

At the beginning of a match the game server starts the match by sending a START message to all players and expects that they reply READY within the given STARTCLOCK time. As soon as all players are ready, game server sends first PLAY message. This PLAY message is different from all the other play messages during the match because it contains a NIL string in sees terms instead of actual sees terms (it is only logical since no move has been played thus no sees terms can be computed). Each player is thus informed that the match has started and now has the PLAYCLOCK time to come up with a move and send it to the game server in a PLAY message. After all players send their move the game server will first check if

the submitted moves are legal. If any of the moves is illegal, server will play a random move for the player that sent that move and only inform her about this move. The same thing happens if a player does not send her move in the specified time limit. If the move is legal, then the game server evaluates the new game state and sends the computed sees terms for specific player to that player alone. If a player has no sees terms, she will receive an empty PLAY message. Thus all players are informed that a turn has been made and they should send their new moves. The exchange of the PLAY messages from player (containing move of current player) and the PLAY messages from game server (containing sees terms for given player) is repeated until the match reaches a terminal state. When a terminal state is reached server sends the STOP message and ends the match. The STOP message also contains sees terms for each player.

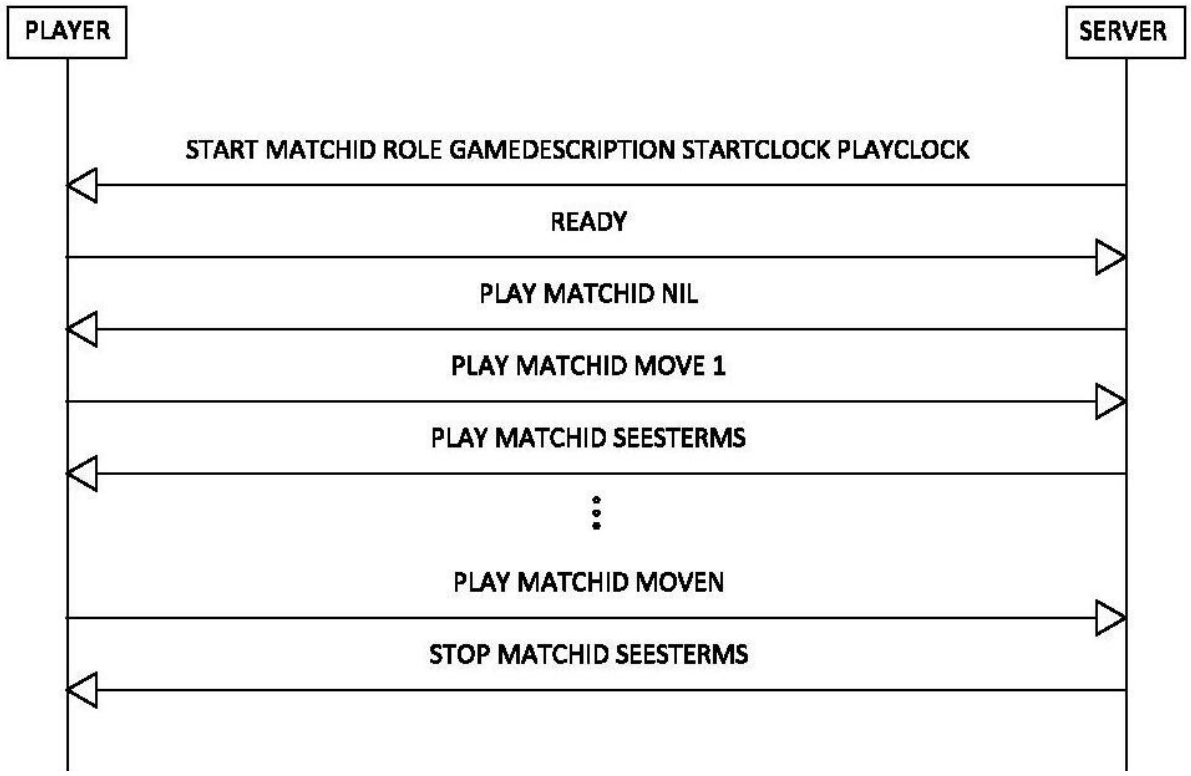


Figure 13: Messages during game

In our implementation (that we discuss later in Chapter 6) we use a standalone game server (a clone of the online GGP server) written in pure java that is suitable for local game playing and development. It can be downloaded as a jar file or you can obtain the source codes directly from http://sourceforge.net/projects/ggpserver/. We use the source codes for gamecontroller-cli-r466.jar.

4.6. Creating GDL-II games

One of the biggest problems of GDL-II is the fact that there are almost no games specified in GDL-II. On the Dresden GGP server there are only 2 such games (one of them being a simple random guessing game with 1 turn) and their state space is fairly limited for testing how well

our player performs. Thus we had to create our own games. We created a Latent Tic-Tac-Toe game that is based on a Tic-Tac-Toe game available on Dresden GGP server. For the complete GDL-II description of Latent Tic-Tac-Toe see Appendix B. The description of Latent Tic-Tac-Toe and how its rules differ from standard Tic-Tac-Toe was given in Section 2.2. We have also written a simple card game taken from (Thielscher, 2010). Before we continue, we would like to discuss creating GDL-II games and how easy/difficult it can be.

When creating a GDL-II game, there are parts that need not be altered from GDL games. Roles, terminal states and goals can stay the same therefore we will not discuss them here. We focus on the parts that are different and try to explain the complications they bring. In the following text we use xplayer to refer to the Tic-Tac-Toe player whose marks are X and to the other player as oplayer.

Probably the biggest difference between GDL and GDL-II is in specifying legal moves. In PIG a legal move is any move that meets the rules of the game. In Tic-Tac-Toe a legal move is any move that places a mark on a blank field (illegal move is any move that places a mark on top of any existing mark). This can be done because we know where our opponent has placed her marks in the previous turns. But what happens if we use the same definition of legal moves in Latent Tic-Tac-Toe? It might seem that it will work fine. And actually it will work fine, unless you make a mark on a place where our opponent has placed her mark. First, we focus only on defining legal moves and for now we disregard the altering of moves that would cause other problems as well.

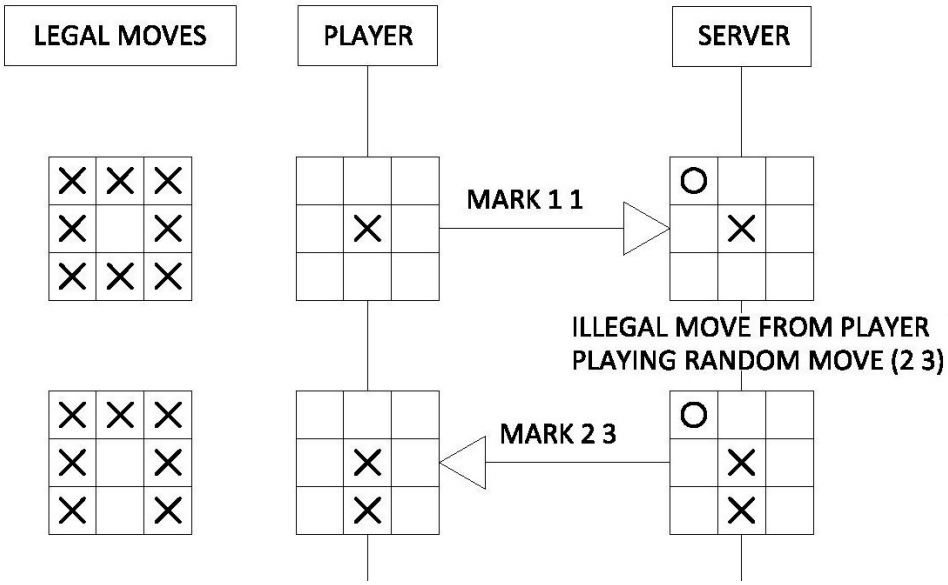


Figure 14: Problems arising from defining the legal moves as in standard Tic-Tac-Toe.

On Figure 14 reader can see 3 game boards. The leftmost boards show all legal moves available for player in a specific state which is represented by the middle column. The rightmost board represents the knowledge of server about the real state of the game. In the first player's state we can see that all moves except the move on position (2,2) are legal. Let's say that the player decides to play on position (1,1) – a legal move based on the

information that the player has available to her. What happens on the server? Because we have defined legal moves as in standard Tic-Tac-Toe the move is evaluated as an illegal move by the server because the server has perfect information about the state of the game and knows that on position (1,1) is an O mark. Therefore, the server penalizes the player by playing a random move for her and lets the other player play. It is clear that this approach is wrong. It does not make sense to penalize a player for a move that she had no chance of knowing is an illegal move. Thus we need to find a different way of defining legal moves.

The next step is to consider that all moves are legal. This solves the problem of penalizing players for their good moves. However, it creates a new problem. Every time a player generates legal moves for any state the legal moves will always be the same regardless of the information the player has received from the server. Let's look on Figure 15. The player sends a move MARK 1 1 to the server. The server correctly considers the move to be legal because all moves are legal. But what if player played move (MARK 2 2)? Or in the next turn player sends the same move as before to the server. These moves should be considered illegal since the player already knew that there was a mark on location (1,1) or (2,2). Of course we can solve this problem by adding additional mechanisms into our player to check if a move was applied before but then the player would not be a general game player but a game specific player. Thus this must be solved on the game definition level.

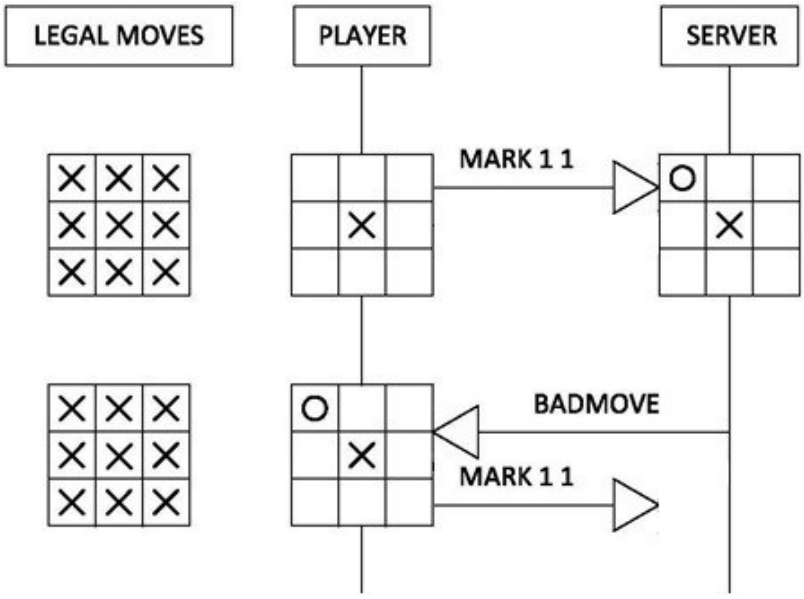


Figure 15: Problems arising when all moves are considered legal.

The above problems can be solved by the idea of a valid move. We claim that a move is valid if, based on the information the player has, she can expect the move to be legal. Thus in our example the move made by xplayer on Figure 15 is a valid move because xplayer had no way of knowing that there is an opponent's mark at location (1,1). To create a working description of Latent Tic-Tac-Toe we need to do 2 more changes. First, we need to change the definition of game state. If we make the game state contain information concerning where we have played before and we change how we define legal moves, then we can easily

generate correct legal moves based on this information and solve the problems on both the player and server side. In our Latent Tic-Tac-Toe game our state is represented not by one game board, but by three. One contains the information concerning the placement of X and O marks (this is the standard board from Tic-Tac-Toe). One holds the information concerning where xplayer has played (let's call it deniedx) and the other has the information concerning where oplayer has played (deniedo). Second, we need to slightly change the definition of legal moves. We consider a move legal when it was not played before (which means that depending on who is playing the deniedx or deniedo cell will be blank). See Figure 16 how this can be done.

```
(<= (legal xplayer (mark ?x ?y))
  (true (deniedx ?x ?y blank))
    (true (control xplayer)))

(<= (legal oplayer (mark ?x ?y))
  (true (deniedo ?x ?y blank))
    (true (control oplayer)))

     (<= (legal xplayer noop)
     (true (control oplayer)))

     (<= (legal oplayer noop)
     (true (control xplayer)))
```

Figure 16: Definition of legal moves in Latent Tic-Tac-Toe.

By doing the three above mentioned steps (adding valid moves, expanding game state and defining legal moves based on the expanded game state) we can generate legal moves with taking into account the moves we have played in the past and the information gained from game server. Therefore we will never play one move twice which should be considered a bad move and as a penalty the game server will play randomly for the player and just send her the information where it played. On Figure 17 is an example of a game state in the middle of a match both from the view of the players and the game server. The main difference between the standard Tic-Tac-Toe and Latent Tic-Tac-Toe is that now we do not generate legal moves from the known state but from the deniedx part of our game state for xplayer and from deniedo part for oplayer. Why does oplayer have 2 different deniedx boards? The answer is simple. Since the last move made in the presented match was MARK 3 1 by xplayer then the server would reply to xplayer with the message BADMOVE and to oplayer with an empty sees term. Thus the oplayer knows that xplayer tried to play on one of her marks. However she does not know if it was on the location $(1,1)$ or $(3,1)$. In our implementation the player will not have 2 deniedx boards but actually she will have 2 game states, one with the upper deniedx board and one with the bottom deniedx board.

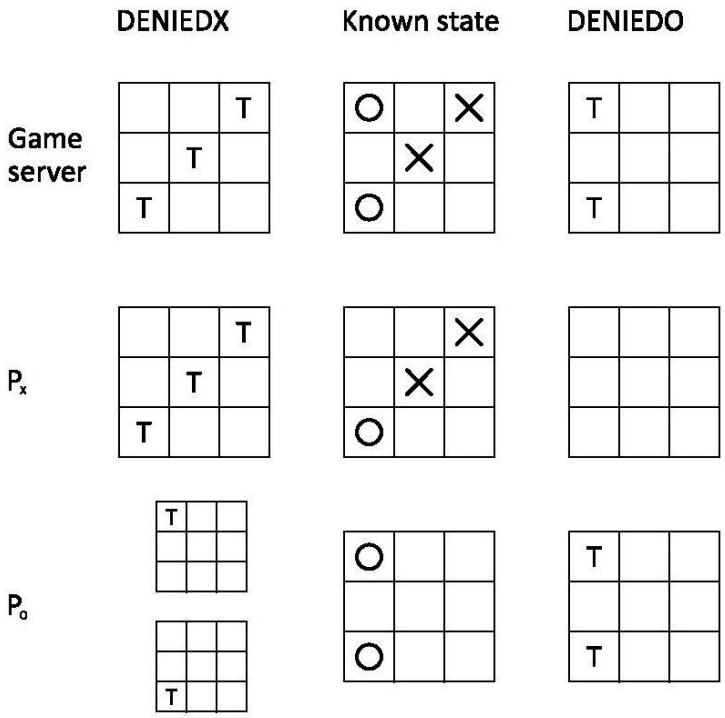


Figure 17: Game state of Latent Tic-Tac-Toe. The game state represents situation after the last move was done by xplayer (mark 3 1). Thus oplayer has 2 possible states, one with the upper DENIEDX board and one with the lower DENIEDX board.

Changing the game state and the legal moves requires us to redefine how the players will alternate turns. That is quite simple. A player will change turn if the move she made was legal and valid. If the move was legal but not valid, then she plays again. We do not use a "valid" keyword anywhere in the rules. The validity is checked by the bold lines on Figure 18.

```
(<= (next (control oplayer))
(does xplayer (mark ?x ?y))
(legal xplayer (mark ?x ?y))
(true (cell ?x ?y b))
(true (control xplayer)))

(<= (next (control xplayer))
(does xplayer (mark ?x ?y))
(legal xplayer (mark ?x ?y))
(true (cell ?x ?y o))
(true (control xplayer)))
```

Figure 18: Definition of alternating turns for xplayer. Bold lines check for the validity of a move.

As we can see, even in the case of a relatively simple game such as Latent Tic-Tac-Toe there are lots of things that need to be taken into consideration when writing the description of the game. You can imagine that creating a game description in GDL-II for more complex games can prove to be a challenge.

In Chapter 4.1 and 4.2 we have introduced the idea of General Game Player and the motivation why General Game Playing competition emerged. Then, in Section 4.3 we described the Game Description Language for games with perfect information and explained its syntax on a 2 player Bomberman game. After that, in Section 4.4 we extended GDL to GDL-II, a language that is able to describe imperfect information games. In Section 4.5 we have discussed in depth the GGP server which is paramount for playing imperfect information games. We described the communication between the game server and players. In the last Section 4.6 of this Chapter we have created our own GDL-II game of Latent Tic-Tac-Toe. On it we described how it was done and what must be taken into consideration when creating such a game.

5 Discussion

In this Chapter we discuss the possible ways which we can use for implementing our player. Because we are considering applying for the general game playing competition we have the following requirements for the approach:

1) The approach must be working well without using any domain-specific knowledge (except for the rules of the game that are provided).
2) The offline learning time must be short enough that we would be able to participate in the GGP 2011 competition (the longest offline time for learning a game that can be found on Dresden GGP server is 600 seconds).
3) Convergence to Nash equilibrium (or any other assurance that the approach will provide a good strategy).

In the following Sections we discuss 3 possible approaches and how do they meet our requirements.

5.1. CFR

First, let's see how well CFR meets the 3 requirements we set.

1) In the poker domain CFR uses 2 domain-specific features: Card abstraction and using the fact that information set tree for poker domain is specific and with each move the number of information sets decreases. This makes CFR unsuitable for GGP. CFR can work without the domain-specific knowledge but the time needed for offline learning increases.
2) Although CFR provides good results for poker domain it is rather time consuming. The Poker player created by University of Alberta had to learn offline for about 2 weeks to play at proficient level:

The 10-bucket abstraction presented in this figure was one of the poker agents the CPRG submitted to the 2007 AAAI Computer Poker Competition. To produce this agent, the algorithm was run for 2 billion iterations over 14 days, using the parallel implementation and 4 CPUs. (Johanson, 2007)

3) The convergence to 2ε Nash equilibrium is guaranteed only for 2 players, zero-sum games.

Because of the major time requirements, we do not think that CFR in its vanilla form is suitable for general game playing. The memory requirements are small, we only need to store 2 information set trees which are generally much smaller than the extensive form game trees. To prove this, let's consider the Texas Hold'em with a standard deck of 52 cards.

First, we are dealt 2 cards. There are $\binom{52}{2} = 1326$ possible hands. Now our opponent is dealt 2 cards (she has $\binom{50}{2} = 1225$ possible hands). If we were building the extensive form tree then we would have a root node (no cards dealt), then the root would have 1326 children nodes (all possible hands dealt to me). Then each of these children nodes would have 1225 children nodes (all possible hands dealt to opponent). That is a total of $1\,624\,350$ nodes at the 2nd ply of the game (after the cards are dealt to both players). If we build only the information set tree (IST), then we only need to remember 1225 'nodes' at the 2nd ply instead of $1\,624\,350$ because from every node in the 1st ply we can join all its 1326 children nodes in the 2nd ply into 1 information set.

5.2. PIMC

Perfect Information sampling Monte Carlo meets our requirements in the following way:

1) It does not require any prior or domain-specific knowledge.
2) It does not need any offline learning time. This time can be used for running game simulations in advance. During gameplay this approach is an anytime algorithm. The more time we have the more simulations we can run and thus the better results we obtain.
3) In 2-player, altering moves, perfect information games the MCTS strategy converges to Nash equilibrium – *In two-player games the move/value returned by UCT converges to the same result as Minimax.* (Sturtevant, 2008). Unfortunately, PIMC has no such guarantees due to the non-locality and strategy fusion problems.

In general game playing we do not know how much the games we receive suffer from the non-locality or strategy fusion errors that we discussed in Section 3.3. However, we can avoid applying PIMC on such games. After we receive the game description we check how much the game suffers from these errors. If it suffers only slightly we can apply our PIMC solution. If the errors would render the PIMC approach worse than a random player we can choose another approach instead of PIMC (this option is not implemented in our work).

5.3. ISS

Information Set Search is a method that has been already tested on the game of kriegspiel (Parker, Nau, & Subrahmanian, Paranoia versus Overconfidence in Imperfect Information Games, 2010) and it proved effective. It outperformed HS, a kriegspiel algorithm introduced in (Parker, Nau, & Subrahmanian, Game-Tree Search with Combinatorially Large Belief States, 2005).

1) It does not require any prior or domain-specific knowledge.
2) No offline learning time is required but as in PIMC this time can be used for running game simulations in advance.

3) In (Parker, Nau, & Subrahmanian, Paranoia versus Overconfidence in Imperfect Information Games, 2010) authors claim that with an accurate opponent model this method produces optimal strategy against a given opponent model. However, since we never have an accurate opponent model we have no guarantees of convergence to Nash equilibrium.

We show analytically that with an accurate opponent model, information-set search produces optimal results. … . Let σ_2 be a strategy for player a_2 and $\sigma_1^*(m|I) = \begin{cases} \frac{1}{|M^*(I|\sigma_1^*,\sigma_2)|}, if\ m \in M^*(I|\sigma_1^*,\sigma_2) \\ 0 \qquad\quad , otherwise \end{cases}$*. Then σ_1^* is a σ_2-optimal* (Parker, Nau, & Subrahmanian, Paranoia versus Overconfidence in Imperfect Information Games, 2010).

5.4. Conclusions

Only CFR provides assurance that the resulting strategy will converge to Nash equilibrium for a two player game. ISS provides an optimal strategy only with an accurate opponent model and PIMC provides none. Compared to CFR which needs an offline learning time, ISS and PIMC need none.

Memory requirements of PIMC are relatively low (in the worst case scenario we have to remember the whole EFG tree which does not happen since mostly we run out of time before we are able to complete the EFG tree). Memory requirements in PIMC can be easily restricted by choosing the maximal depth of the simulation tree that we want to remember during the expansion phase. In ISS the memory requirements are similar to PIMC. In the worst-case the ISS builds the information set tree and samples the histories in each information set. In CFR we need to hold in memory values $R_i^{T,+}(I,a)$ for every information set I and for every action a.

We have decided to implement both the PIMC and the ISS. Unfortunately, due to lack of time we were able to implement only the PIMC. In the next Chapter we discuss the specifics of the implementation of our general game player TIIGR that is based on PIMC.

6 Implementation

In this Chapter we discuss the specifics of how we implemented TIIGR, our general game player, what existing systems we used and what were the pitfalls we encountered. In Section 6.1 we introduce the Palamedes project which is considered as a basis for our work. Then in Section 6.2 we explain in detail how we implemented our general game player, what are her components and the whole architecture of the player.

6.1. Palamedes

Palamedes is an IDE developed by Ingo Keller. It provides user with 3 Eclipse plugins:

1. **UI Plug-In**
2. **GDL Core Plug-In**
3. **KIF Core Plug-In**

From the 3 plugins we only use the GDL Core plugin and thus we only discuss it. For further information regarding the other plugins see (Keller, Palamedes IDE, 2008). The GDL Core plugin provides the general game components for translating the game description into Palamedes GDL components (for interfaces see Figure 19). For each of these interfaces there exists a specific implementation based on one of the 3 supported reasoners: Jocular-0.2, JavaProver, PrologProver.

Interface	Content
IGame	Facade for all game functionality
IReasoner	Interface for reasoner adapter implementations
IFluent	Fluent adapter interface, provided by a reasoner
IMove	Move adapter interface, provided by a reasoner
IGameTree	Interface for game tree implementations
IGameNode	Interface for a minimal game node functionality
IGameState	Interface for a minimal game state functionality
IStatistic	General Interface for statistic support

Figure 19 (Keller, Palamedes: A General Game Playing IDE, 2009): Game Model Interfaces of Palamedes

Palamedes also provides users with a communication layer that can receive and parse messages from the game server and send replies to it. This layer is based on a simple embeddable java HTTP server called NanoHTTPD.

There is a basic general game player in Palamedes that connects the GDL Core Plugin functionality and a communication layer to exchange messages with the game server.

Otherwise, it has no other functionality and serves only as a template upon which users can build their own players. We used this basic general game player as a base for our project.

To sum up, Palamedes parses the game description into its own components so that we can use it for further reasoning. It also provides connectors to the 3 reasoners mentioned above and it has a built-in communication layer. Explanation of how the game description is converted into GDL components of Palamedes, how was Palamedes implemented and other details are not a part of this thesis. If the reader is interested in this project, we recommend him in-depth studies of (Keller, Palamedes: A General Game Playing IDE, 2009), (Keller, Automatic Generation of Game Component Visualization, 2010), and the project's web page http://palamedes-ide.sourceforge.net/ which contains all the information concerning project Palamedes, including source codes, architecture design, class specifications, etc.

6.2. TIIGR

Up till now we have introduced the main system that we use and now we can proceed to explaining how our general game player TIIGR was implemented.

TIIGR consists of the following parts:

1. **Palamedes - Reasoners**
2. **Perfect Information Game Player**
3. **Imperfect Information Generalization**

The architecture of our player is shown below on Figure 20.

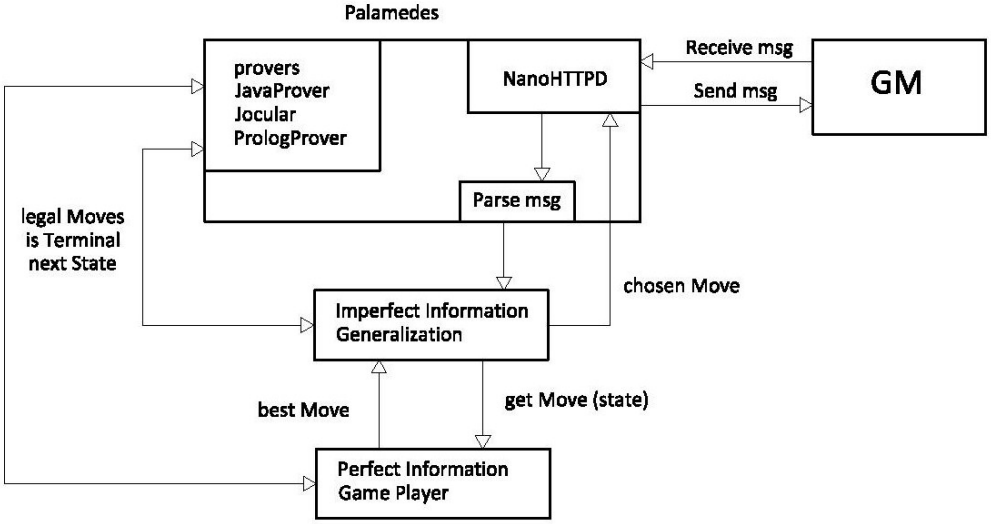


Figure 20: Architecture of our imperfect information player.

It works in the following manner: Game server communicates with our player and vice versa through Palamedes NanoHTTPD. After the message is received it is sorted depending on its type (START, PLAY, STOP), parsed and forwarded to our player (the Imperfect Information Generalization part). There, the information is used to select which states our player can

currently be in. Then, our Perfect Information Game Player is launched on a subset of the possible states our player can be in. One move per each state is returned. We select one of the moves and send it to server. In both the Perfect Information Game Player and the Imperfect Information Generalization we use reasoners to query for legal moves, for what is the next state when we apply a move on a specific state, etc. This is the general overview of how our player works.

Let's explain the building blocks of our player one by one.

6.2.1. Palamedes - Reasoners

Reasoners provide a way to work with the GDL language. At this point, Palamedes already provides 3 reasoners. It also offers the option to use other reasoners as well, as soon as adapters are created. It is important to say that all of the following reasoners were created for reasoning about GDL. The existing reasoners included in Palamedes are:

1. Javaprover – Java based resolver which is easy to use since it runs in the same JVM. Unfortunately, it is the slowest of all 3 resolvers and does not support reasoning about GDL-II.
2. Jocular – Also a Java based resolver which is faster than Javaprover but slower than Prologprover. Its disadvantage is the inability to parse some special characters like +, even though they are allowed in KIF format. It also does not support reasoning about GDL-II.
3. Prologprover – the fastest reasoner of the three. It is a prolog, not Java reasoner and thus requires an installation of TkEclipse to work. In Java it then creates an embedded Tkeclipse connector. It is harder to activate but the main reason why we do not use this reasoner is that it also, like Jocular and JavaProver, does not support reasoning about GDL-II.

On top of each of these reasoners there is a reasoner adapter. These provide several functions that are needed for general game playing. These functions are:

- getLegalMoves(state, role) - returns a list of legal moves for the player with specified role.
- getNextState(state, move) – based on a state and a move it returns a new state that is gained by applying supplied move in given state.
- getGoalValues(state, role) – returns a goal value of supplied state for the player with given role.
- getRoleNames – returns the roles of players in this match.
- getInitialState – returns the initial state of this match.

All of the reasoners we listed do not support GDL-II. However, we were only discussing the versions that are available in Palamedes. To be able to work with GDL-II game descriptions we had to extend Palamedes reasoner Adapter interface to accommodate a newer version

of Javaprover that we took from the Dresden GGP Server. This version is able to parse and work with GDL-II game descriptions and so we are all set to start creating our own player.

6.2.2. Perfect Information Game Player

This part of our player is the part that is able to play perfect information games and is further used in imperfect information games. Basically, this is the implementation of Monte Carlo Tree Search with upper confidence bound for trees as described in Section 3.3. It accepts a single perfect information state and returns a single move. During its search for the best move to play it extensively uses the reasoner's functions defined in previous section. Now we discuss our specific implementation of each part of MC-UCT:

Selection – From root we traverse the simulation tree until we hit a leaf. In our implementation we consider that a leaf is any state, that does not have all its children expanded (there is at least one action in this state that has not been tried yet). Also the selection is made according to the UCT formula $v_i + C * \sqrt{\frac{\ln N}{n_i}}$. Parameter C and how to tune it is discussed in Section 7.3.

Expansion – We expand only 1 node during each run of the MCTS loop.

Simulation – In this step the moves of TIIGR and our opponent are random. We do not use any opponent modeling.

Back propagation – is done exactly as explained in Section 3.3. We propagate the reward of the particular game through the simulation tree up to the root and update the v_i at each node.

Since the MC-UCT was already discussed in detail in Section 3.3 let's move on to the generalization of our Perfect Information Player for IIGs.

6.2.3. Imperfect Information Generalization

In IIG we need to reason about in which states we currently are. At the beginning of every game there is 1 initial state. During the course of the game the number of states we can be in increases and we need to be able to select the right states based on the information we receive from the game server. This is done in the following manner. Every time we receive a PLAY message from game server we apply the algorithm shown on Figure 21 and a graphical example of how this algorithm works is shown on Figure 22.

```
1     seesTermsFromGS = getSeesTermsFromGS();
2     FOR state : currentStates DO
3           legalMoves = getLegalMoves();
4           FOR legalMove : legalMoves DO
5                 IF legalMove.myMove == moveSentToGSpreviousTurn
6                 THEN
7                 {
8                       localSeesTerms = getSeesTerms(state, legalMove)
9                       IF localSeesTerms == seesTermsFromGS
10                      THEN
11                      {
12                         newState = createNewState(state,legalMove)
13                         IF !newState.isTerminal
14                               newStates.add(newState)
15                         END
16                      }
17                }
18          END
19    END
20    currentStates = newStates;
```

Figure 21: Algorithm for choosing states a player can currently be in. GS stands for game server.

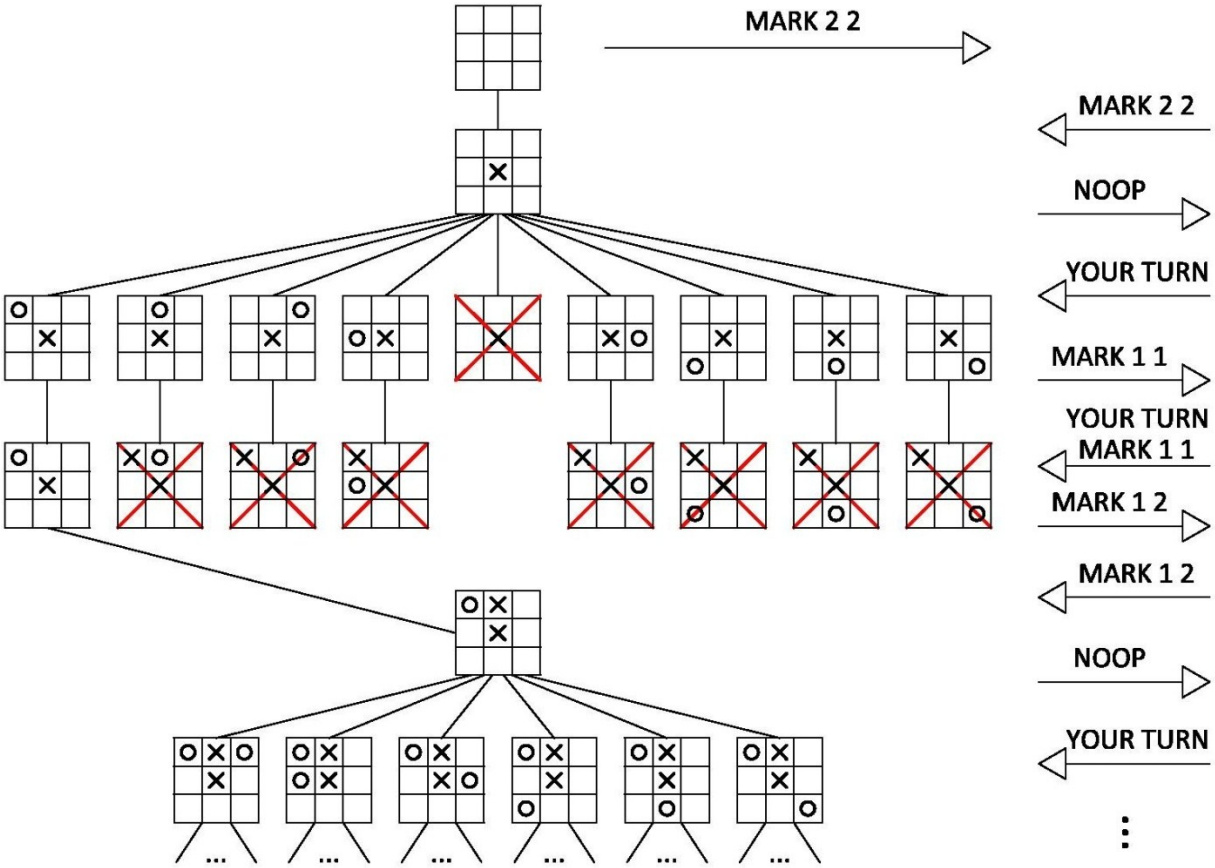


Figure 22: Filtering of states depending on sees terms received from server. Messages sent from the state are sent to the game server. Those send to our states are the sees terms sent by game server. The crossed out states are those that do not match the sees terms received from game server.

We now explain the above algorithm. At the beginning in the PLAY message we receive sees terms from game server which we save for future use (Line 1). Then we take every `state`

we have in our `currentStates` set and for each state we obtain all legal joint moves (our moves combined with opponent moves) (Line 2-3). From these moves we select only those which contain our move that we have previously sent to game server (Line 5). The moves that pass this condition are applied to the `state` and we obtain sees terms (Line 8). The sees terms are then checked against the sees terms we received from game server (Line 9). If they match we create a new state which can be our new current state (Line 12). The last check is to see if the `newState` is not terminal (does not make sense to add a terminal state to the possible states we can be in since the match would already be ended by the game server) (Line 13). If the state is not terminal, then add it to a set `newStates` (Line 14). After we are finished with all states we replace the existing `currentStates` set with our `newStates` set (Line 20). A graphical representation of this algorithm is shown on Figure 22.

The other option is to set a required time for 1 state. Then, depending on the PLAYCLOCK time the player computes how many states she will be able to search in the remaining time. Advantage of this approach is that the maximum amount of states will be searched and each of these states will have the minimum specified time.

At this moment our implementation uses the first option. We check the time after we have computed all possible solutions and we divide the remaining PLAYCLOCK time in a manner that each of our fixed number of states receives the maximal possible time.

When we have decided on the number of states, we now need to determine how to select them. There are numerous approaches but we will mention the border cases. First approach is the overconfident approach. In this case we select states randomly and by doing this we simulate the opponent's lack of knowledge (or lack of ability) to deduce what moves are good or bad for her. This approach works particularly well against random player because this player perfectly simulates a player with the lack of ability to deduce good moves.

The second approach is the paranoid approach. In this situation we assume that the opponent knows our strategy and thus is able to make such a move that will lead her to the best possible state for her (thus it will be the worst possible state for us). Therefore we select states that are favorable for our opponent. Decision about which states are favorable for us or the opponent can be made based on the information gained from previous MC-UCT runs. For example we can use the confidence values of moves to determine which state is favorable for us and which is favorable for our opponent. This approach will perform poorly against random player but should perform better against a more intelligent player.

In our player we have implemented the overconfident approach.

At this moment we have applied our perfect information game player on each of the chosen states and from each of them we have received 1 best action. How should we decide which action is the best? First idea is to count the number of times each action was returned. Thus,

if 4 states return move A and 2 states move B, we would select move A. This is the most basic method and therefore we do not use it. Another method is to add a confidence value to every move that is returned. We chose the number of times the action was taken in the specific MC simulation to represent action's confidence. Then there are 2 possibilities how to select the best action. We either select the action with the best confidence or we make an average of confidence values of the same moves, and then pick the move with the best averaged confidence value. The second option is the one that we use in our implementation.

In this chapter we have introduced TIIGR, our general game player for games with imperfect information. First, in Section 6.1 we introduced the Palamedes project which is the base of TIIGR. In Section 6.2 we proceed to explaining TIIGR's architecture and all the vital components that she consists of. Then in separate sections we discussed each of the main parts more in detail. Then, in Section 6.2.1 we listed 3 main reasoners that can be used and we have chosen JavaProver for TIIGR. After that, in Section 6.2.2 we showed the implementation of Perfect Information Player that is integral part of our player. At the end, in Section 6.2.3 we revealed how TIIGR reasons about states in games with imperfect information.

7 Experiments

We performed several experiments with our player. All of the experiments were run on a computer with dual core processor P6100 with frequency 2Ghz and 3 GB RAM. Most of our experiments were on our Latent Tic-Tac-Toe game. Other experiments were performed on the rest of the existing GDL-II games.

The main parameters that should influence the performance of our algorithm and that we focus on are:

- **Time to search 1 state**
- **States to be searched** (we explained this term in Section 6.2.3)
- C **parameter of MC-UCT**

Our goal is to discover how substantive effect does the 3 above mentioned parameters have on TIIGR's performance and how to set these parameters to obtain the best results from TIIGR. We played around 8000 games with different settings. Mostly for each setting it was 100 games as XPLAYER and 100 games as OPLAYER. Each match was scored 100 for the winner, 0 for the loser and 50 for both players in case of a draw. In the following experiments if not stated otherwise, we use a random player as our opponent.

7.1. Variable time

By these experiments we are trying to verify that increasing the time that we give to our player influences her performance. We fix the number of "States to be searched" set to 5 states. If there are fewer states then the set size than they would all be searched. For each state there was a time set ranging from $t = 100ms$ to $t = 5000ms$. We collected data about the average score, win/loss/draw ratio and number of turns during a match. First, we present the average score of our player on Figure 23.

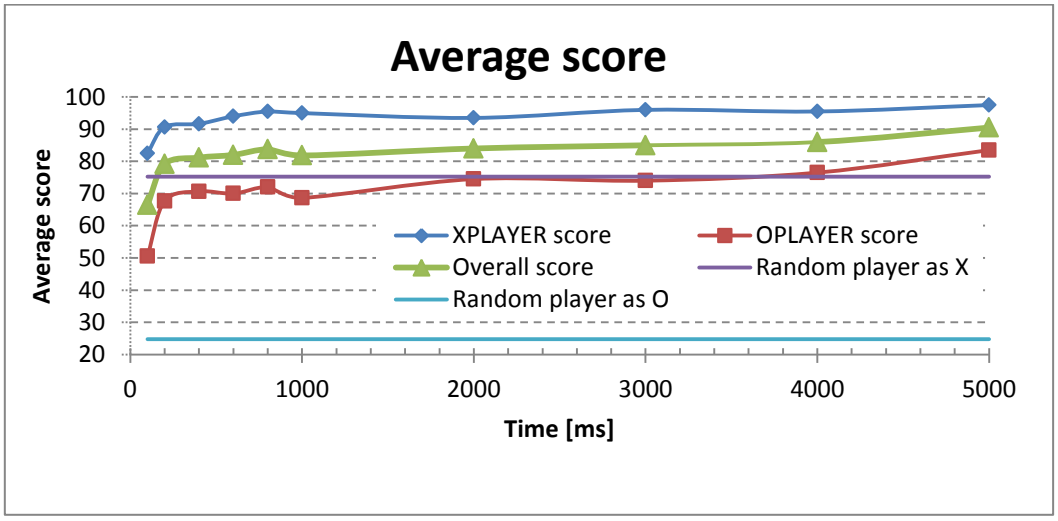


Figure 23: Graph with the average score with a fixed size of "States to be searched" set = 5.

As we can see there are 5 graphs on Figure 23. First, let's start with the Random player graphs. They represent the average score of a random player playing against another random player. We use these 2 graphs as a reference to which we compare our player's performance. The graph labeled "XPLAYER score" shows how the performance of our player changes depending on the time given if the player plays only as XPLAYER (this player always starts first at the beginning of each match). The graph labeled "OPLAYER score" is equivalent to the previously described graph, only it represents the OPLAYER (this player always plays second). And the last graph represents the average overall score of our player. As we can see on Figure 23 the results were to be expected. At all times our player performs better than the random player. If we increase the amount of time given to our player her performance increases. Interesting thing is that in the game of Latent Tic-Tac-Toe the starting player seems to have a great advantage over the second player. Even when our player was playing almost randomly (with the time $t = 100$ ms we were approaching this behavior) it still had the average score of 82.5 compared to when playing as the OPLAYER where the score was only 50. The behavior of all graphs for $400 < Time < 1000$ (the fluctuations of average score) are caused by the fact that during the MC-UCT there are moves that might seem good when we explore their sub tree to some extent. However, if we have more time and explore the sub tree more thoroughly, we discover those moves to be bad. This can cause fluctuations such as we can see on Figure 23. After a certain amount of time (in Latent Tic-Tac-Toe it is 1000 ms) the average overall score monotonically increases and these fluctuation stop.

On Figure 24 we can see that the percentage of wins steadily increases with increasing time at the expense of the percentage of losses which decreases. The percentage of draws is more or less the same. With $Time > 2000$ we can see that our player more often manages to finish the game with a win or draw and not lose as much as for $Time < 2000$.

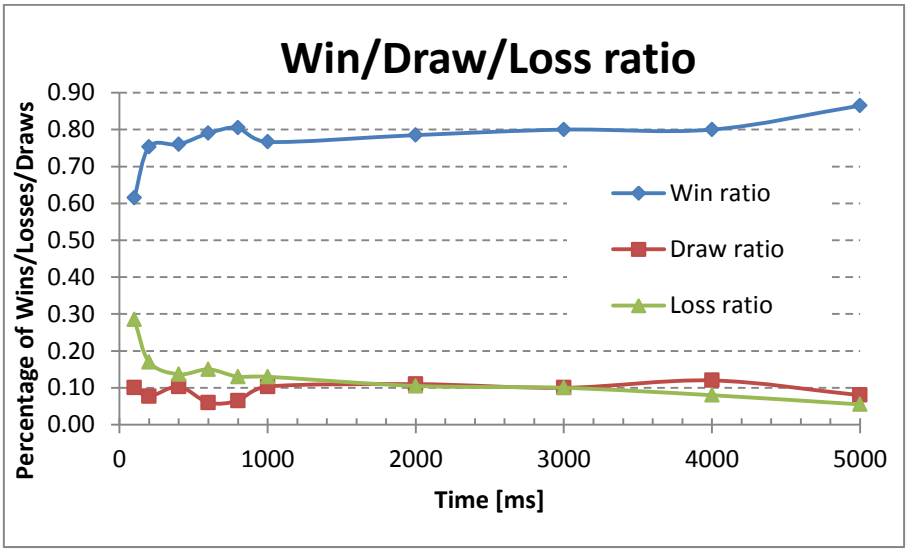


Figure 24: The percentage of wins, draws and losses depending on time and with a fixed size of "States to be searched" set = 5.

To prove the previous results we have measured the average number of turns it takes to finish a game. We expect that the average number of turns for both roles (XPLAYER and OPLAYER) will decrease with increasing time. Another behavior that we expect is that when playing as an OPLAYER our player will have a higher average number of turns in a match than XPLAYER. There are 2 reasons for it. One is the fact that the OPLAYER starts second and thus has +1 turn. The second reason is that the player must sometimes make a move that does not help her win but is necessary to block the opponent from winning. If the player cannot win, she will drag out the game as long as possible in her attempt to achieve a draw.

The results of this experiment are shown on Figure 25. We can see that our expectations were correct and the average number of turns decreases with increasing computation time for both players. This confirms our hypothesis that with increasing time our player (either as XPLAYER or OPLAYER) is able to finish the game faster by winning as shown in Figure 24.

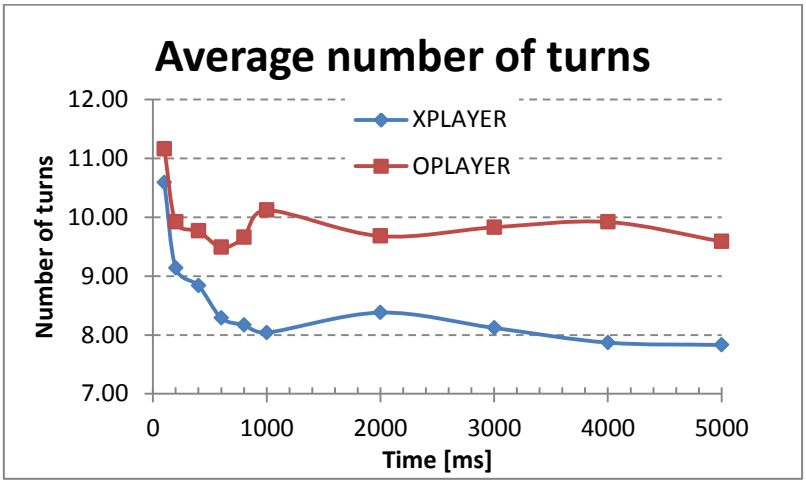


Figure 25: Average number of turns it takes to finish a match depending on the role of player.

Below, on Table 4 are the values measured during our experiments. We have highlighted in bold the most important values and that is the average score per match. We can see that it is increasing with the increasing time.

5 states - fixed	100	200	400	600	800	1000	2000	3000	4000	5000
wins [%]	0,62	0,75	0,76	0,79	0,81	0,77	0,79	0,80	0,80	0,87
losses [%]	0,29	0,17	0,14	0,15	0,13	0,13	0,11	0,10	0,08	0,06
draws [%]	0,10	0,08	0,10	0,06	0,07	0,10	0,11	0,10	0,12	0,08
Average score per match	**66,50**	**79,17**	**81,17**	**82,00**	**83,75**	**81,83**	**84**	**85**	**86**	**90,50**
Average score for xplayer	82,50	90,67	91,67	94	95,50	95,00	93,50	96,00	95,50	97,50
Average score for oplayer	50,50	67,67	70,67	70	72	68,67	74,50	74,00	76,50	83,50
Avg number of turns xplayer	10,59	9,14	8,84	8,29	8,17	8,04	8,38	8,12	7,87	7,83
Avg number of turns oplayer	10,16	8,92	8,77	8,49	8,66	9,12	8,68	8,83	8,92	8,59

Table 4: Measured values with variable time and fixed size of "States to be searched" set

From all the above figures and table we can see that the time given to our player influences her performance. The more time we provide the better TIIGR performs. From the

experiments we can also see that the time of $2000\ ms$ is a good balance between time and results.

In the previous experiments we were using random player as our opponent. However to see how TIIGR would perform against a better opponent than a random player we have pitched her against herself. We expect that the average score of TIIGR will be severely decreased compared to the average score which TIIGR obtained against a random player. With increasing time the performance of our player will still increase but in this experiment the average score will not increase because both players are improving. What we should see is that at some point the average score will stop changing and become constant. The results of this experiment are shown on

Figure 26 Figure 26. We show only the change in the average score of XPLAYER since it is similar to the development of average score for OPLAYER. As we have expected, playing against herself has influenced the average score. It is clear that the average score has decreased compared to playing against random player. The maximum difference is around 28 points.

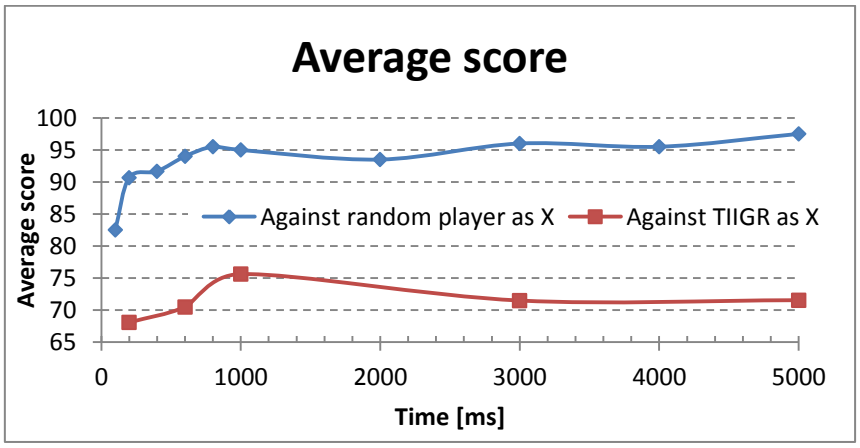


Figure 26: Comparison of TIIGR's performance against random player and against herself.

Now that we have presented our findings we move to experiments with a variable size of "States to be searched" set.

7.2. Variable "States to be searched" size

In the following set of experiments we are trying to prove that with the increasing size of "States to be searched" the performance of our player also increases. We did several tests to prove this hypothesis with different number of "States to be searched" (1, 3 and 5 states) and several different times ($0 < Time \leq 5000$). In these experiments we collected the values of average score and the win/draw/lose ratio.

On Figure 27 there are total of 6 graphs showing the average score of XPLAYER and OPLAYER depending on the size of "States to be searched" set. The results support our hypothesis. The more states we include in the "States to be searched" set the better our player is able to estimate which state she is in and thus her performance increases. We can notice that when

playing as XPLAYER player's performance is not influenced much by the number of states, if the number is larger or equal to 2. On the other hand, the OPLAYER receives a slight increase in her performance when we increased the number of states in "States to be searched" set.

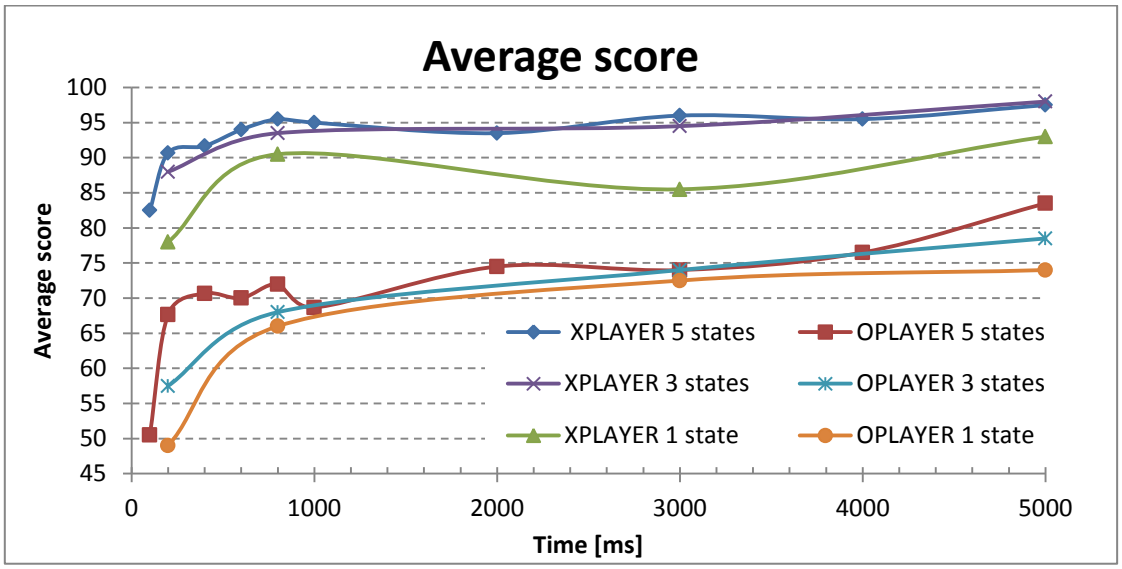


Figure 27: Average XPLAYER and OPLAYER score for 3 different sizes of "States to be searched" set.

The following Figure 28 shows the win/loss ratio of our player based on the different size of "States to be searched" set. The results we can see on the win ratio are as expected. The more states we add to the "States to be searched" set, the higher winning ratio we achieve. If we decrease the size of "States to be searched" set then we can expect the loss ratio to increase.

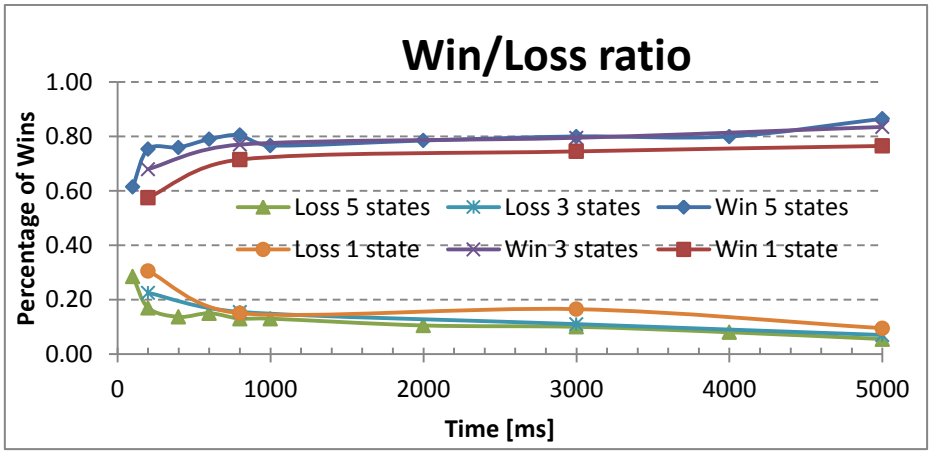


Figure 28: Win percentage of our player 3 different sizes of "States to be searched" set.

Below on Table 5 are the values measured during our experiments.

Time [ms]	200			800			3000			5000		
# of states	1	3	5	1	3	5	1	3	5	1	3	5
wins [%]	0,58	0,68	0,75	0,72	0,77	0,81	0,75	0,8	0,8	0,77	0,84	0,87
losses [%]	0,31	0,23	0,17	0,15	0,16	0,13	0,17	0,11	0,1	0,10	0,07	0,06
draws [%]	0,12	0,095	0,08	0,135	0,08	0,07	0,09	0,10	0,1	0,14	0,10	0,08
Average score per match	63,5	72,75	79,17	78,25	80,75	83,75	79	84,3	85	83,5	88,3	90,5
Average score for xplayer	78	88	90,67	90,5	93,5	95,5	85,5	94,5	96	93	98	97,5
Average score for oplayer	49	57,5	67,67	66	68	72	72,5	74	74	74	78,5	83,5

Table 5: Measured values with variable time and variable size of "States to be searched" set

We can see that in the game of Latent Tic-Tac-Toe TIIGR's performance increases with increasing time and the number of "States to be searched" set. From the above experiments it is clear that 3 states in the "States to be searched" set are a good balance between performance (increasing the size of the set to 5 did not bring significant improvement) and time.

7.3. Variable C

In the following experiments we want to show what effect the parameter C in MC-UCT has on the performance of our player. What can we expect with the change of parameter C? Let us discuss the equation that is used in selection part of MC-UCT:

$$v_i + C * \sqrt{\frac{\ln N}{n_i}} \tag{14}$$

If we select a small C compared to the v_i then the second part of the equation will have no influence on the selection and the MC-UCT will become a greedy algorithm (we will prefer exploitation over exploration). On the other hand, if we set the C too high then our player will start to behave as a random player. This is caused by the fact that we will always explore the game tree, and when the MC-UCT has to select the best move, all of the moves will have the same confidence value and thus a random move will be chosen. Consequently, we expect that for too small and too great values of parameter C the average score will be smaller than for values of C between those 2 extremes. This is a game specific parameter and thus a different game will have a different optimal value of C. On Figure 29 we present the results of our experiments with parameter C.

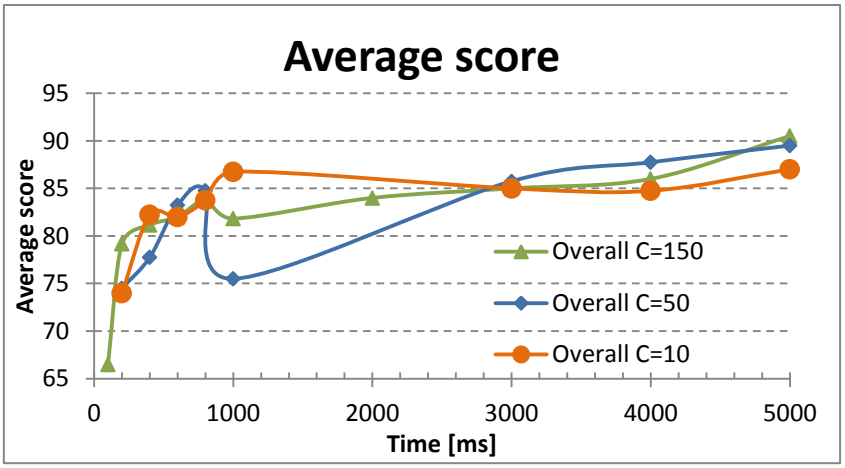


Figure 29: Average score of our player depending on the C parameter.

As we can see on Figure 29 for short times it is hard to tell which value of C is better. This is caused by the fact that for small C the PIMC behaves as a depth-first search algorithm thus it focuses the search to the first good result it finds. This might pay off for short times. However, when the time grows we can see that value of C starts to get worse compared to the bigger C values. We can see that for $C = 150$ we obtain a monotonically increasing function from $Time = 1000ms$. Therefore from the 3 tested values of C the highest one performs the best.

Now let's continue with experiments done on different games.

7.4. Simple card game

Different tests were performed on a simple card game. In this game, there are 3 cards (Jack, Queen and King). Each of the players receives 1 card from dealer (who is represented by the random player in GDL-II) and the 3rd card is discarded without anyone seeing it. Then each player can either bet or fold (none of the players sees what the opponent did – it's a simultaneous decision). If both players bet, then the one with the highest card wins (receives the reward of 100 and the opponent 0). If one of them folds and the other bets, then the one who bets receives a reward of 75 and the one who folded receives 25. If both fold they both receive 50. Let's have a look on a log from 1 of the games we played. The log below is self-explanatory and needs no further discussion (the bold lines are our comments added to the log generated by TIIGR).

BEGINNING OF LOG

// In this game we play as the second player
// This command starts the match
`Command: (PLAY scg1 NIL)`

// In this moment the dealer is dealing cards to players, therefore our only legal move is NOOP
`Response to game server:NOOP`
// We received the card we were dealt
`Command: (PLAY scg1 (QUEEN))`

// Now we check in which states we can be

// If the dealer (RANDOM) dealt JACK to the first player and QUEEN to us, then we would receive a sees term that we received a QUEEN card.

Get sees terms => Moves: (DOES RANDOM (DEAL JACK QUEEN)) (DOES P1 NOOP) (DOES P2 NOOP) , Size: 3

// Therefore this state is accepted as one that we can be in

State was ACCEPTED!

// If the dealer dealt JACK to the first player and QUEEN to us, then we would NOT receive a sees term that we received a KING card.

INFO [Thread-6] (ReasonerAdapter.java:42) - Get sees terms => Moves: (DOES RANDOM (DEAL JACK KING)) (DOES P1 NOOP) (DOES P2 NOOP) , Size: 3

// Thus this state is rejected

State was REJECTED!

// The above is done for all combinations of dealt cards – QUEEN, JACK

INFO [Thread-6] (ReasonerAdapter.java:42) - Get sees terms => Moves: (DOES RANDOM (DEAL QUEEN JACK)) (DOES P1 NOOP) (DOES P2 NOOP) , Size: 3

State was REJECTED!

// All card combinations – QUEEN, KING; …; KING, QUEEN

⋮

// The number of states we can be in is 2 – either our opponent got a jack or a king.

INFO [Thread-6] (TestMyPlayer.java:209) - States we can be in = 2

// Now it is player 1's turn to decide if she goes all in or folds we have only 1 action - NOOP

```
Response to game server: NOOP
```

// This command shows that opponent has decided what to do and it is now our turn.

```
Command: (PLAY scg1 (YOURTURN))
```

// Now we will check again for all possible states our player can be in

INFO [Thread-9] (ReasonerAdapter.java:42) - Get sees terms => Moves: (DOES RANDOM NOOP) (DOES P1 ALLIN) (DOES P2 NOOP) , Size: 3

State was ACCEPTED!

INFO [Thread-9] (ReasonerAdapter.java:42) - Get sees terms => Moves: (DOES RANDOM NOOP) (DOES P1 FOLD) (DOES P2 NOOP) , Size: 3

State was ACCEPTED!

INFO [Thread-9] (ReasonerAdapter.java:42) - Get sees terms => Moves: (DOES RANDOM NOOP) (DOES P1 ALLIN) (DOES P2 NOOP) , Size: 3

State was ACCEPTED!

INFO [Thread-9] (ReasonerAdapter.java:42) - Get sees terms => Moves: (DOES RANDOM NOOP) (DOES P1 FOLD) (DOES P2 NOOP) , Size: 3

State was ACCEPTED!

// The number of states we can be in is 4 – for each of the possible cards dealt opponent could have either played all in or folded.

INFO [Thread-9] (TestMyPlayer.java:209) - States we can be in = 4

// Now we run MC-UCT on each of the possible states (since our limit is 5 and we got only 4)

```
MC started
```

// In this state opponent has a KING. If we (p2) were to go all in the Average reward we would obtain is 0 (because we always lose).

```
Action: (random noop) (p1 noop) (p2 allin) # of times visited(confidence value):420 Average reward = 0.0
```

// If we were to fold our Average reward would be 25. As we can see the confidence value is much higher than in the previous all in action.

```
Action: (random noop) (p1 noop) (p2 fold) # of times visited(confidence value):293152 Average reward = 25.0
```

// Therefore if we knew we are in this specific state (opponent has king and has gone all in) our action would be fold.

```
MC started
Action: (random noop) (p1 noop) (p2 allin) # of times visited(confidence value):302540 Average reward = 75.0
Action: (random noop) (p1 noop) (p2 fold) # of times visited(confidence value):422 Average reward = 50.0

MC started
Action: (random noop) (p1 noop) (p2 allin) # of times visited(confidence value):299835 Average reward = 100.0
Action: (random noop) (p1 noop) (p2 fold) # of times visited(confidence value):50 Average reward = 25.0

MC started
Action: (random noop) (p1 noop) (p2 allin) # of times visited(confidence value):238616 Average reward = 75.0
Action: (random noop) (p1 noop) (p2 fold) # of times visited(confidence value):410 Average reward = 50.0
```

// Because we do not know in which state we are the choosing of action is harder. At this moment we were choosing the best action as the action with the highest confidence value. Thus we select ALLIN action. Even if we were choosing the best average value we would go all in.

```
Response to game server: ALLIN
```

// Command that informs us that the match is finished

```
Command: (STOP scg1 ())
```

END OF LOG

We played 4175 matches and the results are presented in Table 6. The settings of TIIGR were $t = 400ms$ with 5 states to be searched each turn. As we can see from the log of this game, player can never have more than 4 states in her "States to be searched" set. The specified number of "States to be searched" is the maximum number of states that are searched. If there are fewer states, then the player searches through all available states. As we can see from the results in Table 6 TIIGR performs better than random player whom she was facing.

Results	TIIGR's action	Random player's action	
TIIGR won	All in	All in	825
Opponent folded	All in	Fold	1067
Both folded	Fold	Fold	1038
TIIGR folded	Fold	All in	1055
Opponent won	All in	All in	190
Average score			57,677

Table 6: Results of our simple card game

Let's analyze the results. We have a probability of 1/3 that we receive J, Q or K. It is easy to see what action we take if we receive J or K. For J it is to fold and for K it is to go all in in all cases. Not so clear is the situation with Q. If we have Q our opponent can have either J or K and he can go all in or fold. Thus we have 4 possible situations:

1. Opponent has K and goes all in – in this situation our best response is to fold because we have no chance of winning
2. Opponent has K and folds – here it would be best to go all in
3. Opponent has J and goes all in – in this case it is also best to go all in
4. Opponent has J and folds – again in this case it is best for us to go all in

Since our player considers that she can be in all states with the same probability (we apply the overconfident approach in selecting the possible states that TIIGR can be in – as discussed in Section 6.2.3) we obtain that the best action is to go all in.

We see in Table 6 that 2/3 of the time we played all in (in case we have Q or K) and 1/3 of the time we folded. If we changed the selection of nodes to be in to the paranoid approach we would see that 2/3 of the time our player folded and 1/3 he played all in.

7.5. Liar's dice

The last set of experiments was run on a game called Liar's dice that is available on the Dresden GGP Server (Dresden GGP server, 2010) under the name Meier. This is also a simple game. The game starts with one player rolling 2 dice. She sees the roll and claims some value (she can either tell the truth or bluff). Her opponent does not see the roll and has 2 options. Either claim that the first player is bluffing which ends the game. Or she can roll her own dice and claim that she rolled a greater value than the first player. And this goes on until one of the players calls the other roll a bluff. If a player 1 calls a bluff and player 2 has the values she claimed, then player 1 is scored 0 and player 2's reward is 100. If on the other hand player 1 does not have the values she claimed then she loses and receives a reward of 0 and her opponent receives 100. Again, as in the previous simple card game let's have a look on a log from 1 of the games we played. The log below is commented and shows what occurs during the gameplay (the bold lines are our comments added to the log generated by TIIGR).

BEGINNING OF LOG

// Command to start the match. First player 1 is rolling dice (that is us)

```
Command: (PLAY meierid1 NIL)
```

// The dice rolling is done by the random player thus our only legal move is NOOP

```
Response to game server: NOOP
```

// We see that we have rolled 1 and 3

```
Command: (PLAY meierid1 ((MY_DICE 1 3) (DOES P1 NOOP) (DOES P2 NOOP) ))
```

// Now we check which states we can be in.

INFO [Thread-6] (TestMyPlayer.java:402) - Getting sees terms for node Depth[0] := Moves No Move Info => ((previous_claimed_values 0 0) (rolling_for p1)) and moves [random (roll p1 1 1), p1 noop, p2 noop]

State was REJECTED!

INFO [Thread-6] (TestMyPlayer.java:402) - Getting sees terms for node Depth[0] := Moves No Move Info => ((previous_claimed_values 0 0) (rolling_for p1)) and moves [random (roll p1 1 2), p1 noop, p2 noop]

State was REJECTED!

// Only the state where we have rolled 1 3 is accepted because the sees terms from server match our generated sees terms.

INFO [Thread-6] (TestMyPlayer.java:402) - Getting sees terms for node Depth[0] := Moves No Move Info => ((previous_claimed_values 0 0) (rolling_for p1)) and moves [random (roll p1 1 3), p1 noop, p2 noop]

State was ACCEPTED!

INFO [Thread-6] (TestMyPlayer.java:402) - Getting sees terms for node Depth[0] := Moves No Move Info => ((previous_claimed_values 0 0) (rolling_for p1)) and moves [random (roll p1 6 3), p1 noop, p2 noop]

State was REJECTED!

INFO [Thread-6] (TestMyPlayer.java:402) - Getting sees terms for node Depth[0] := Moves No Move Info => ((previous_claimed_values 0 0) (rolling_for p1)) and moves [random (roll p1 6 4), p1 noop, p2 noop]

State was REJECTED!

INFO [Thread-6] (TestMyPlayer.java:402) - Getting sees terms for node Depth[0] := Moves No Move Info => ((previous_claimed_values 0 0) (rolling_for p1)) and moves [random (roll p1 6 5), p1 noop, p2 noop]

State was REJECTED!

INFO [Thread-6] (TestMyPlayer.java:402) - Getting sees terms for node Depth[0] := Moves No Move Info => ((previous_claimed_values 0 0) (rolling_for p1)) and moves [random (roll p1 6 6), p1 noop, p2 noop]

State was REJECTED!

// We can be only in 1 state and that is the state where we have rolled 1 3

States we can be in = 1

// And it is our turn so run a MC on the state we can be in and select the best action to play

```
MC started
Action random noop p1 (claim 3 1) p2 noop , # of times visited(confidence value): 186 Average reward = 88.70967741935483
```

```
Action random noop p1 (claim 3 2) p2 noop , # of times visited(confidence
value): 16 Average reward = 25.0
Action random noop p1 (claim 4 1) p2 noop , # of times visited(confidence
value): 14 Average reward = 14.285714285714286
Action random noop p1 (claim 4 2) p2 noop , # of times visited(confidence
value): 14 Average reward = 14.285714285714286
Action random noop p1 (claim 4 3) p2 noop , # of times visited(confidence
value): 14 Average reward = 14.285714285714286
Action random noop p1 (claim 5 1) p2 noop , # of times visited(confidence
value): 14 Average reward = 14.285714285714286
Action random noop p1 (claim 5 2) p2 noop , # of times visited(confidence
value): 14 Average reward = 14.285714285714286
Action random noop p1 (claim 5 3) p2 noop , # of times visited(confidence
value): 14 Average reward = 14.285714285714286
Action random noop p1 (claim 5 4) p2 noop , # of times visited(confidence
value): 15 Average reward = 20.0
Action random noop p1 (claim 6 1) p2 noop , # of times visited(confidence
value): 12 Average reward = 8.333333333333334
Action random noop p1 (claim 6 2) p2 noop , # of times visited(confidence
value): 15 Average reward = 20.0
Action random noop p1 (claim 6 3) p2 noop , # of times visited(confidence
value): 14 Average reward = 14.285714285714286
Action random noop p1 (claim 6 4) p2 noop , # of times visited(confidence
value): 14 Average reward = 14.285714285714286
Action random noop p1 (claim 6 5) p2 noop , # of times visited(confidence
value): 15 Average reward = 20.0
Action random noop p1 (claim 1 1) p2 noop , # of times visited(confidence
value): 14 Average reward = 14.285714285714286
Action random noop p1 (claim 2 2) p2 noop , # of times visited(confidence
value): 14 Average reward = 14.285714285714286
Action random noop p1 (claim 3 3) p2 noop , # of times visited(confidence
value): 12 Average reward = 8.333333333333334
Action random noop p1 (claim 4 4) p2 noop , # of times visited(confidence
value): 15 Average reward = 20.0
Action random noop p1 (claim 5 5) p2 noop , # of times visited(confidence
value): 14 Average reward = 14.285714285714286
Action random noop p1 (claim 6 6) p2 noop , # of times visited(confidence
value): 14 Average reward = 14.285714285714286
Action random noop p1 (claim 2 1) p2 noop , # of times visited(confidence
value): 11 Average reward = 0.0
```

// From MC we get that the best action is to tell the truth and claim that we rolled 3 1

```
Response to game server: (CLAIM 3 1)
```

// Now opponent saw what are the values that we claim and we filter our states.

```
Command: (PLAY meierid1 ((DOES P1 (CLAIM 3 1)) (DOES P2 NOOP) ))
```

INFO [Thread-8] (TIIGRPlayer.java:183) – Didn't play this move (claim 3 2), my move was (CLAIM 3 1). State REJECTED!

⋮

INFO [Thread-8] (TIIGRPlayer.java:183) – Didn't play this move (claim 6 6), my move was (CLAIM 3 1). State REJECTED!

INFO [Thread-8] (TIIGRPlayer.java:181) – IT IS THE MOVE I PLAYED (claim 3 1) myMove = (CLAIM 3 1). CHECKING SEES TERMS!

INFO [Thread-8] (TIIGRPlayer.java:419) – Getting sees terms for node Depth[1] := Moves[random (roll p1 2 1), p1 noop, p2 noop] => ((previous_claimed_values 0 0) (claiming p1) (has_dice p1 2 1)) and moves [random noop, p1 (claim 2 1), p2 noop]

INFO [Thread-8] (TIIGRPlayer.java:211) – was ACCEPTED!

```
INFO [Thread-8] (TIIGRPlayer.java:226) - Stavu, ve kterych muzu byt = 1
```

// We made our claim and now it is opponents turn to select if she will roll our say that we bluff. Thus we have only 1 legal action NOOP.

```
Response to game server: NOOP
```

// The game has ended. From the sees terms in the STOP message we see that the opponent did not choose to roll her dice but chose to call our claimed value a bluff. Because we did not bluff we have won.

```
Command: (STOP meierid1 ((DOES P1 NOOP) (DOES P2 YOU_BLUFF)))
```

END OF LOG

The results for this game are presented in Table 7.

Number of times won	194
Number of times lost	6
average score	97

Table 7: Results of meier game

As we can see our player performs extremely well in this game. These games were all played against a random player and thus the results can be easily explained. If we are playing against a random player and the opponent is rolling dice first then our player always calls a bluff. Since the probability is $\frac{35}{36}$ that the opponent is bluffing we win most of the times. If we are rolling the dice first we always claim the rolled value. Random player chooses 50% of the time to call a bluff which results in our victory. If he rolls the dice and claims any value we call his value a bluff. The probability of the opponent bluffing is again $\frac{35}{36}$. Therefore it is no surprise that our player performs so well.

In this Chapter we have tested TIIGR on all existing GDL-II games. By our experiments we have proven that performance of our player increases with increasing time and increasing size of “States to be searched” set. In the next Chapter we conclude our work and present options on how to improve TIIGR.

8 Conclusions

In this thesis we have given a review of the existing state-of-the-art techniques for playing imperfect information games. Based on this analysis we have chosen MC-UCT as the most promising approach for creating an imperfect information general game playing agent.

With an approach chosen we have created TIIGR, one of the first general game playing agents compatible with GDL-II. Due to the lack of games written in GDL-II we have created our own game of Latent Tic-Tac-Toe and rewritten a simple card game. Then on these games and the game of Meier (Liar's dice) that is available on Dresden GGP server we have proven that our agent is able to play all of the games, be it card, board or a guessing game. The performance of our player is tested and thoroughly studied on several thousands of games played against random opponent and against herself.

8.1. Evaluation

We will list the requirements of our thesis and answer how it was accomplished.

1. **Our goal was to study the state-of-the-art techniques for playing specific imperfect information games, and create a review of existing approaches that can be used for playing imperfect information games. Then we had to analyze their requirements and discuss their usability in a GGP agent.**

   We have created an overview of existing approaches in Chapter 3 - General game playing algorithms and later in Chapter 5 - Discussion we have analyzed and discussed the usability of 3 approaches: Counterfactual regret minimization, Information set search and Perfect information sampling Monte Carlo.

2. **Another objective was to get familiar with the rules, requirements and main principles of the General Game Playing competition organized at the AAAI conference. In particular, we had to study the new variant of Game Description Language (GDL-II), which allows specifying imperfect information games.**

   We have introduced the General Game Playing competition, its rules and requirements in Chapter 4 - General Game Playing. We have introduced both GDL and GDL-II and have created our own game in this language (Latent Tic-Tac-Toe – which we provided to the community and it can be downloaded from Dresden GGP Server) and coded a simple card game that was described in (Thielscher, 2010).

3. **The last goal was to design, implement and experimentally evaluate a GGP agent. The agent does not have to fully support GDL-II, but it should be able to play multiple formally specified imperfect information games.**

   This goal was met by creating TIIGR, one of the first general game playing agents that fully support GDL-II. We have performed various experiments on all existing GDL-II games. The experiments have proved our hypothesis that with the increasing time and increasing size of "States to be searched" set the performance of TIIGR improves. For more details and discussions of our experiments see Chapter 7.

8.2. Future Work

During the experiments we discovered several ways how TIIGR can be improved.

1. The JavaProver reasoner that we use is 4 times slower than PrologProver reasoner. If we added the support of GDL-II to the PrologProver and use it instead of JavaProver reasoner, the performance of our player would be dramatically boosted since she will need less time to achieve the same results.
2. Fine tune the C parameter of MC-UCT for general games. The C parameter depends on a specific game. If we compare more games and the influence of parameter C on the results, we expect to find an interval in which MC-UCT performs better for every game.
3. Implement the information set search approach and run MC-UCT on the information set tree instead of on the perfect information state tree. Then compare the results of these 2 approaches. If the information set search would provide good results then make it part of our player.
4. A useful add-on to our program would be to add a mechanism to check if a game we are going to play suffers from the non-locality or strategy fusion errors. And based on the information we can decide if we will use MC-UCT or another approach to play the specified game.

9 Bibliography

Dresden GGP server. (2010). Retrieved December 26, 2010, from Dresden GGP server: http://euklid.inf.tu-dresden.de:8180/ggpserver/

Finnsson, H. (2007). CADIA-Player: A General Game Playing Agent. *Master Thesis*. Reykjavík University.

Genesereth, M., Love, N., & Pell, B. (2005). General Game Playing: Overview of the AAAI Competition. *AI magazine, 26*(2), pp. 62-72.

Chaslot, G., Bakkes, S., Szita, I., & Spronck, P. (2008). Monte-Carlo tree search: A new framework for game AI. *BNAIC 2008* (pp. 389-390). University of Twente.

Johanson, M. B. (2007). Robust Strategies and Counter-Strategies: Building a Champion Level Computer. *Master Thesis*. Edmonton: University of Alberta.

Keller, I. (2008). Retrieved April 11, 2011, from Palamedes IDE: http://palamedes-ide.sourceforge.net/index.html

Keller, I. (2009). Palamedes: A General Game Playing IDE. *Student Thesis*. Dresden University of Technology.

Keller, I. (2010). Automatic Generation of Game Component Visualization. *Master Thesis*. Dresden University of Technology.

Kocsis, L., & Szepesvári, C. (2006). Bandit Based Monte Carlo Planning. *Machine Learning: ECML 2006*, *Lecture Notes in Artificial Intelligence 4212*, pp. 282-293.

Long, J. R., Sturtevant, N. R., Buro, M., & Furtak, T. (2010). Understanding the Success of Perfect Information Monte Carlo Sampling in Game Tree Search. *AAAI-10*, (pp. 134-140). Atlanta.

Love, N., Hinrichs, T., Haley, D., Schkufza, E., & Genesereth, M. (2008). General game playing: Game description language specification. *Technical Report LG-2006-01*. Stanford University.

Luckhardt, C. A., & Irani, K. B. (1986). An Algorithmic Solution of n-person Games. *AAAI-86*, (pp. 158-162).

Méhat, J., & Cazenave, T. (2010). *Ary, a general game playing program.* Paris.

Osborne, M. J., & Rubinstein, A. (1994). *A Course in Game Theory.* The MIT Press.

Parker, A., Nau, D., & Subrahmanian, V. (2006). Overconfidence or Paranoia? Search in Imperfect-Information Games. *The Twenty-first National Conference on Artificial Intelligence*, (pp. 1045-1050). Boston.

Parker, A., Nau, D., & Subrahmanian, V. S. (2005). Game-Tree Search with Combinatorially Large Belief States. *IJCAI*, (pp. 254-259).

Parker, A., Nau, D., & Subrahmanian, V. S. (2010). Paranoia versus Overconfidence in Imperfect Information Games. In *Heurisitcs, Probability and Causality: A Tribute to Judea Pearl* (pp. 63-87). College Publication.

Polak, B. (2007). *Game Theory.* Retrieved January 27, 2010, from Open Yale Courses: http://oyc.yale.edu

Rasmusen, E. (2006). *Games and Information: An Introduction to Game Theory* (4th ed.). Blackwell Publishers.

Russell, S., & Norvig, P. (2003). *Artificial Intelligence: A Modern Approach* (2nd ed.). Prentice Hall.

Shoham, Y., & Leyton-Brown, K. (2010). *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations.* Cambridge University Press.

Schiffel, S., & Thielscher, M. (2007). Fluxplayer: A Successful General Game Player. *AAAI-07* (pp. 1191-1196). AAAI Press.

Sturtevant, N. R. (2008). An Analysis of UCT in Multi-player Games. *ICGA Journal, 31*(4), 195–208.

Thielscher, M. (2010). A General Game Description Language for Incomplete Information Games. *AI magazine*, pp. 994-999.

Zinkevich, M., Johanson, M., Piccione, C., & Bowling, M. (2008). Regret Minimization in Games with Incomplete Information. *NIPS* (pp. 1729-1736). MIT Press.

10 Appendix A

GDL Example of 2 player Bomberman game downloaded from (Dresden GGP server, 2010):

```
(role bomberman)
(role bomberwoman)
(init (location bomberman 1 1))
(init (location bomberwoman 8 8))
(init (blockednorth 2 1))
(init (blockednorth 4 1))
(init (blockednorth 5 1))
(init (blockednorth 7 1))
(init (blockednorth 2 2))
(init (blockednorth 4 2))
(init (blockednorth 5 2))
(init (blockednorth 7 2))
(init (blockednorth 2 3))
(init (blockednorth 4 3))
(init (blockednorth 5 3))
(init (blockednorth 7 3))
(init (blockednorth 2 5))
(init (blockednorth 4 5))
(init (blockednorth 5 5))
(init (blockednorth 7 5))
(init (blockednorth 2 6))
(init (blockednorth 4 6))
(init (blockednorth 5 6))
(init (blockednorth 7 6))
(init (blockednorth 2 7))
(init (blockednorth 4 7))
(init (blockednorth 5 7))
(init (blockednorth 7 7))
(init (blockednorth 1 8))
(init (blockednorth 2 8))
(init (blockednorth 3 8))
(init (blockednorth 4 8))
(init (blockednorth 5 8))
(init (blockednorth 6 8))
(init (blockednorth 7 8))
(init (blockednorth 8 8))
(init (blockedeast 1 2))
(init (blockedeast 1 4))
(init (blockedeast 1 5))
(init (blockedeast 1 7))
(init (blockedeast 2 2))
(init (blockedeast 2 4))
(init (blockedeast 2 5))
(init (blockedeast 2 7))
(init (blockedeast 3 2))
(init (blockedeast 3 4))
```

```
(init (blockedeast 3 5))
(init (blockedeast 3 7))
(init (blockedeast 5 2))
(init (blockedeast 5 4))
(init (blockedeast 5 5))
(init (blockedeast 5 7))
(init (blockedeast 6 2))
(init (blockedeast 6 4))
(init (blockedeast 6 5))
(init (blockedeast 6 7))
(init (blockedeast 7 2))
(init (blockedeast 7 4))
(init (blockedeast 7 5))
(init (blockedeast 7 7))
(init (blockedeast 8 1))
(init (blockedeast 8 2))
(init (blockedeast 8 3))
(init (blockedeast 8 4))
(init (blockedeast 8 5))
(init (blockedeast 8 6))
(init (blockedeast 8 7))
(init (blockedeast 8 8))
(init (step 1))
(<= (legal ?char (move ?dir))
    (role ?char)
    (true (location ?char ?x ?y))
    (legalstep ?dir ?x ?y))
(<= (legal ?char dropbomb)
    (role ?char))
(<= (next (blockednorth ?x ?y))
    (true (blockednorth ?x ?y)))
(<= (next (blockedeast ?x ?y))
    (true (blockedeast ?x ?y)))
(<= (next (location ?char ?x2 ?y2))
    (role ?char)
    (true (location ?char ?x1 ?y1))
    (does ?char (move ?dir))
    (nextcell ?dir ?x1 ?y1 ?x2 ?y2))
(<= (next (location ?char ?x ?y))
    (role ?char)
    (true (location ?char ?x ?y))
    (does ?char dropbomb))
(<= (next (location bomb3 ?x ?y))
    (role ?char)
    (true (location ?char ?x ?y))
    (does ?char dropbomb))
(<= (next (location bomb2 ?x ?y))
    (true (location bomb3 ?x ?y)))
(<= (next (location bomb1 ?x ?y))
    (true (location bomb2 ?x ?y)))
(<= (next (location bomb0 ?x ?y))
```

```
    (true (location bomb1 ?x ?y)))
(<= (next (location fire ?xf ?y))
    (true (location bomb0 ?xb ?y))
    (not (true (blockedeast ?xb ?y)))
    (index ?xf))
(<= (next (location fire ?x ?yf))
    (true (location bomb0 ?x ?yb))
    (not (true (blockednorth ?x ?yb)))
    (index ?yf))
(<= (next (step ?n++))
    (true (step ?n))
    (succ ?n ?n++))
(<= terminal
    bombermanburned)
(<= terminal
    bomberwomanburned)
(<= terminal
    timeout)
(<= (goal bomberman 0)
    (not timeout)
    (not bombermanburned)
    (not bomberwomanburned))
(<= (goal bomberman 0)
    bombermanburned
    (not bomberwomanburned))
(<= (goal bomberman 50)
    bombermanburned
    bomberwomanburned)
(<= (goal bomberman 50)
    timeout
    (not bombermanburned)
    (not bomberwomanburned))
(<= (goal bomberman 100)
    (not bombermanburned)
    bomberwomanburned)
(<= (goal bomberwoman 0)
    (not timeout)
    (not bombermanburned)
    (not bomberwomanburned))
(<= (goal bomberwoman 0)
    (not bombermanburned)
    bomberwomanburned)
(<= (goal bomberwoman 50)
    bombermanburned
    bomberwomanburned)
(<= (goal bomberwoman 50)
    timeout
    (not bombermanburned)
    (not bomberwomanburned))
(<= (goal bomberwoman 100)
    bombermanburned
```

```
    (not bomberwomanburned))
(<= (legalstep north ?x ?y)
    (++ ?y ?ynew)
    (cell ?x ?ynew)
    (not (blocked ?x ?y ?x ?ynew)))
(<= (legalstep south ?x ?y)
    (-- ?y ?ynew)
    (cell ?x ?ynew)
    (not (blocked ?x ?y ?x ?ynew)))
(<= (legalstep east ?x ?y)
    (++ ?x ?xnew)
    (cell ?xnew ?y)
    (not (blocked ?x ?y ?xnew ?y)))
(<= (legalstep west ?x ?y)
    (-- ?x ?xnew)
    (cell ?xnew ?y)
    (not (blocked ?x ?y ?xnew ?y)))
(<= (legalstep nowhere ?x ?y)
    (cell ?x ?y))
(<= (nextcell north ?x ?y ?x ?ynew)
    (index ?x)
    (++ ?y ?ynew))
(<= (nextcell south ?x ?y ?x ?ynew)
    (index ?x)
    (-- ?y ?ynew))
(<= (nextcell east ?x ?y ?xnew ?y)
    (index ?y)
    (++ ?x ?xnew))
(<= (nextcell west ?x ?y ?xnew ?y)
    (index ?y)
    (-- ?x ?xnew))
(<= (nextcell nowhere ?x ?y ?x ?y)
    (cell ?x ?y))
(<= (blocked ?x ?y1 ?x ?y2)
    (true (blockednorth ?x ?y1))
    (++ ?y1 ?y2))
(<= (blocked ?x ?y2 ?x ?y1)
    (true (blockednorth ?x ?y1))
    (++ ?y1 ?y2))
(<= (blocked ?x1 ?y ?x2 ?y)
    (true (blockedeast ?x1 ?y))
    (++ ?x1 ?x2))
(<= (blocked ?x2 ?y ?x1 ?y)
    (true (blockedeast ?x1 ?y))
    (++ ?x1 ?x2))
(<= (distinctcell ?x1 ?y1 ?x2 ?y2)
    (cell ?x1 ?y1)
    (cell ?x2 ?y2)
    (distinct ?x1 ?x2))
(<= (distinctcell ?x1 ?y1 ?x2 ?y2)
    (cell ?x1 ?y1)
```

```
    (cell ?x2 ?y2)
    (distinct ?y1 ?y2))
(<= bombermanburned
    (true (location bomberman ?x ?y))
    (true (location fire ?x ?y)))
(<= bomberwomanburned
    (true (location bomberwoman ?x ?y))
    (true (location fire ?x ?y)))
(<= timeout
    (true (step 50)))
(index 1)
(index 2)
(index 3)
(index 4)
(index 5)
(index 6)
(index 7)
(index 8)
(cell 1 8)
(cell 2 8)
(cell 3 8)
(cell 4 8)
(cell 5 8)
(cell 6 8)
(cell 7 8)
(cell 8 8)
(cell 1 7)
(cell 2 7)
(cell 3 7)
(cell 4 7)
(cell 5 7)
(cell 6 7)
(cell 7 7)
(cell 8 7)
(cell 1 6)
(cell 2 6)
(cell 3 6)
(cell 4 6)
(cell 5 6)
(cell 6 6)
(cell 7 6)
(cell 8 6)
(cell 1 5)
(cell 2 5)
(cell 3 5)
(cell 4 5)
(cell 5 5)
(cell 6 5)
(cell 7 5)
(cell 8 5)
(cell 1 4)
```

```
(cell 2 4)
(cell 3 4)
(cell 4 4)
(cell 5 4)
(cell 6 4)
(cell 7 4)
(cell 8 4)
(cell 1 3)
(cell 2 3)
(cell 3 3)
(cell 4 3)
(cell 5 3)
(cell 6 3)
(cell 7 3)
(cell 8 3)
(cell 1 2)
(cell 2 2)
(cell 3 2)
(cell 4 2)
(cell 5 2)
(cell 6 2)
(cell 7 2)
(cell 8 2)
(cell 1 1)
(cell 2 1)
(cell 3 1)
(cell 4 1)
(cell 5 1)
(cell 6 1)
(cell 7 1)
(cell 8 1)
(++ 1 2)
(++ 2 3)
(++ 3 4)
(++ 4 5)
(++ 5 6)
(++ 6 7)
(++ 7 8)
(-- 8 7)
(-- 7 6)
(-- 6 5)
(-- 5 4)
(-- 4 3)
(-- 3 2)
(-- 2 1)
(succ 1 2)
(succ 2 3)
(succ 3 4)
(succ 4 5)
(succ 5 6)
(succ 6 7)
```

```
(succ 7 8)
(succ 8 9)
(succ 9 10)
(succ 10 11)
(succ 11 12)
(succ 12 13)
(succ 13 14)
(succ 14 15)
(succ 15 16)
(succ 16 17)
(succ 17 18)
(succ 18 19)
(succ 19 20)
(succ 20 21)
(succ 21 22)
(succ 22 23)
(succ 23 24)
(succ 24 25)
(succ 25 26)
(succ 26 27)
(succ 27 28)
(succ 28 29)
(succ 29 30)
(succ 30 31)
(succ 31 32)
(succ 32 33)
(succ 33 34)
(succ 34 35)
(succ 35 36)
(succ 36 37)
(succ 37 38)
(succ 38 39)
(succ 39 40)
(succ 40 41)
(succ 41 42)
(succ 42 43)
(succ 43 44)
(succ 44 45)
(succ 45 46)
(succ 46 47)
(succ 47 48)
(succ 48 49)
(succ 49 50)
```

11 Appendix B

Latent Tic-Tac-Toe defined in GDL-II. Based on a Tic-Tac-Toe game from (Dresden GGP server, 2010).

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Roles
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

  (role xplayer)
  (role oplayer)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Initial State
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

  (init (cell 1 1 b))
  (init (cell 1 2 b))
  (init (cell 1 3 b))
  (init (cell 2 1 b))
  (init (cell 2 2 b))
  (init (cell 2 3 b))
  (init (cell 3 1 b))
  (init (cell 3 2 b))
  (init (cell 3 3 b))
  (init (deniedx 1 1 b))
  (init (deniedx 1 2 b))
  (init (deniedx 1 3 b))
  (init (deniedx 2 1 b))
  (init (deniedx 2 2 b))
  (init (deniedx 2 3 b))
  (init (deniedx 3 1 b))
  (init (deniedx 3 2 b))
  (init (deniedx 3 3 b))
  (init (deniedo 1 1 b))
  (init (deniedo 1 2 b))
  (init (deniedo 1 3 b))
  (init (deniedo 2 1 b))
  (init (deniedo 2 2 b))
  (init (deniedo 2 3 b))
  (init (deniedo 3 1 b))
  (init (deniedo 3 2 b))
  (init (deniedo 3 3 b))
;; Who has control at start
  (init (control xplayer))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Dynamic Components
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; For deniedx and deniedo the X character means TRUE,
;; b means false

  (<= (next (cell ?m ?n x))
      (does xplayer (mark ?m ?n))
      (true (cell ?m ?n b)))

  (<= (next (deniedx ?m ?n x))
      (does xplayer (mark ?m ?n))
      (true (deniedx ?m ?n b)))

  (<= (next (cell ?m ?n o))
      (does oplayer (mark ?m ?n))
      (true (cell ?m ?n b)))

  (<= (next (deniedo ?m ?n x))
      (does oplayer (mark ?m ?n))
      (true (deniedo ?m ?n b)))

  (<= (next (cell ?m ?n ?w))
      (true (cell ?m ?n ?w))
      (distinct ?w b))

  (<= (next (cell ?m ?n b))
      (does ?w (mark ?j ?k))
      (true (cell ?m ?n b))
      (or (distinct ?m ?j) (distinct ?n ?k)))

  (<= (next (deniedx ?m ?n x))
      (true (deniedx ?m ?n x)))

  (<= (next (deniedo ?m ?n x))
      (true (deniedo ?m ?n x)))

  (<= (next (deniedx ?m ?n b))
      (does xplayer (mark ?j ?k))
      (true (deniedx ?m ?n b))
      (or (distinct ?m ?j) (distinct ?n ?k)))

  (<= (next (deniedo ?m ?n b))
      (does oplayer (mark ?j ?k))
      (true (deniedo ?m ?n b))
      (or (distinct ?m ?j) (distinct ?n ?k)))

  (<= (next (deniedo ?m ?n b))
      (does xplayer (mark ?m ?n))
      (true (deniedo ?m ?n b)))
```

```
(<= (next (deniedx ?m ?n b))
    (does oplayer (mark ?m ?n))
    (true (deniedx ?m ?n b)))

(<= (next (deniedo ?m ?n b))
    (does xplayer (mark ?j ?k))
    (true (deniedo ?m ?n b))
    (or (distinct ?m ?j) (distinct ?n ?k)))

(<= (next (deniedx ?m ?n b))
    (does oplayer (mark ?j ?k))
    (true (deniedx ?m ?n b))
    (or (distinct ?m ?j) (distinct ?n ?k)))

(<= (row ?m ?x)
    (true (cell ?m 1 ?x))
    (true (cell ?m 2 ?x))
    (true (cell ?m 3 ?x)))

(<= (column ?n ?x)
    (true (cell 1 ?n ?x))
    (true (cell 2 ?n ?x))
    (true (cell 3 ?n ?x)))

(<= (diagonal ?x)
    (true (cell 1 1 ?x))
    (true (cell 2 2 ?x))
    (true (cell 3 3 ?x)))

(<= (diagonal ?x)
    (true (cell 1 3 ?x))
    (true (cell 2 2 ?x))
    (true (cell 3 1 ?x)))


(<= (line ?x) (row ?m ?x))
(<= (line ?x) (column ?m ?x))
(<= (line ?x) (diagonal ?x))

(<= open
    (true (cell ?m ?n b)))

(<= (next (control oplayer))
    (does xplayer (mark ?x ?y))
    (legal xplayer (mark ?x ?y))
    (true (cell ?x ?y b))
    (true (control xplayer)))

(<= (next (control xplayer))
    (does xplayer (mark ?x ?y))
    (legal xplayer (mark ?x ?y))
```

```
      (true (cell ?x ?y o))
      (true (control xplayer)))

  (<= (next (control xplayer))
      (does oplayer (mark ?x ?y))
      (legal oplayer (mark ?x ?y))
      (true (cell ?x ?y b))
      (true (control oplayer)))

  (<= (next (control oplayer))
      (does oplayer (mark ?x ?y))
      (legal oplayer (mark ?x ?y))
      (true (cell ?x ?y x))
      (true (control oplayer)))

;; DEFINING SEES TERMS

;; If oplayer made a move which was legal but NOT valid
;; then he should see that his move was not accepted
  (<= (sees oplayer controlo)
      (does oplayer (mark ?x ?y))
      (legal oplayer (mark ?x ?y))
      (true (cell ?x ?y x))
      (true (control oplayer)))

;; If xplayer made a move which was legal but NOT valid
;; then he should see that his move was not accepted
  (<= (sees xplayer controlx)
      (does xplayer (mark ?x ?y))
      (legal xplayer (mark ?x ?y))
      (true (cell ?x ?y o))
      (true (control xplayer)))

;; When oplayer makes a valid and legal move xplayer gains
control
  (<= (sees xplayer controlx)
      (does oplayer (mark ?x ?y))
      (legal oplayer (mark ?x ?y))
      (true (cell ?x ?y b))
      (true (control oplayer)))

;; When xplayer makes a valid and legal move oplayer gains
control
  (<= (sees oplayer controlo)
      (does xplayer (mark ?x ?y))
      (legal xplayer (mark ?x ?y))
      (true (cell ?x ?y b))
      (true (control xplayer)))

;;If players move is legal, he should see it
  (<= (sees ?w (mark ?x ?y))
```

```
      (does ?w (mark ?x ?y))
      (legal ?w (mark ?x ?y)))

;; DEFINING LEGAL MOVES

  (<= (legal xplayer (mark ?x ?y))
      (true (deniedx ?x ?y b))
      (true (control xplayer)))

  (<= (legal oplayer (mark ?x ?y))
      (true (deniedo ?x ?y b))
      (true (control oplayer)))

  (<= (legal xplayer noop)
      (true (control oplayer)))

  (<= (legal oplayer noop)
      (true (control xplayer)))

;; DEFINING GOALS

  (<= (goal xplayer 100)
      (line x))

  (<= (goal xplayer 50)
      (not (line x))
      (not (line o))
      (not open))

  (<= (goal xplayer 0)
      (line o))

  (<= (goal oplayer 100)
      (line o))

  (<= (goal oplayer 50)
      (not (line x))
      (not (line o))
      (not open))

  (<= (goal oplayer 0)
      (line x))

;; DEFINING TERMINAL STATES

  (<= terminal
      (line x))

  (<= terminal
      (line o))
```

```
(<= terminal
    (not open))
```