VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

# AGENT BASED GAMEPLAYING SYSTEM

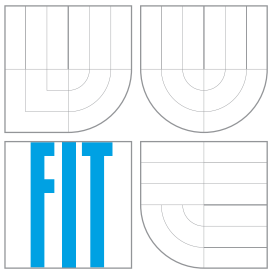
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

MICHAL TRUTMAN

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

# AGENTNÍ SYSTÉM PRO HRANÍ HER

AGENT BASED GAMEPLAYING SYSTEM

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

MICHAL TRUTMAN

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JIŘÍ KRÁL

BRNO 2015

# Abstrakt

Tato práce se zabývá universálními agentními systémy pro hraní her. Oproti běžným agentům, kteří jsou určeni pouze pro určitý druh činnosti nebo konkrétní hru, universální agent musí být schopen hrát prakticky libovolnou hru popsanou ve formálním deklarativním jazyce. Výzvou je především to, že pravidla hry nejsou předem známa, což znemožňuje použití některých optimalizací nebo vytvoření dobré heuristické funkce. Práce je rozdělena na teoretickou a praktickou část. První část představuje oblast univerzálních herních agentů, definuje jazyk GDL pro popis pravidel her a zabývá se vytvářením heuristických funkcí a jejich aplikací v algoritmu Monte Carlo stromové hledání. V praktické části je představen obecný způsob, jak vytvořit novou heuristickou funkci, která je poté integrována do vlastního herního agenta a ten je pak porovnán s dalšími existujícími systémy.



# Abstract

This thesis deals with general game playing agent systems. On the contrary with common agents, which are designed only for a specified task or a game, general game playing agents have to be able to play basically any arbitrary game described in a formal declarative language. The biggest challenge is that the game rules are not known beforehand, which makes it impossible to use some optimizations or to make a good heuristic function. The thesis consists of a theoretical and a practical part. The first part introduces the field of general game playing agents, defines the Game Description Language and covers construction of heuristic evaluation functions and their integration within the Monte Carlo tree search algorithm. In the practical part, a general method of creating a new heuristic function is presented, which is later integrated into a proper agent, which is compared then with other systems.



# Klíčová slova

agentní systém, hraní obecných her, Monte Carlo stromové hledání, heuristika, jazyk pro popis her

# Keywords

agent system, general game playing, Monte Carlo tree search, heuristics, game description language


# Citace

Michal Trutman: Agent Based Gameplaying System, diplomová práce, Brno, FIT VUT v Brně, 2015

# Agent Based Gameplaying System

## Declaration

I hereby declare that I have written this master's thesis by myself under the supervision of Ing. Jiří Král. All the sources that I have used for this project are listed in the bibliography section.

.....
Michal Trutman
May 24, 2015

## Acknowledgment

I would like to thank to my supervisor at Reykjavík University, Dr. rer. nat. Stephan Schiffel, for his valuable comments and advices. I am grateful for his time spent by giving me insights into hidden corners of General Game Playing.

# Contents

# Introduction

Most of the current agent systems are designed to perform only one specific task during their life time. For example, game playing agents are developed with only one game in mind. Teaching them, how to do something else, even very similar to their original task, would essentially involve major remake of that agent. Moreover, most of the intelligence of such agents heavily relies on human analysis of that specific problem, therefore the most difficult part is not done by the agents themselves, but instead by their programmers in advance.

However, another approach exists and that is, how to make an agent as generic as possible and this is what this thesis deals with. It is called *General Game Playing* (GGP) and it is concerned with the development of intelligent agents, that can play well any arbitrary game, given only its rules specified in a formal language.

There are many challenges to be overcome with the generic approach. The problem analysis must be done entirely by the agent itself and it has to use its own knowledge to decide, which algorithms to choose. Therefore, it is very difficult for the agent to come up with a good heuristic function or to do some effective optimizations. Because of this, generic agents are always the second, when it comes to compare them with their problem specific alternatives.

As any environment with possibly other agents in it, which changes according to rules, can be described as a game, general game playing agents should be able to solve large variety of problems. Although GGP systems are not suited for all environments (such as critical systems), they can be well used for example in autonomous robots or in prototyping to develop a good strategy, which can be later hard-coded into a problem specific agent.

The main target of this thesis is to develop a good heuristic for a general game playing agent, that can be used well within *Monte Carlo tree search* (MCTS) algorithm, which is currently the far most popular search space algorithm.

This thesis consists of five chapters. The first one gives an introduction into the GGP world and its main part is focused on the specification of the *Game Description Language* (GDL), a declarative language, in which game rules for GGP agents are represented. It is also shown, how games in GDL are simulated and how agents communicate between themselves during a match. The following chapter explains MCTS algorithm in detail and goes over common heuristics that can be used to guide the search.

In the third chapter, I propose a general algorithm to generate a new heuristic function. It uses regression to construct the heuristic from the game rules and a fuzzy logic to evaluate it. I also demonstrate, how additional knowledge about the game can be used to further improve its performance. Then, I implement an MCTS agent in Java and Prolog in the subsequent chapter and which is enriched with the heuristic function proposed.

In the final part, the agent is matched with other players in different testing schemes and in various games. Finally, the results are analysed and the agent's strengths and weaknesses are determined and explained.

# Chapter 1

# General Game Playing

*General Game Playing* (GGP) concerns with the development of general game playing agents. A *general game player* is a system, which accepts descriptions of any arbitrary games and uses them in such a way, that it is able to play those games effectively without human intervention. The game rules are not known to the player until the game starts [1].

General Game Playing is a relatively new field of study in artificial intelligence, although the idea, how to build a system, which can play large set of different games, can be traced back to year 1968, when the first general game player was built [2], albeit at that time it was limited to only various chess-like games played on a rectangular board with pieces. Many years later, in 1993, this idea was revisited again with a concept of Metagamer [3]. However, it took another decade until 2005, when GGP was truly brought into light with an introduction of the AAAI GGP competition [4]. From that year, the competition is held annually and several research groups were established around it. Since that, GGP did big steps forward and the state-of-the-art has been greatly advanced.

In these days, the AAAI competition is the most established event in GGP. Its intention is to test abilities of systems in various games and sharing knowledge. It is open to any player, which qualifies in a qualifying round. Then, players are matched against each other in series of games in the second round. The system to win the most games in this round becomes the winner of the competition.

This chapter gives first a short introduction into games and agents. Then, it explains syntax and semantics of the *Game Description Language* (GDL) – a first order logic language used for a description of game rules. The later part goes over game management, communication protocol and it introduces different types of logic reasoners.

## 1.1 Agents and Games

An *intelligent agent* is an autonomous entity perceiving its environment through sensors and acting upon that environment through actuators [5]. By this definition, agents are for example humans, animals, computer programs or robots. Each agent has an *agent function*, that maps from any possible percepts history to an action. The agent function determines agent's intelligence.

A *rational agent* is an agent, which always selects an action, which maximizes its performance given the agent's current knowledge. A rational agent is able to identify its goal and takes actions in order to fulfil it.

An *environment* is anything in which the agent operates. It could be a physical room with obstacles for a vacuum cleaning robot or a state space for a software agent.

A *game* is a strategic interaction between agents, defined by rules, that results in a quantifiable outcome [6]. In essence, a game is any situation described by rules, where agents are trying to reach certain (possibly conflicting) goals with optimal results.

A *strategy* for a player maps every possible game state to an action for that player. A player's strategy completely determines the action the player will take at any stage of the game.

Although previous definitions allow different types of agents, the focus of this thesis lies on software agents only.

### 1.1.1 Game Types

Games can be characterized by certain features and split into following groups:

- A *turn-taking game* is a game where only one player at time is allowed to perform an action. Otherwise it is a *simultaneous move game*, that is, players can do their actions concurrently.
- In *zero sum games*, the sum of payoffs for all players always adds up to zero (or a constant) for every set of strategies. Essentially, one player wins, only if the other one loses.
- A *deterministic game* is game, where the next state is completely determined by the current state and actions the agents do, i.e. there is no randomness. Otherwise, the game is called *non-deterministic* or *stochastic*.
- A *finite game* is a game with finite number of players, with finitely many actions the players can do, with finite number of possible game states and the game must terminate after finite number of steps. For example, a simple game where players roll a die, where the player with the highest sum wins, is not finite, if it is allowed to roll again after rolling a 6.
- In *games with perfect information*, all players have complete view of the current game state. In the contrary, where some information is hidden from any player, it is an *imperfect information game*.

## 1.2 Game Description Language

In general game playing, game rules are defined in a formal language called *Game Description Language* (GDL) [7]. GDL is a first order logic language with purely declarative semantic, so that no prior knowledge (such as arithmetic) is assumed. Its syntax and sematic is very similar to Datalog or Prolog.

This section explains first the underlying game model and characterises games, which GDL is capable to describe. Then the syntax and semantics of GDL are described and shown on example. Finally, some restrictions are established, so the game rules can be effectively processed.

### 1.2.1 Game Model

Games in GDL are modelled as finite state machines. That means, we are restricted to discrete games with fixed number of players and with finitely many states. The transitions between the states depend on *joint moves*, which is a list of moves, one for each player. To make a transition, a joint move has to be formed by all players selecting one of their legal moves. Effects of moves occur instantaneously and simultaneously – moves have no duration and all players play their action synchronously at the same time. Thus games in GDL are always described as games with simultaneous moves. Turn-taking games can be modelled by only introducing a move with no effect for players that do not have a turn, which is usually called a *noop move*.

Formally, a game is defined as follows [8]:

**Definition 1.1** (Game)**.** A game is a state transition system $(R, s_0, T, l, u, g)$ over sets of states $\mathcal{S}$ and actions $\mathcal{A}$, where:

- $R$, a finite set of roles;
- $s_0 \in \mathcal{S}$, the initial state of the game;
- $T \subseteq \mathcal{S}$, the set of terminal states;
- $l\colon R \times \mathcal{A} \times \mathcal{S}$, the legality relation;
- $u\colon (R \to \mathcal{A}) \times \mathcal{S} \to \mathcal{S}$, the transition or update function;
- $g\colon R \times \mathcal{S} \to \mathbb{N}$, the reward or goal function.

Each game starts in state $s_0$ and progresses until some terminal state is reached. Each player then receives a reward defined by the reward function $g$.

A game state in GDL consists of a set of *facts*, also called *fluents*, which are true in that state. Players in GDL are called *roles*, moves are also sometimes called *actions*.

There are more restrictions coming from the GDL game model. First, there is no randomness – all actions are deterministic and the environment is static. Second, GDL does not allow describing games with imperfect information. Thus, GDL permits to describe a large range of finite deterministic perfect-information simultaneous-move games with an arbitrary number of players.

An example of games that cannot be typically modelled in GDL includes large variety of card games as they contain both an element of chance (a shuffled deck of cards) and imperfect information (cards in opponents' hands).

### 1.2.2 Syntax

GDL is a first order logic based language, a variant of Prolog, that is, a game in GDL is a *logic program*. Its structure is described with the following statements:

- A *logic program* in GDL is a set of clauses.
- A *clause* it is one of:
  - A *rule* of form "`Head :- Body.`". Reads as, head is true, if body is true. The `':-'` symbol is a reverse implication symbol.

  - A *fact* is a rule without body, denoted as "`Head.`". Facts are always true.

- A *head* is an atom.
- A *body* is conjunction of literals delimited by a comma.
- A *literal* is an atom or its negation.
- An *atom* is a relational symbol applied to terms as its arguments. It is written as "`relation(Term1, Term2, ...)`".
- A *term* is either:
  - A *variable*. Variables are denoted by starting capital letter.
  - A *function* symbol applied to terms as arguments such as "`function(Term1, Term2, ...)`". Function names must start with a lower case letter or a digit.
  - A *constant*. A constant is a function symbol with no arguments.

There is also alternate prefix syntax to the Prolog syntax just introduced. It is known as *Knowledge Interchange Format* (KIF) [9] and it is the standard syntax used in communication between agents in GGP tournaments, so all current systems support it. The relationship between the two notations is direct, and it is easy to translate from one to the other. However, as the Prolog syntax is more human readable, it is used in the rest of this thesis.

Examples of both syntaxes are shown in Table 1.1.

| **Prolog syntax** | **KIF syntax** | **Description** |
|---|---|---|
| hello | hello | constant |
| 123 | 123 | constant |
| *Var* | ?*var* | variable |
| foo$(X, Y)$ | (foo ?$x$ ?$y$) | function |
| **not**(p(a, b, c)) | (**not** (p a b c)) | negation |
| q(a, b, c), r($X$) | (q a b c) (r ?$x$) | conjunction of literals |
| p(a, b, c). | (p a b c) | fact |
| p($Y$) :– q(a, $Y$), r(b, $Y$). | (<= (p ?$y$) (q a ?$y$) (r b ?$y$)) | rule |

Table 1.1: Examples of Prolog and KIF syntaxes of GDL.

A *ground rule* is a rule that does not contain any variables. A transformation of non-ground rule to ground rule is called *grounding* and it involves creating a set of new rules, where all variables are substituted with all combinations of values from their domains. A rule with $N$ different variables, each having domain of size $D$, will generate $D^N$ new rules, which is an exponential growth.

### 1.2.3 Semantics

The semantics of GDL is also similar to Prolog, however it is purely declarative. GDL contains 10 special keywords, used to describe various features of the game state machine. Notably, GDL must define roles, an initial state, terminal states, legal moves, payoffs and a state update function. These are also the only relations in GDL with fixed meaning.

- **role**($a$) means that $a$ is a role in the game.

- **init**$(p)$ means that the fluent $p$ is true in the initial state.
- **true**$(p)$ means that the fluent $p$ is true in the current state.
- **does**$(r, a)$ means that role $r$ performs action $a$ in the current state.
- **next**$(p)$ means that the fluent $p$ is true in the next state.
- **legal**$(r, a)$ means that $a$ is a legal move to play for role $r$ in the current state.
- **goal**$(r, n)$ means that role $r$ in the current state has payoff $n$, $n$ is an integer between 0 and 100.
- **terminal** means that the current state is a terminal state.
- **distinct**$(x, y)$ means that terms $x$ and $y$ are syntactically different.
- **not**$(x)$ means that term $x$ is not true.

GDL uses *negation-as-failure* semantics, that means, everything that cannot be proved to be true is considered false. In the contrary with Prolog, semantics of GDL does not depend on the order of clauses, nor on the order of literals in their bodies.

As mentioned before, a game in GDL is a logic program and it must give definitions for following relations:

- Complete definition of roles and an initial state using **role** and **init** as facts.
- Definition of legal moves, terminal states and payoffs using **legal**, **terminal** and **goal** in terms of **true** relation.
- Definition of a next state by **next** in terms of **does** and **true**.
- There must not be any clauses with **does** or **true** in their heads.

### 1.2.4 Example Game: Tic Tac Toe

As it is hard to understand GDL just from its definition, this section shows on an example, how to encode complete rules for a simple game in GDL. This game is called Tic Tac Toe and its rules are explained in Appendix A, where also rules for any other games used in examples through the thesis can be found.

We start with a definition of roles. There are two players: $x$ and $o$.

$$\textbf{role}(\mathsf{xplayer}). \tag{1.1}$$
$$\textbf{role}(\mathsf{oplayer}). \tag{1.2}$$

Then we specify an initial state, which is an empty 3×3 board. Also, we need to declare, that player $x$ starts the game (1.12).

$$\textbf{init}(\mathsf{cell}(1,1,\mathsf{blank})). \tag{1.3}$$
$$\textbf{init}(\mathsf{cell}(1,2,\mathsf{blank})). \tag{1.4}$$
$$\textbf{init}(\mathsf{cell}(1,3,\mathsf{blank})). \tag{1.5}$$
$$\textbf{init}(\mathsf{cell}(2,1,\mathsf{blank})). \tag{1.6}$$
$$\textbf{init}(\mathsf{cell}(2,2,\mathsf{blank})). \tag{1.7}$$

$$\textbf{init}(\mathsf{cell}(2,3,\mathsf{blank})). \tag{1.8}$$
$$\textbf{init}(\mathsf{cell}(3,1,\mathsf{blank})). \tag{1.9}$$
$$\textbf{init}(\mathsf{cell}(3,2,\mathsf{blank})). \tag{1.10}$$
$$\textbf{init}(\mathsf{cell}(3,3,\mathsf{blank})). \tag{1.11}$$
$$\textbf{init}(\mathsf{control}(\mathsf{xplayer})). \tag{1.12}$$

Now, we shall define legal moves. When a player is on turn, it is legal for it to mark a blank cell (1.13), otherwise it can play only a noop action (1.14).

$$\textbf{legal}(P, \mathsf{mark}(X,Y)) \text{ :– } \textbf{true}(\mathsf{control}(P)), \textbf{true}(\mathsf{cell}(X,Y,\mathsf{blank})). \tag{1.13}$$
$$\textbf{legal}(P, \mathsf{noop}) \text{ :– } \textbf{role}(P), \textbf{not}(\textbf{true}(\mathsf{control}(P))). \tag{1.14}$$

Next, we create state update rules. When the $\mathsf{mark}(X,Y)$ action is taken, then the corresponding cell is marked with the player's symbol (1.15, 1.16). Cells other than the marked cell keep their original value (1.17, 1.18). Lastly, control is passed to the other player (1.19, 1.20).

$$\textbf{next}(\mathsf{cell}(M,N,\mathsf{x})) \text{ :– } \textbf{does}(\mathsf{xplayer}, \mathsf{mark}(M,N)). \tag{1.15}$$
$$\textbf{next}(\mathsf{cell}(M,N,\mathsf{o})) \text{ :– } \textbf{does}(\mathsf{oplayer}, \mathsf{mark}(M,N)). \tag{1.16}$$
$$\textbf{next}(\mathsf{cell}(M,N,C)) \text{ :– } \textbf{true}(\mathsf{cell}(M,N,C)), \textbf{does}(P, \mathsf{mark}(X,Y)), \textbf{distinct}(X,M). \tag{1.17}$$
$$\textbf{next}(\mathsf{cell}(M,N,C)) \text{ :– } \textbf{true}(\mathsf{cell}(M,N,C)), \textbf{does}(P, \mathsf{mark}(X,Y)), \textbf{distinct}(Y,N). \tag{1.18}$$
$$\textbf{next}(\mathsf{control}(\mathsf{oplayer})) \text{ :– } \textbf{true}(\mathsf{control}(\mathsf{xplayer})). \tag{1.19}$$
$$\textbf{next}(\mathsf{control}(\mathsf{xplayer})) \text{ :– } \textbf{true}(\mathsf{control}(\mathsf{oplayer})). \tag{1.20}$$

The game terminates, when either player builds a line of its markers or if the board is full (not open). The board is open, if a blank cell exists.

$$\textbf{terminal} \text{ :– } \mathsf{line}(\mathsf{x}). \tag{1.21}$$
$$\textbf{terminal} \text{ :– } \mathsf{line}(\mathsf{o}). \tag{1.22}$$
$$\textbf{terminal} \text{ :– } \textbf{not}(\mathsf{open}). \tag{1.23}$$
$$\mathsf{open} \text{ :– } \textbf{true}(\mathsf{cell}(X,Y,\mathsf{blank})). \tag{1.24}$$

Finally, goal conditions. A player receives 100 points, if it builds a line of appropriate markers, or receives 0 points, if its opponent does it. If neither of them has a line, the game ends in a draw with 50 points.

$$\textbf{goal}(\mathsf{xplayer}, 100) \text{ :– } \mathsf{line}(\mathsf{x}). \tag{1.25}$$
$$\textbf{goal}(\mathsf{xplayer}, 50) \text{ :– } \textbf{not}(\mathsf{line}(\mathsf{x})), \textbf{not}(\mathsf{line}(\mathsf{o})). \tag{1.26}$$
$$\textbf{goal}(\mathsf{xplayer}, 0) \text{ :– } \mathsf{line}(\mathsf{o}). \tag{1.27}$$
$$\textbf{goal}(\mathsf{oplayer}, 100) \text{ :– } \mathsf{line}(\mathsf{o}). \tag{1.28}$$
$$\textbf{goal}(\mathsf{oplayer}, 50) \text{ :– } \textbf{not}(\mathsf{line}(\mathsf{x})), \textbf{not}(\mathsf{line}(\mathsf{o})). \tag{1.29}$$
$$\textbf{goal}(\mathsf{oplayer}, 0) \text{ :– } \mathsf{line}(\mathsf{x}). \tag{1.30}$$

And the one last thing remaining to be defined is the definition of a line. A line can be either a horizontal line (1.31), a vertical line (1.32) or one of the diagonal lines (1.33, 1.34).

$$\mathsf{line}(P) \text{ :– } \textbf{true}(\mathsf{cell}(1,Y,P)), \textbf{true}(\mathsf{cell}(2,Y,P)), \textbf{true}(\mathsf{cell}(3,Y,P)). \tag{1.31}$$

$$\mathsf{line}(P) \text{ :– } \mathbf{true}(\mathsf{cell}(X,1,P)), \mathbf{true}(\mathsf{cell}(X,2,P)), \mathbf{true}(\mathsf{cell}(X,3,P)). \quad (1.32)$$

$$\mathsf{line}(P) \text{ :– } \mathbf{true}(\mathsf{cell}(1,1,P)), \mathbf{true}(\mathsf{cell}(2,2,P)), \mathbf{true}(\mathsf{cell}(3,3,P)). \quad (1.33)$$

$$\mathsf{line}(P) \text{ :– } \mathbf{true}(\mathsf{cell}(1,3,P)), \mathbf{true}(\mathsf{cell}(2,2,P)), \mathbf{true}(\mathsf{cell}(3,1,P)). \quad (1.34)$$

### 1.2.5 Restrictions

Not all possible descriptions in GDL are actually corresponding with valid games, because agents must be able to use the rules to simulate games effectively. To ensure, that a game match runs smoothly, we must impose some restrictions. The first type of restriction guarantees that all computations done with the game description terminate, while the second type ensures, that the description is meaningful, i.e. each player has a legal move to play.

#### Finite Derivability

Finite derivability restriction ensures, that answering a query in GDL is decidable and always terminates, which is essential for efficient automated reasoning with rules.

**Definition 1.2** (Safety)**.** A clause is *safe* if and only if every variable in the clause appears in some positive literal in the body. A GDL description is safe if and only if all clauses are safe.

Note, that $\mathbf{distinct}(t_1, t_2)$ is a negative literal. Examples of unsafe rules follow:

$$\mathsf{r}(X,X). \quad (1.35)$$

$$\mathsf{p}(\mathsf{a}) \text{ :– } \mathbf{not}(\mathsf{q}(X)). \quad (1.36)$$

The problem with the rule (1.35) is, that it can produce infinitely many consequences, therefore facts must be always ground. On the other hand, the rule (1.36) has unclear semantics. Suppose that $\mathsf{q}(\mathsf{a})$ is true and $\mathsf{q}(\mathsf{b})$ false, then what is negation of $\mathsf{q}(X)$? In safe rules, variables within a negation can be always grounded first and such things cannot happen.

**Definition 1.3** (Dependency graph)**.** The *dependency graph* for a set of clauses $\mathcal{C}$ is a directed labelled graph where its nodes are predicate symbols from $\mathcal{C}$ and its edges are:

- A positive edge $p \xrightarrow{+} q$ if $\mathcal{C}$ contains a clause $p(\dots) \text{ :– } \dots, q(\dots), \dots$.
- A negative edge $p \xrightarrow{-} q$ if $\mathcal{C}$ contains a clause $p(\dots) \text{ :– } \dots, \mathbf{not}(q(\dots)), \dots$.

**Definition 1.4** (Stratified negation)**.** A *stratified negation* for a set of clauses is met, if and only if there are no cycles involving a negative edge in its dependency graph.

Stratified negation ensures, that a set of clauses has always single unique model. A following non-stratified rule illustrates the problem:

$$\mathsf{p}(X) \text{ :– } \mathsf{q}(X), \mathbf{not}(\mathsf{p}(X)). \quad (1.37)$$

**Definition 1.5** (Recursion restriction)**.** A *recursion restriction* for a set of clauses $\mathcal{C}$ holds, when if the rules contain a clause:

$$p(s_1,\ \ldots,\ s_m) \text{ :- } b_1(\bar{t}_1),\ \ldots,\ q(v_1,\ \ldots,\ v_k),\ \ldots,\ b_n(\bar{t}_n).$$

such that $p$ and $q$ occurs in a cycle in the dependency graph $G$ for $\mathcal{C}$, then for every $i \in \{1,\ \ldots,\ k\}$:

- $v_i$ is variable-free, or
- $v_i$ is one of $s_1,\ \ldots,\ s_m$, or
- $v_i$ occurs in some $\bar{t}_j, j \in \{1,\ \ldots,\ n\}$, such that $b_j$ does not occur in a cycle with $p$ in the dependency graph $G$.

In other words, recursion restriction says, that terms cannot grow indefinitely and every recursion terminates.

Finally, we can define finite derivability in terms of previously introduced supporting concepts.

**Definition 1.6** (Finite derivability)**.** A set of clauses is *finitely derivable*, if and only if it is safe and it fulfils both stratified negation and recursion restriction. Any valid GDL must be finitely derivable.

### Well-formed Games

This section brings us to a definition of *well-formed* games which ensures that any GDL describes a meaningful game with no invalid states and with well-defined goals.

**Definition 1.7** (Termination)**.** A game description in GDL *terminates* if all infinite sequences of legal moves from the initial state of the game reach a terminal state after finite number of steps.

In praxis, termination is usually achieved by adding a *step counter* into the game, so it is terminated prematurely in a draw after reaching certain number of steps.

**Definition 1.8** (Playability)**.** A game description in 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 1.9** (Winnability)**.** A game description in GDL is *strongly winnable* if and only if, for some role, there is a sequence of individual actions of that role that leads to a terminal state of the game where that role's goal value is maximal no matter what the others do. A game description in GDL is *weakly winnable* if and only if, for every role, there is a sequence of joint actions of all roles that leads to a terminal state where that role's goal value is maximal.

In other words, a game is strongly winnable for some role, if and only if that role can enforce victory, regardless of what the other roles do. A game is weakly winnable, if and only if for all roles, that role can win a game, if all roles cooperate on it.

**Definition 1.10** (Outcome Definedness)**.** A game description in GDL is *outcome defined* if and only if there is exactly one goal value for every role in every terminal state reachable from the initial state.

A definition of well-formedness differs between [7, 1, 8], but for the purpose of this thesis, the following definition will suffice. We assume, that all games are well-formed.

**Definition 1.11** (Well-formedness)**.** A game description in GDL is *well-formed* if it terminates, is outcome defined and is both playable and weakly winnable.

### 1.2.6 Fluent Properties

A fluent is a simple fact, that either holds or not in a given game state. The simplest property that can be identified is the information about fluent persistence as defined in [10]:

**Definition 1.12** (Fluent persistence)**.** A fluent is *persistent true* if, once it is true in a state, it will persist to be true in all future states. A fluent is *persistent false* if, once it becomes false in a state, it will never become true in any future state.

For instance, $\mathsf{cell}(1,1,\mathsf{x})$ is persistent true in Tic Tac Toe, as once marked cell stays marked until the game end. In contrast, $\mathsf{cell}(1,1,\mathsf{b})$ is persistent false in this game.

Second identifiable property is a *fluent mode*, which distinguishes between input and output arguments of the fluent [11]:

**Definition 1.13** (Fluent mode)**.** Consider an $n$-ary fluent $f$, denoted as $f/n$. By a *fluent mode* of $f$ we mean a function $m_f$ from $\{1,\ldots,n\}$ to the set $\{'+', '-'\}$. If $m_f(i) = '+'$, then $i$ is called an *input argument* of $f$. If $m_f(i) = '-'$, then $i$ is called an *output argument* of $f$.

Mode $m_f$ is written in a more readable form $f(m_f(1),\ldots,m_f(n))$.

For example, the mode of $\mathsf{cell}(1,1,\mathsf{b})$, that is the mode of $\mathsf{cell}/3$, is $\mathsf{cell}(+,+,-)$ meaning the first two arguments are inputs and the last one is an output argument. In general, one fluent can have more than one mode.

### 1.2.7 GDL-II

One disadvantage of GDL is, that it allows describing only perfect information deterministic games, which are far from real world situations, as real world always contains some randomness or something unknown. This issue was resolved in a new version of GDL, known as GDL-II [12]. It introduces two new keywords to define a random element and to control, which information is seen by each player.

However, the choice of publicly available games in GDL-II is still low in these days and so is the support in the current players. For this reason, this thesis considers only games written in pure GDL.

## 1.3 Game Management

In order to enrol general game players in a match, we need a central point – a *game manager* – to arrange and oversee the match. The game manager is responsible for distribution of game rules to players, maintenance of an official game state, verification of legality of moves and determination of winners [4].

To ensure the match progresses fast and smoothly, the game manager enforces two time constraints. *Play clock* is time in seconds the players can use to deliberate about each move, while *start clock* is extra time at the beginning of the match before the first turn starts.

### 1.3.1 Communication

A communication between the game manager and players is done through TCP/IP protocol, where the game manager acts as a client who connects to individual agents. To start a new match, game players must be already running and listening on the specified port. Then, we can select a game and assign its roles to the players. Finally, we specify start clock and play clock and start the game.

As the game is started, the game master connects to the players and exchanges messages with them using a simple request-response protocol. The messages are encoded in KIF syntax and the first parameter of each message sent to the players is a match ID – an arbitrary string ensuring, that all players are engaged in the same match.

The game begins with a `START` message containing a role the player is playing for, rules of the game in GDL, start clock and play clock. After waiting for start clock seconds and then every play clock seconds, game master sends a `PLAY` message which contains a joint move taken by all players in the last turn. The players should adjust their current game state accordingly, then deliberate about the next turn and reply with a move, they want to play. This repeats again and again until a terminal state is reached. When it happens, the game master sends a `STOP` message containing the last joint move, so the players can determine, who won the game. Table 1.2 shows sample communication between the game manager and a player.

In case a player fails to submit reply to the `PLAY` message in the time specified by play clock, the game master will take random legal move instead. The same happens, when the player responds in the time, but the move provided is not legal. In these situations, the player should be able to resume later and continue playing in subsequent turns.

| **Game Manager Message** | **Game Player Response** |
|---|---|
| `(START MATCH.1366 WHITE` *description* `60 10)` | `READY` |
| `(PLAY MATCH.1366 (NIL NIL))` | `(MARK 2 2)` |
| `(PLAY MATCH.1366 ((MARK 2 2) NOOP))` | `NOOP` |
| `(PLAY MATCH.1366 (NOOP (MARK 1 1))` | `(MARK 1 2)` |
| `(PLAY MATCH.1366 ((MARK 1 2) NOOP))` | `NOOP` |
| ⋮ | ⋮ |
| `(STOP MATCH.1366 ((MARK 3 3) NOOP)` | `DONE` |

Table 1.2: Sample game communication.

## 1.4 Reasoners

Having a formal game description is only one part of work in GGP. As game rules are written in logic, some form of automated reasoning is required to be able to play a game.

A hearth of each general game player is a *reasoner*. The reasoner is doing all computations with game rules in GDL, that is calculating legal moves, state update, checking state for terminal and computing goal values. As all current players spend most of their time in the reasoner, its performance significantly affects overall performance of a player. Having a fast reasoner is therefore crucial and even the fastest of the current general game playing reasoners cannot face a comparison with their game specific counterparts, as they are slower in orders of magnitude [13].

Two main approaches for interpreting the game descriptions exist: a direct interpretation of the GDL by means of logical programming or translation of the game rules to some alternative representation such as propositional networks.

### 1.4.1 Theorem Provers

Most of the current players rely on answering queries in GDL in a scope of logic programming, that is usually by proving using SLD resolution. Some of these players use custom made GDL interpreters, though they are rather slow as creating a custom reasoner is a highly demanding task. Others rely on translation of GDL rules to Prolog and interpret them using off-the-shelf Prolog engine.

The translation to Prolog is more or less straightforward, as GDL is a variant of Prolog. However, some precautions must be taken to ensure, that semantics stays the same. This applies mainly to a negation operator, as the standard negation-as-failure operator may produce non-logical results if its operand contains free variable. A solution is to reorder literals in a rule in such way, that the variables are bound first or to use sound negation operator instead (which delays until the operand in bound).

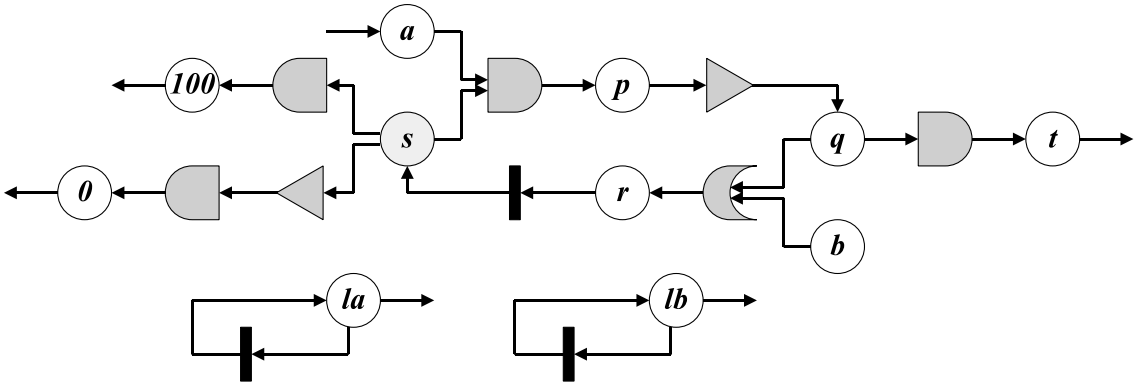
While custom based interpreters provide poor performance, translation to Prolog is a robust solution offering a reasonable speed.

### 1.4.2 Propositional Networks

A *propositional network* (propnet) is a structure in its essence very similar to an electric circuit with logic gates implementing a boolean function. Formally, a propositional network is a directed bipartite hypergraph consisting of *propositions* alternating with *connectives* (inverters, and-gates, or-gates, and transitions) [1]. There 3 types of propositions:

- *Base propositions* represent state fluents and their values correspond with values of matching **true**$(f)$ relation in the current state. Inputs of these components are constructed from **next**$(f)$ relation and the initial values are taken from **init**$(f)$.
- *Input propositions* correspond with **does**$(r, a)$ relation. When a successor state is to be computed, propositions appropriate to the joint move played are set to true.
- *Output propositions* are of three types, which correspond with **legal**$(r, a)$, **goal**$(n, r)$ and **terminal** relations. Their boolean value give information, whether given move is legal, which goal is reached or if the current state is terminal respectively.

Figure 1.1 shows a sample propnet for a simple game together with its description in GDL.




**role**(white). (1.38)

**legal**(white, a). (1.39)

**legal**(white, b). (1.40)

p :– **does**(white, a), **true**(s). (1.41)

q :– **not**(p). (1.42)

r :– q. (1.43)

r :– **does**(white, b). (1.44)

**next**(s) :– r. (1.45)

**goal**(white, 100) :– **true**(s). (1.46)

**goal**(white, 0) :– **not**(**true**(s)). (1.47)

**terminal** :– q. (1.48)

Figure 1.1: Sample propnet with its description in GDL for an incomplete game.

Every time a value of an input proposition changes (plus at the start of a game), it is propagated through the network. After this is done, base propositions reflect the current game state and output propositions can be used to determine legal moves and so.

Reasoning with propnets is very fast, the speed-up is in orders of magnitude comparing to prover reasoners. On the other hand, there are also significant disadvantages. Propositional networks cannot handle GDL with variables, which is frankly a feature of nearly all games. All is not lost, but the game rules must be grounded first, which takes considerable time and size of the game rules can grow up exponentially. That means, propnets are less robust as they can be used only in certain games.

An interesting fact about propnet reasoners is, that they can be entirely implemented in hardware. This goes very well together with field programmable gate arrays (FPGAs), because their logic can be reconfigured at any time. This suits the needs of GGP as a new propositional network could be loaded at the beginning of each match. As all reasoning would be done in hardware, it should be extremely fast. Nevertheless, none attempted to do it so far.

# Chapter 2

# Heuristic Search

Being able to systematically reason with game rules in GDL is only part of the puzzle, as every game player must also perform some kind of state space search. In the early days of GGP, different variants of MiniMax search were used, however this algorithm had several downsides. Its memory consumption was big and when exhaustive search of a state space was not feasible, because it was huge, some kind of heuristic was required to estimate values of unexplored subtrees. Although MiniMax was enhanced by many improvements, it could not compete with new *Monte Carlo tree search* (MCTS) algorithm. This simulation-based search space algorithm has started a new trend in the development of general game players. Although MCTS does not need any heuristic, as it implicitly focuses on the most promising parts of a game tree, a good heuristic function further improves its performance.

Automated generation of a useful and fast heuristic still remains one of the most difficult problems in GGP. This is in the contrary with game-specific players, where it is usually constructed by a programmer using his own knowledge about the game. In GGP, this process must be fully automated and the heuristic must be entirely created by the game player itself. There are two ways to create heuristics. First, *offline heuristic* relies on game analysis and feature detection before the game starts and once it is generated, it is used throughout the game. On the other hand, *online heuristic* is learned and constantly improved during the game play.

This chapter first describes the Monte Carlo tree search algorithm. Then, it shows, how to enhance it with a heuristic and introduces common control schemes.

## 2.1 Monte Carlo Tree Search

Monte Carlo tree search with UCT (*Upper Confidence Bounds Applied to Trees*) [14] represents the current state-of-the-art between search algorithms in GGP and it is widely used among game players. A clear evidence of its success is, that it has been used in winning players of the AAAI competition since 2007.

MCTS works by running complete simulations of a game, that is, repeatedly playing a simulation of a game starting at the current state and stopping when a terminal state is reached. The simulations are used to gradually build a game tree in memory. The nodes in this tree store an average reward (goal value) achieved by executing a certain action in the node. When deliberation time is up, the player plays the best move in the root node of the tree. Each simulation consists of four steps as depicted on Figure 2.1:

1. *Selection:* selecting actions in the tree based on their average reward until a leaf node of the tree is reached.
2. *Expansion:* adding one or several nodes to the tree.
3. *Playout:* playing randomly from the leaf node of the tree until a terminal state is reached.
4. *Back-Propagation:* updating values of the nodes in the tree with the reward achieved in the playout.

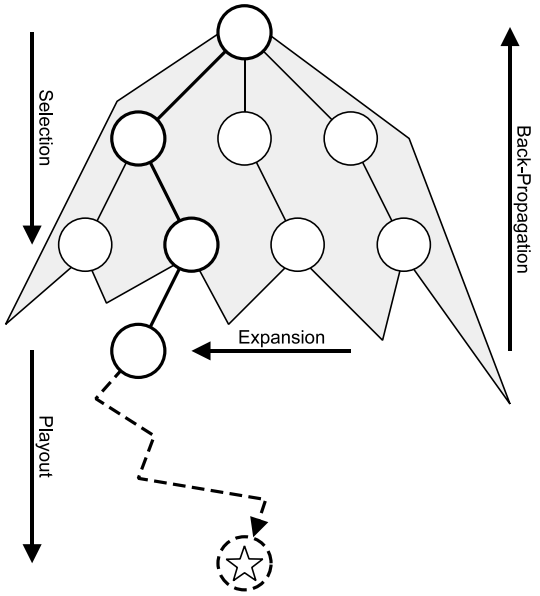



Figure 2.1: Schema of Monte Carlo tree search.

In the selection phase, there are two strategies, how to select an action:

- *Exploitation:* select the most promising action with the highest average score in a hope for the best result.
- *Exploration:* select the most unknown action with the lowest number of simulations in a hope for exploring something new.

Of course, none of these extremes is good, a solution is somewhere between. This is where UCT comes in, as it allows setting a trade-off between the two. In the selection phase, an action $a$ with the highest UCT value is chosen instead of relying on the average score $Q(s, a)$ entirely:

$$Q_{UCT}(s, a) = Q(s, a) + C * \sqrt{\frac{\ln(N(s))}{N(s, a)}}$$

where $N(s)$ is the number of simulations from the state $s$ and $N(s, a)$ is the number of simulations from $s$ with the action $a$. The constant $C$ controls exploitation vs. exploration. Typically, its value is empirically set between 20 and 40, where higher number means more exploration.

During the back-propagation step, $N(s)$ and $N(s,a)$ must be properly increased for every state $s$ and action $a$ on the path, as well as the average $Q(s,a)$ must be updated with the goal value achieved in the playout phase.

When applying MCTS to simultaneous moves games, the number of simulations and the average score with each action needs to be stored separately for each player. In the selection phase, each player chooses its action independently and then a joint move is formed. When propagating rewards up, the values for each player are updated with the reward reached by that player.

Monte Carlo tree search has several advantages over conventional MiniMax approach, which is the reason, why it so popular between current players. Firstly, it is suitable even for large games, as it does not require to expand full game tree in memory, instead a game tree is built gradually as the game progresses; it also achieves better results in games with higher branching factor, as the tree grows asymmetrically. Second, it does not require any heuristic, as an average of goal values or UCT is used instead to guide the search. On the other hand, MCTS does not terminate, it is just slowly converging to the true MiniMax value [15], however this helps to use deliberation time effectively, as the simulations can be interrupted at any time.

## 2.2 Heuristics

Although MCTS does not need any heuristic, a good heuristic function can significantly improve its performance. Heuristics used in search come traditionally in the form of a state evaluation function, that is, an evaluation of non-terminal states in a game. Especially in the context of MCTS, it seems advantageous to evaluate actions instead of states. In perfect-information turn-taking games, there is no difference between the value of an action and the value of the state that is reached by executing that action. However, games in GGP can have simultaneous moves in which case the successor state depends on the actions of all players. Even in the case of turn-taking games, evaluating an action directly instead of the successor state reached by that action saves the time needed to compute one state update. In GGP, this time is often significant compared to the time for computing the heuristics [13]. Both reasons make it beneficial to evaluate actions instead of states. The two types of evaluation are called *action heuristic* and *state heuristic* respectively.

In general, there are three ways, how a heuristic can be utilized in the MCTS player to control different parts of the search:

- *Tree heuristic:* a heuristic is used to control the game tree growth in the selection phase.
- *Playout heuristic:* a random action selection in the playout phase is biased towards more promising moves.
- *Combined heuristic:* a combination of the two previous approaches, a heuristic is used to guide the action selection both during the random playouts and in the game tree.

### 2.2.1 Related Work

Several ways were suggested of how to automatically generate heuristic offline. While in [16] they try to build a heuristic upon detecting common game features like a game board, game pieces or quantities; in [17] game properties like termination, control over the board

and payoff are used as components in the evaluation function. In [18], fuzzy logic is used to evaluate the goal condition in an arbitrary state and the value is used as a measure of how close the state is to a goal state. The approach is further improved by using feature discovery and was used in Fluxplayer, when winning the GGP competition in 2006.

A different approach relies on learning a heuristic online from simulations of the game. The first notable enhancement of MCTS was *Rapid Action Value Estimation* (RAVE) [19], a method to speed up the learning process of action values inside the game tree. A similar technique to learn state and move knowledge was based on which state fluents mostly occur in the winning states and which moves lie on the winning paths [20]. The state-of-the-art has also been greatly advanced by *Move-Average Sampling Technique* (MAST) [21]. MAST is a control scheme used in the playout phase of MCTS which learns the general value of an action independent from the context the action is used in. This and other control schemes such as *Features-to-Action Sampling Technique* (FAST) [22] were used by CadiaPlayer, a successful player that won the GGP competition three times. Recently, the MAST concept was made more accurate by using sequences of actions of given length (N-grams) instead of just single actions [23]. It has also been shown, that is possible to get more information from the playouts by assessing the lengths of simulations and evaluating the quality of the terminal state reached [24].

## 2.3 Search Controls

Some of the beforementioned methods are worth of closer look and they are explained in details in this section. We will use these control schemes later together to create a new heuristic. All methods are described in [22] and [18].

### 2.3.1 Move-Average Sampling Technique

In the standard MCTS with UCT, actions are selected uniformly at random during the playout phase. However, if we have any information on which actions are good, it is better to bias the action selection in favour of more promising moves. One way, how to determine good actions, is *Move-Average Sampling Technique* (MAST) [21], which relies on the fact, that actions that are good in one state are often good also in other states. To exploit this property, an agent must keep an average value $Q_{MAST}(a)$ for every action independent of a state. When a terminal is reached, $Q_{MAST}(a)$ is updated with the current outcome for every action $a$ on the path of a simulation. This works well in games, where certain actions are good not depending on the state, however it fails in highly strategic games like Chess.

Actions are then selected using the Gibbs (Boltzmann) distribution:

$$P(a) = \frac{e^{H(a)/\tau}}{\sum_{a'} e^{H(a')/\tau}}$$

where $P(a)$ is the probability that the action $a$ will be chosen in the current playout state and $H(a)$ is the action heuristic function, that is $Q_{MAST}(a)$ in this case. The parameter $\tau$ is *temperature* and controls the degree of randomness. Whereas high values makes it rather uniform distribution; $\tau \to 0$ means that more valued actions are chosen more likely.

### 2.3.2 Predicate-Average Sampling Technique

*Predicate-Average Sampling Technique* (PAST) [25] further improves MAST scheme by taking state context into account. It works as MAST, but instead of just keeping average

action values, it stores average values $Q_{PAST}(p, a)$ for each action and fluent pair. During back-propagation in a state $s$ where action $a$ was taken, $Q_{PAST}(p, a)$ is updated for all fluents $p$, which are true in the state $s$. An action is selected as in MAST in the playout phase, except $H(a)$ is substituted with $Q_{PAST}(p', a)$, where $p'$ is the true fluent in the state $s$ with the highest $Q_{PAST}$ value for action $a$.

Although PAST is more expensive than MAST, it can realize, that some actions are good only in certain situations.

### 2.3.3 Rapid Action Value Estimation

An example of tree heuristic is *Rapid Action Value Estimation* (RAVE ) [19], which is used to speed up the learning process inside the game tree. When backing up the outcome of a simulation, not only $Q(s, a)$ is updated for the action $a$ taken, but also special value $Q_{RAVE}(s, a')$ for any action $a'$, that occurs further down on the path below the state $s$. Essentially, $Q_{RAVE}(s, a')$ is an average outcome of simulations, where action $a'$ was taken in any state on the path below $s$.

Initially, only the heuristic is used to give an estimate of the action value, but as the sampled action value $Q(s, a)$ becomes more reliable with more simulations executed, it should be more trusted over the heuristic $H(s, a)$. This is achieved by using a weighted average:

$$Q(s, a)' = \beta(s) \times H(s, a) + (1 - \beta) \times Q(s, a)$$

with

$$\beta(s) = \sqrt{\frac{k}{3 \times N(s) + k}}$$

The *equivalence parameter* $k$ controls, how many simulations are needed for both estimates to have equal weights and the $N(s)$ function returns number of visits of the state $s$.

### 2.3.4 Goal Heuristic

Another idea, how to construct a heuristic function, is to estimate, how much the current state differs from a goal state. It was shown, that fuzzy logic can be used to evaluate the degree of truth of a goal condition [18]. The idea is to assign certain values to fluents depending on their truth value and then use fuzzy logic to evaluate complex formulas.

**Definition 2.1** (Fuzzy Evaluation)**.** For non-atomic formulas the evaluation function against the current game state $s$ is defined as:

$$\begin{aligned} eval(f \wedge g, s) &= \top(eval(f, s), eval(g, s)) \\ eval(f \vee g, s) &= \bot(eval(f, s), eval(g, s)) \\ eval(\neg f, s) &= 1 - eval(f, s) \end{aligned}$$

where $\top$ and $\bot$ are Yager family t-norms and t-co-norms:

$$\begin{aligned} \top(a, b) &= 1 - \bot(1 - a, 1 - b) \\ \bot(a, b) &= (a^q + b^q)^{\frac{1}{q}} \end{aligned}$$

All remaining atoms of the heuristic formula are of the form **true**$(X)$, these are evaluated as:

$$eval(\textbf{true}(f), s) = \begin{cases} p & \text{if } f \text{ is true in } s \\ 1 - p & \text{otherwise} \end{cases}$$

The value for $p$ parameter should be from interval $(0.5, 1]$, where $p = 1$ essentially transforms fuzzy logic into boolean logic. In general, the value of this parameter could vary between fluents, where higher values could be used for more important fluents (such as fluents, for which it is hard to switch their truth value in the game). The parameter $q$ determines appropriate form of t-norm and t-co-norm. The recommended values for both parameters are $q = 15$ and $p = 0.775$.

A higher value is returned by the evaluation function for states that are more similar to a goal state, if evaluated with goal condition state formula.

### 2.3.5 Combined Approaches

Previous control schemes can be combined together and a heuristic can be used to guide the action selection both during the random playouts and in the game tree. As it was shown in [25], this combination has a synergic effect, when RAVE was used as a tree heuristic and MAST as a playout heuristic.

# Chapter 3

# Generating Heuristic

Previous approaches to generate heuristics in GGP are very limited in the sense that the learned heuristics are very simple and often ignore the context in which an action is executed (MAST, RAVE), or it is a state evaluation function, and therefore not suitable for MCTS (such as goal heuristic).

A good heuristic function for MCTS algorithm should have following properties:

- An action evaluation function instead of a state evaluation function.
- Fast to evaluate. That is especially important, when used in playouts.
- Taking context into account.
- Not misleading. It should lead an agent towards fulfilling a goal.

Based on general knowledge about heuristics described earlier, I propose a new heuristic function. An idea for generation of such heuristic is to create an action-based version of the goal heuristic [18], which uses fuzzy logic to evaluate the degree of truth of a goal condition in a given state.

The key step is turning a state evaluation function into an action heuristic, which is achieved by taking the goal condition and by regressing it one step. This essentially yields a new condition which must be satisfied in the current game state, when this state is only one step away from the goal state.

## 3.1 Regression

The definition of regression in this thesis is based on regression in the *situation calculus* [26]. Similar to situation formulas in situation calculus, a *state formula* in a game is defined as any first-order formula over the predicate, function and constant symbols of the game description with the exception of the **does** predicate and any predicate depending on **does**.

Thus, the truth value of a state formula can be determined in any state independently on the actions that players choose in that state. For example, **goal**(xplayer, 100) is a state formula in the Tic Tac Toe game, because **goal** may not depend on **does** according to GDL restrictions. For the purpose of this thesis, it is only considered variable-free state formulas and game descriptions. Generalizing the proposed algorithm to non-ground game descriptions should be straight-forward.

**Definition 3.1** (Regression)**.** The regression of a variable-free state formula $F$ by one step, denoted as $\mathcal{R}[F]$, is defined recursively as follows:

- If $F$ is an atom $\textsf{true}(X)$, then:

$$\mathcal{R}[\textsf{true}(X)] = F_1 \vee F_2 \vee \ldots \vee F_n$$

  where $F_i$ are the bodies of all rules of the following form in the game description:

$$\textsf{next}(X) \text{ :- } F_i.$$

- If $F$ is an atom $\textsf{distinct}(a_1, a_2)$, then:

$$\mathcal{R}[\textsf{distinct}(a_1, a_2)] = \textsf{distinct}(a_1, a_2)$$

- If $F$ is any other atom $p(a_1, a_2, \ldots, a_n)$, then:

$$\mathcal{R}[F] = \mathcal{R}[F_1] \vee \mathcal{R}[F_2] \vee \ldots \mathcal{R}[F_n]$$

  where $F_i$ are the bodies of all rules of the following form in the game description:

$$p(a_1, a_2, \ldots, a_n) \text{ :- } F_i.$$

- If $F$ is a non-atomic formula then the regression is defined as follows:

$$\begin{aligned} \mathcal{R}[F_1 \vee F_2] &= \mathcal{R}[F_1] \vee \mathcal{R}[F_2] \\ \mathcal{R}[F_1 \wedge F_2] &= \mathcal{R}[F_1] \wedge \mathcal{R}[F_2] \\ \mathcal{R}[\neg F] &= \neg \mathcal{R}[F] \end{aligned}$$

Note, that GDL is finitely derivable by its definition, so computing the regression of a formula always terminates.

Once having the regression defined, it can be applied on a state formula. Let us have a look, what happens, when we try to regress $\textsf{true}(\textsf{cell}(1,1,\textsf{x}))$ from the well-known game Tic Tac Toe. The relevant part of the game rules follows:

$$\textsf{next}(\textsf{cell}(1,1,\textsf{x})) \text{ :- } \textsf{true}(\textsf{cell}(1,1,\textsf{b})), \textsf{does}(\textsf{xplayer}, \textsf{mark}(1,1)). \tag{3.1}$$

$$\textsf{next}(\textsf{cell}(1,1,\textsf{x})) \text{ :- } \textsf{true}(\textsf{cell}(1,1,\textsf{x})). \tag{3.2}$$

The regression of $\textsf{true}(\textsf{cell}(1,1,\textsf{x}))$ is computed by simply replacing it with the disjunction of the bodies of the two matching **next** rules:

$$\begin{aligned} \mathcal{R}[\textsf{true}(\textsf{cell}(1,1,\textsf{x}))] = \big(&\textsf{true}(\textsf{cell}(1,1,\textsf{b})) \wedge \textsf{does}(\textsf{xplayer}, \textsf{mark}(1,1))\big) \vee \\ &\textsf{true}(\textsf{cell}(1,1,\textsf{x})) \end{aligned} \tag{3.3}$$

The regressed formula (3.3) essentially says, when $\textsf{true}(\textsf{cell}(1,1,\textsf{x}))$ is true in the current state, then in the previous game state, there had to be $\textsf{true}(\textsf{cell}(1,1,\textsf{x}))$ already true, or the player $x$ had played the action $\textsf{mark}(1,1)$ and then $\textsf{true}(\textsf{cell}(1,1,\textsf{b}))$ had to be true.

We see, that the assertion about the previous state differs depending on, which action the player $x$ has taken. By substituting $\textsf{does}(\textsf{xplayer}, \textsf{mark}(1,1))$ with true, we get a state formula related to the previous state, when action $\textsf{mark}(1,1))$ was taken from that state by player $x$. This is essentially a heuristic for the action $\textsf{mark}(1,1))$.

## 3.2 Algorithm

Based on the previous definition, following algorithm to generate a heuristic function for each action $a$ of a player $p$ is proposed. The algorithm consists of following steps:

1. Compute $\mathcal{R}[\textbf{goal}(p, 100)]$, the regression of $\textbf{goal}(p, 100)$. $\mathcal{R}[\textbf{goal}(p, 100)]$ represents a condition on a state and actions of players that – when fulfilled – allow to reach a goal state for player $p$.

   For now, only the highest valued goal is taken into consideration for each player. Combining different goals could be done similar to the way described in [18], but would make the heuristics more expensive to compute.

2. $\mathcal{R}[\textbf{goal}(p, 100)]$ contains conditions on actions of players. However, we want a formula that indicates when it is a good idea for player $p$ to execute action $a$. To obtain such a formula, we restrict $\mathcal{R}[\textbf{goal}(p, 100)]$ to those parts that are consistent with $\textbf{does}(p, a)$. In practice this is achieved by replacing all occurrences of $\textbf{does}(r, b)$ for any $r$ and $b$ in $\mathcal{R}[\textbf{goal}(p, 100)]$ as follows:

   (a) In case $p = r$ and $a = b$, the occurrence of $\textbf{does}(r, b)$ is replaced with the boolean constant true.

   (b) In case $p = r$, but $a \neq b$, the occurrence of $\textbf{does}(r, b)$ is replaced with the boolean constant false.

   (c) In case $p \neq r$, the condition is on an action for another player. In that case, the replacement depends on whether or not the game is turn-taking:

      i. In case of a game with simultaneous moves, the occurrence of $\textbf{does}(r, b)$ is replaced with unknown value in three-valued logic. This represents, that we are not sure, which action the opponent decides to play.

      ii. In case of a turn-taking game, if $b$ is a noop action, the occurrence of $\textbf{does}(r, b)$ is replaced with true, otherwise with false (because $r$ must do a noop action if $p$ is doing a non-noop action such as $a$).

3. The formula is simplified according to laws of three-valued logic. In particular, any constants introduced in previous steps are propagated up and the formula is partially evaluated.

For performance reasons, it is better to perform all the steps in one go, that is recursively regress the formula, while replacing **does** and suitable **true** with their values and return already simplified formula from each recursion step.

### 3.2.1 Example: Tic Tac Toe

This section demonstrates on an example, how the algorithm works on a simplified version of the game Tic Tac Toe, where the goal was reduced to build only any of the two diagonal lines. The grounded and expanded version of the goal for the player *xplayer* is:

$$\begin{aligned}\textbf{goal}(\mathsf{xplayer}, 100) \text{ :- } &\big(\textbf{true}(\mathsf{cell}(1,1,\mathsf{x})) \wedge \textbf{true}(\mathsf{cell}(2,2,\mathsf{x})) \wedge \textbf{true}(\mathsf{cell}(3,3,\mathsf{x}))\big) \vee \\ &\big(\textbf{true}(\mathsf{cell}(1,3,\mathsf{x})) \wedge \textbf{true}(\mathsf{cell}(2,2,\mathsf{x})) \wedge \textbf{true}(\mathsf{cell}(3,1,\mathsf{x}))\big).\end{aligned} \tag{3.4}$$

Assume, we are computing the heuristic function for the action $\mathsf{mark}(1, 1)$ for the role *xplayer*. The first atom to regress in (3.4) is $\textbf{true}(\mathsf{cell}(1, 1, \mathsf{x}))$. Its regression has been

thoroughly shown in the section 3.1, so here is the result only:

$$\big(\textbf{true}(\mathsf{cell}(1,1,\mathsf{b})) \wedge \textbf{does}(\mathsf{xplayer},\mathsf{mark}(1,1))\big) \vee \textbf{true}(\mathsf{cell}(1,1,\mathsf{x})) \tag{3.5}$$

The **does** predicate in (3.5) is replaced with boolean true, because both the role and the action match the ones we are interested in right now (3.6) and the formula is simplified (3.7). Thus (3.7) is the final replacement for $\textbf{true}(\mathsf{cell}(1,1,\mathsf{x}))$.

$$\big(\textbf{true}(\mathsf{cell}(1,1,\mathsf{b})) \wedge \mathrm{T}\big) \vee \textbf{true}(\mathsf{cell}(1,1,\mathsf{x})) \tag{3.6}$$

$$\textbf{true}(\mathsf{cell}(1,1,\mathsf{b})) \vee \textbf{true}(\mathsf{cell}(1,1,\mathsf{x})) \tag{3.7}$$

Going back to the goal condition (3.4), the next part of the formula to be regressed is $\textbf{true}(\mathsf{cell}(2,2,\mathsf{x}))$. The matching **next** rules are:

$$\textbf{next}(\mathsf{cell}(2,2,\mathsf{x})) \text{ :- } \textbf{true}(\mathsf{cell}(2,2,\mathsf{b})), \textbf{does}(\mathsf{xplayer},\mathsf{mark}(2,2)). \tag{3.8}$$

$$\textbf{next}(\mathsf{cell}(2,2,\mathsf{x})) \text{ :- } \textbf{true}(\mathsf{cell}(2,2,\mathsf{x})). \tag{3.9}$$

The regression of $\textbf{true}(\mathsf{cell}(2,2,\mathsf{x}))$ therefore yields formula (3.10). This time, the **does** keyword is replaced with boolean false, because the role matches, but the action does not (3.11). This can be simplified to (3.12), which equals the term we have started with, meaning that $\textbf{true}(\mathsf{cell}(2,2,\mathsf{x}))$ stays in the formula untouched.

$$\big(\textbf{true}(\mathsf{cell}(2,2,\mathsf{b})) \wedge \textbf{does}(\mathsf{xplayer},\mathsf{mark}(2,2))\big) \vee \textbf{true}(\mathsf{cell}(2,2,\mathsf{x})) \tag{3.10}$$

$$\big(\textbf{true}(\mathsf{cell}(2,2,\mathsf{b})) \wedge \mathrm{F}\big) \vee \textbf{true}(\mathsf{cell}(2,2,\mathsf{x})) \tag{3.11}$$

$$\textbf{true}(\mathsf{cell}(2,2,\mathsf{x})) \tag{3.12}$$

We repeat the same steps for any other **true** keywords in the goal (3.4). As in the last case, each term is replaced by the term itself and nothing in the formula is changed. The final heuristic formula for *xplayer* taking action $\mathsf{mark}(1,1)$ follows (3.13):

$$\begin{aligned}&\Big(\big(\textbf{true}(\mathsf{cell}(1,1,\mathsf{b})) \vee \textbf{true}(\mathsf{cell}(1,1,\mathsf{x}))\big) \wedge \textbf{true}(\mathsf{cell}(2,2,\mathsf{x})) \wedge \textbf{true}(\mathsf{cell}(3,3,\mathsf{x}))\Big) \vee \\ &\quad\Big(\textbf{true}(\mathsf{cell}(3,1,\mathsf{x})) \wedge \textbf{true}(\mathsf{cell}(2,2,\mathsf{x})) \wedge \textbf{true}(\mathsf{cell}(1,3,\mathsf{x}))\Big)\end{aligned} \tag{3.13}$$

As it can be seen, this condition describes a situation in which *xplayer* taking action $\mathsf{mark}(1,1)$ would lead to a winning state. Thus, a boolean evaluation of such conditions for all legal moves in a state is equivalent to doing 1-ply lookahead.

## 3.3 Evaluation

Using the algorithm above, the heuristic formula can be constructed for any role and any potentially legal move during start clock. During game play, the formula for each legal action can be evaluated against the current game state using fuzzy logic as described in section 2.3.4 about goal heuristic. In essence, a higher value of the evaluation means, that more prerequisites are satisfied for a particular action to lead to a goal state.

However, during experimenting with the heuristic, it was discovered, that the fuzzy evaluation function from Definition 2.1 is too slow compared with the time needed to calculate legal moves and state updates. Further investigation revealed, that the main reasons for the slowdown were the power and root operations used in the Yager family t-norms and

t-co-norms. Instead, these are replaced with a product t-norm $\top$ and a probabilistic sum t-co-norm $\perp$ as follows:

$$\begin{aligned} \top(a,b) &= a \cdot b \\ \perp(a,b) &= a + b - a \cdot b \end{aligned}$$

This change resulted into an approximately tenfold speedup in the evaluation, although the heuristic is less accurate than it was with the original t-norm and t-co-norm as used in [18]. The value for the parameter $p$ defining the evaluation of a true fluent must be adjusted accordingly and $p = 0.97$ is used.

The heuristic values (after normalization) for each legal action in the aforementioned version of the game Tic Tac Toe are shown in Table 3.1.

| | | |
|---|---|---|
| (1, 3)<br>25 | (2, 3)<br>0 | (3, 3)<br>25 |
| (1, 2)<br>0 | (2, 2)<br>50 | (3, 2)<br>0 |
| (1, 1)<br>25 | (2, 1)<br>0 | (3, 1)<br>25 |

Table 3.1: Heuristic evaluation of actions in the simplified version of Tic Tac Toe.

The heuristic function was evaluated in the initial game state (an empty board). We can see, that the action with the highest value, is to take the middle cell ($\mathsf{mark}(2,2)$), followed by 4 actions taking one of the corner cells. Indeed, this corresponds with the fact that marking the middle cell as the first action leads to the most options for winning the game.

## 3.4 Control Schemes

This heuristic can be easily incorporated in the MCTS algorithm and it can be used both in the playouts and within the game tree. Therefore three control schemes can be established: a playout heuristic, a tree heuristic and a combined heuristic mixing the two approaches together.

We can use Gibbs distribution to guide the random playouts, as described in the MAST section, while weighted average as in RAVE can be used within the selection phase. Any heuristic values must be normalized and scaled appropriately before use.

Experiments have shown, that good values for the $\tau$ parameter are somewhere between 0.5 and 2; $\tau = 1$ was used. The equalization constant $k$ was set to 5. These values are different than in MAST and RAVE, because of the different distribution of values of the action heuristic.

However, the tree heuristic scheme can be slightly improved as the value of the equivalence parameter $k$ is static throughout the game match, which is not optimal. As the game progresses, the number of MCTS simulations is increasing from turn to turn, because they are getting shorter and so the equivalence parameter should reflect this change.

What we typically want, is giving the heuristic relatively more influence, when the number of simulations is low, but less influence as the number increases. This requirement

is satisfied even with static value of $k$, however it does not scale properly as its influence might be too much with low number of simulations and rather negligible with high number.

Following formula can be used to adjust the value of the equivalence parameter at the beginning of each turn to reflect this trend:

$$k = \frac{\sqrt{N}}{d}$$

where $N$ is the number of simulations done during the previous turn and $d$ is a *divisor constant*. When $d$ is set to 20, the resulting value of equivalence parameter is 5 for 10000 simulations done, which can be considered as a standard value.

## 3.5 Additional Knowledge

This section describes, how an additional knowledge about the game can be used to further improve properties of the heuristic in some situations. The additional knowledge can be obtained either by formal proves [27, 10] or by looking up for certain syntactic structures in GDL rules [16]. Additionally, a hypothesis can be formulated, that is verified with certain probability in random game states.

### 3.5.1 Persistent Fluents

Simplest type of additional knowledge is information about fluent persistence. In case this knowledge could be (automatically) inferred, the fuzzy evaluation function from Definition 2.1 is modified as follows:

$$eval(\mathbf{true}(f), s) = \begin{cases} 1 & \text{if } f \text{ is true in } s \text{ and } f \text{ is persistent true} \\ p & \text{if } f \text{ is true in } s \text{ and } f \text{ is not persistent true} \\ 0 & \text{if } f \text{ is false in } s \text{ and } f \text{ is persistent false} \\ 1-p & \text{otherwise} \end{cases}$$

### 3.5.2 Fluent Modes

Fluent mode provides useful information, because it restricts the number of fluents with the same name, arity and the same input arguments to at most one in any reachable game state. Consider $\mathsf{control/1}$ fluent with mode $\mathsf{control}(-)$. As there is no input argument, this essentially means, that there is no more than one $\mathsf{control/1}$ fluent in any given state. Similarly for $\mathsf{cell}(1,1,\mathsf{x})$, which has mode $\mathsf{cell}(+,+,-)$. The first two parameters are inputs, so there can be at most one fluent of form $\mathsf{cell}(1,1,p)$ in any reachable state for any arbitrary $p$. Thus, any state in Tic Tac Toe can contain only one of the following fluents: $\mathsf{cell}(1,1,\mathsf{x})$, $\mathsf{cell}(1,1,\mathsf{o})$, $\mathsf{cell}(1,1,\mathsf{b})$ or none.

How can we use the fluent modes to improve the heuristic? Consider the fluents $\mathsf{cell}(1,1,\mathsf{x})$ and $\mathsf{cell}(1,1,\mathsf{o})$, which are both true persistent. If one of those becomes true in a state $s$, the other one must not hold in that state. Moreover, the other fluent gets a new property, as it becomes false persistent from any state following the state $s$. This can be reflected in the heuristic function, that $\mathsf{cell}(1,1,\mathsf{x})$ will be replaced with $\mathsf{cell}(1,1,\mathsf{x}) \wedge \neg\mathsf{cell}(1,1,\mathsf{o})$.

In general, any fluent $f(x,y)$ in the heuristic formula will be replaced with $f(x,y) \wedge \neg\big(f(x,b_1) \vee \ldots \vee f(x,b_n)\big)$ when $f$ has a mode with $x$ as input and $y$ as output arguments and $f(x,b_1), \ldots, f(x,b_n)$ are all the true persistent fluents of the form $f(x,b)$ for any $b \neq y$.

### 3.5.3 Move Preconditions

When computing a regression of a goal condition for a player $p$ restricted to an action $a$, we can use additional knowledge, that we have about the action $a$. As the heuristic will be used to estimate values only of the actions, that are legal in a given state, we know, that $\textbf{legal}(p, a)$ must hold. With some luck, truth values of some fluents occurring in the **legal** relation could be calculated. In particular, if there is only one $\textbf{legal}(p, a)$ rule for the action $a$ and the player $p$ and the body of this rule is a conjunction (which is by default), then all the conjuncts must be also true to satisfy the final condition.

Once the truth value of some fluent is determined, then its mode can be used to infer truth values of other fluents with the same name, arity and input arguments. Finally, all the occurrences of this fluent can be replaced with the logic value assigned to it, which results in a simplified formula.

Unfortunately, this improvement works only in certain games, because the **legal** condition is usually too complex to be able to find out truth values of any useful fluents. In most cases, truth value of control fluent can be found easily, however this fluent usually does not occur in the regression of a goal.

In the example of Tic Tac Toe, let us assume we want to compute precondition for the move $\mathsf{mark}(1, 1)$ with the following **legal** relation for the player *xplayer*:

$$\textbf{legal}(\mathsf{xplayer}, \mathsf{mark}(1, 1)) \text{ :– } \textbf{true}(\mathsf{cell}(1, 1, \mathsf{b})). \tag{3.14}$$

As the **legal** relation is very simple, it clear that $\textbf{true}(\mathsf{cell}(1, 1, \mathsf{b}))$ must hold any state, where $\mathsf{mark}(1, 1)$ is the legal move to play. This allows simplifying the heuristic function (3.13) as it was created in the section 3.2.1 to the much shorter formula (3.15):

$$\begin{aligned}\big(\textbf{true}(\mathsf{cell}(2, 2, \mathsf{x})) \wedge \textbf{true}(\mathsf{cell}(3, 3, \mathsf{x}))\big) \vee \big(\textbf{true}(\mathsf{cell}(3, 1, \mathsf{x})) \wedge \textbf{true}(\mathsf{cell}(2, 2, \mathsf{x})) \wedge \\ \textbf{true}(\mathsf{cell}(1, 3, \mathsf{x}))\big)\end{aligned} \tag{3.15}$$

### 3.5.4 Turn Taking Games

Another type of knowledge is the information, whether the game is turn taking or with simultaneous moves, because special precautions can be taken in turn taking games. This knowledge is already incorporated in the step 2(c) of the algorithm generating the heuristic. While any **does** in the formula is replaced with either true or false in turn taking games depending on its arguments, in case of simultaneous games, also unknown value can be used, which introduces extra uncertainty into the heuristic formula.

In case we are not able to discover this information properly and a noop move cannot be determined, we must assume that the game has simultaneous moves.

# Chapter 4

# Implementation

In this chapter, I present an implementation of a new general game player. This includes implementing the action heuristic that has been proposed earlier in this thesis, choosing a good third party GGP framework and a reasoner, creating a MCTS algorithm with other supporting concepts and integrating all the parts together.

At first, architecture of the player is described and the frameworks, on which it is built, are characterized. Later, an implementation of both generation and evaluation of the action heuristic is explained.

## 4.1 Frameworks

Developing a general game playing system from scratch is a very demanding task; especially when it comes to develop a custom reasoner. Luckily, a few frameworks that ease the new agent creation exist, so all effort could be spent on developing a good agent function.

The agent, which is built in this thesis, uses components of the following game playing systems:

- *GGP-base* [28] - a complete framework for a development of GGP agents, that is designed in order to make the development of new agents as simple as possible. In the simplest case, only the agent decision function needs to be implemented, while other stuff such as network communication is completely handled by the framework itself. It comes with a slow theorem prover reasoner.
- *Fluxplayer* [18] - an agent written in Prolog, that uses Prolog engine as a reasoner. Fluxplayer is also known that it can prove certain properties of the game and then uses them to its advantage [27, 10].
- *Sancho player* [29] - a complex agent built on the GGP-base framework. Although full source code of the agent is available, the only part used in this thesis is the efficient implementation of a propnet state machine.

### 4.1.1 GGP-base

In the GGP-base framework, a new player can be created by extending the class `StateMachineGamer` and implementing some of the basic 8 methods. Notably, the following functions are worth of closer look:

- `getName` – can return some fancy name of the player;

- `getInitialStateMachine` – this function is used to choose the state machine implementation, which will be used to simulate the match. It should return a new object that is subclass of the `StateMachine` class. By default, simple logic-based prover is available, however Sancho player provides also an efficient propnet implementation;
- `stateMachineMetaGame` – called during the time assigned for start clock and can be used by the player to analyse the game, initialize all structures and deliberate about the game. The timeout parameter specifies a deadline, until this function must return;
- `stateMachineSelectMove` – this is the core function of each player, that is used to deliberate about each turn in a game. It is called once per turn allowing play clock to be used for deliberation. The function must return a move, that the player wishes to play before the timeout specified as the first parameter expires;
- `stateMachineStop` – called, when the game match finishes normally;
- `stateMachineAbort` – similar to `stateMachineAbort`, but it is called, when the match ends abruptly.

The GGP-base framework calls these functions automatically when necessary. Within the `StateMachineGamer` class, the game can be simulated using the state machine created, that provides handy functions to calculate state updates or legal moves and to verify terminal or goal. The gamer automatically handles all communication with the game master and it is responsible for keeping the current game state up to date.

## 4.2 Architecture

All general game players share the same basic architecture and consist of three basic modules that can be implemented separately and reused again when developing a new agent. These are:

- *Networking* – handles all communication with the game master;
- *State machine* – simulates the game;
- *Artificial intelligence* – the agent function, represents the agent's logic.

In case of the agent, that is being developed, it was decided to split the implementation into two logically divided parts. The core part is built on top of the GGP-base framework, where the built-in logic-based reasoner was replaced with much faster propnet reasoner from Sancho player. This part is entirely written in Java and it is responsible for MCTS search and evaluation of the heuristic function. The second part is in charge of generating the heuristic and it is built on Fluxplayer codebase, that is written in Prolog. This was decided, because GDL as a logic language is a variant of Prolog and Prolog is well suited for processing it, as it can handle any transformations of logic formulas efficiently. The Fluxplayer codebase is used to ground game rules (as this is necessary to generate the heuristic) plus it can also prove some interesting game properties, that can be later used.

Communication between the two parts is done via temporary files, as there is no need for interactive communication, because the Prolog code is run only once at the beginning of a match to generate the heuristic.

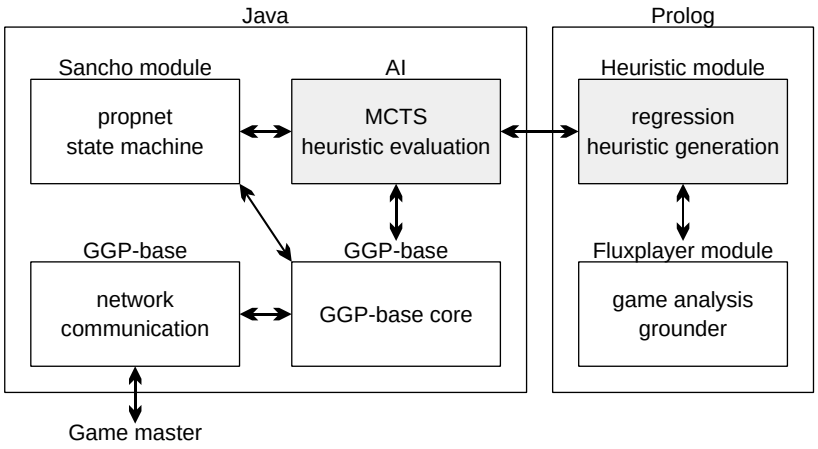



Figure 4.1: Architecture of the agent.

The architecture of the agent is depicted on Figure 4.1. Grey components in the image represent the parts that are being developed in this thesis.

The following search control schemes have been implemented to analyse different behaviour of the heuristic: playout, tree and a combined scheme. Moreover, I have implemented also MAST, RAVE and MAST+RAVE players to have a comparison with well-established concepts plus a non-heuristic player is also available.

## 4.3 Generating Heuristic

The heuristic is generated according to the algorithm, that has been described in the section 3.2 with the difference, that the three steps are all done recursively in one pass. In contrary with doing them one by one, this allows keeping the size of all intermediate results within a manageable range, as the big expansion in the size of all formulas, which is introduced in the first step, is being immediately reduced by performing the two remaining steps.

The algorithm is implemented in Prolog, in the ECLiPSe Constraint Programming System.

Game rules are loaded from a temporary file specified. Then, the rules are grounded and the game is analysed for interesting properties by the Fluxplayer engine. Afterwards, the heuristic is generated and exported into a file in GDL. This is done during start clock at the beginning of each match.

During the processing, all formulas are represented as standard logic expressions in Prolog. That is, conjunction is denoted by a comma, disjunction by a semicolon and negation either by a tilde `'~'` or by `'\+'` symbol. Keywords in GDL are represented as Prolog predicates, however `'d_'` prefix is prepended. For instance, $\textbf{legal}(\mathsf{xplayer}, \mathsf{mark}(1,1))$ becomes `d_legal(xplayer, mark(1, 1))` in Prolog.

All the three improvements with use of additional knowledge are implemented as they were described in the section 3.5. Moreover, all resulting formulas are optimized using absorption law and all duplicate operands of conjunctions and disjunctions are detected and removed (idempotence law). Especially these two optimizations are a little bit tricky to do, when logic formulas are represented as standard Prolog expressions, because all conjunctions and disjunctions with multiple inputs are constructed from multiple binary

operators (taking two inputs). This hides certain properties of the formula and makes the expression syntax tree deeper than necessary. For optimization purposes, consecutive operations of the same type are grouped into one operation with multiple inputs.

### 4.3.1 Implementation in Prolog

The following code sample 4.1 shows, how the heuristic can be obtained in Prolog. The variables *Role* and *Move* represent the role and the action for which the regression is calculated, such as *xplayer* and mark(1, 1). At first, a goal for given role is picked and then the heuristic is generated by `regress_goal/3` function.

Listing 4.1: Getting a goal

```
1 Goal = d_goal(Role, 100),
2 regress_goal(Goal, d_does(Role, Move), Heuristic),
```

The `regress_goal/3` predicate recursively processes the goal formula. Notably, the first rule of the simplified code snippet 4.2 performs a replacement of a fluent with the matching **next** rule, which is the key step of the regression. Moreover, the `ggp_ground_axiom/2` succeeds, when the first argument is a head of a rule in GDL and the second argument unifies with a body of that rule.

Listing 4.2: Computing regression

```
1 regress_goal(d_true(Fluent), Does, Return) :- !,          %fluent
2   replace_does(d_next(Fluent), Does, Return).
3 regress_goal(true, _, true) :- !.                         %true
4 regress_goal(fail, _, fail) :- !.                         %false
5 regress_goal((X, Y), Does, Return) :- !,                  %conjunction
6   regress_goal(X, Does, XX),
7   regress_goal(Y, Does, YY),
8   evaluate_and(XX, YY, Return).
9 regress_goal((X; Y), Does, Return) :- !,                  %disjunction
10  regress_goal(X, Does, XX),
11  regress_goal(Y, Does, YY),
12  evaluate_or(XX, YY, Return).
13 regress_goal((\+ X), Does, Return) :- !,                 %negation
14  regress_goal(X, Does, Y),
15  evaluate_not(Y, Return).
16 regress_goal(X, Does, Return) :- !,                      %predicate
17  findall(Axiom_body, ggp_ground_axiom(X, Axiom_body), Axioms),
18  list_to_or(Axioms, Ors),
19  regress_goal(Ors, Does, Return).
```

When the regression is obtained, all occurrences of **does** must be evaluated in relation to the current role and the action. This is implemented in the predicate `replace_does/3`, whose incomplete implementation is shown in the listing 4.3. The missing rules are in essence very similar to the lines 3 to 19 from the listing 4.2.

Listing 4.3: Replacing **does**

```
1 %matching role and action
2 replace_does(d_does(Role, Action), d_does(Role, Action), true) :- !.
3 %non matching action
```

```
4 replace_does(d_does(Role, _), d_does(Role, _), fail) :- !.
5 %non matching role, turn taking game
6 replace_does(d_does(Role, Action), d_does(_, _), Return) :-
7   is_only_one_players_choice, get_noop_move(Role, Noop), !,
8   (Noop = Action -> Return = true ; Return = fail).
9 %non matching role, simultaneous move game
10 replace_does(d_does(_, _), d_does(_, _), unknown) :- !.
11 %fluent
12 replace_does(d_true(X), _, d_true(X)) :- !.
13 ...
```

The last step of the algorithm is to simplify the formula using logic laws, which is performed in the `evaluate_and/3`, `evaluate_or/3` and `evaluate_not/2` functions.

All the code listings above have been simplified in a way that they show only the main idea, but they omit other advanced features such as caching, optimizations or use of additional knowledge.

### 4.3.2 Precomputing Replacements

Because grounded game descriptions are much bigger than their non-grounded versions, efficiency of the implementation is a serious concern. To make the algorithm reasonably fast, it is necessary to avoid repeated work, which is achieved by caching intermediate results in a hash map. However, this does not solve all the speed related problems. Consider following example from the grounded description of the game Skirmish:

$$\mathsf{gen\_exists\_54(a,2,a,2)} \text{ :- } \textbf{does}(\mathsf{white}, \mathsf{move(wk,a,1,a,2)}). \tag{4.1}$$

$$\mathsf{gen\_exists\_54(a,2,a,2)} \text{ :- } \textbf{does}(\mathsf{white}, \mathsf{move(wr,a,1,a,2)}). \tag{4.2}$$

$$\vdots \tag{4.3}$$

$$\mathsf{gen\_exists\_54(a,2,a,2)} \text{ :- } \textbf{does}(\mathsf{white}, \mathsf{move(br,h,2,a,2)}). \tag{4.4}$$

The rule $\mathsf{gen\_exists\_54(a,2,a,2)}$ was generated by the grounder and it has 76 realizations. Then, during the heuristic generation process, it is scanned for any **does**, which is replaced with boolean value depending on whether its arguments match the role and the action it is currently regressed. This is done for every legal move and there are about 1729 potentially legal moves for each role in Skirmish. As there are many more rules like this, it is clear, that these calculations become soon too costly. Caching does not help here, as the result might be potentially different for each legal move.

However, there is a room for improvement, as only one rule of those with the same head is relevant for any given move, while the replacement in all the other rules produces false. In the context, where multiple rules with same head are understood as a disjunction, rules producing false as a result can be ignored. However it is not known before, which rules can be skipped until they are evaluated, thus the idea is to find out which rules are relevant for which moves in advance. This is done by calculating two possible results of the replacement and keeping a set of relevant moves for each rule, where one result is valid for any move from the set and the second is the result for moves that are not in the relevant set.

This improvement does not work on all formulas as it covers only some simple cases, however similar rules as above are very common in grounded game descriptions. A replacement of **does** in the formula is then calculated just once instead of doing this separately for each move.

## 4.4 Evaluating Heuristic

Once the heuristic formula is generated, it can be evaluated using fuzzy logic and utilized in the MCTS player. It is crucial to do the evaluation as fast as possible, as any slowdown can easily counterweight gains of the heuristic function.

### 4.4.1 Propositional Networks

For the purpose of evaluating the heuristic, a propositional network representing the heuristic formula is built. This has the advantage, that any duplicated parts of the formula (and there are always many of them as formulas for similar moves have usually similar subparts) are represented by the same component and evaluated just once.

There are two different representations of a propnet implemented in the player. First implementation uses dynamic arrays and bidirectional (but oriented) bindings between components. This implementation is well suited for further modifications of the propnet in memory, but it is slower on evaluation. This propnet can perform some simple optimizations such as constant propagation. The second representation is more compact and performance oriented and uses fixed size structures and one-way bindings between components, so it cannot be modified once built.

The idea is to have the first type dynamic propnet and keep it optimized all the times, while using the static propnet that is fast to evaluate and recreate it from the first propnet every times it is modified. This way is the highest speed possible assured.

When the root state of a game tree is updated, it is checked whether any true or false persistent fluents have already reached their persistent status. If so, then they are replaced with true or false constants respectively and their values are further propagated through the propnet. This results into much smaller network and faster evaluation as the match progresses in some games.

### 4.4.2 Fuzzy Evaluation

For determining a value of a propnet component, lazy evaluation is used. That is nothing is computed, when a new game state is loaded into the propnet, until the value of such component is needed. An up-to-dateness of each component is represented by an integer counter – the component value is considered to be current, if the value of its counter is the same as the value of the global propnet counter. In such case, cached value is returned, otherwise a new value is calculated recursively from other components.

The heuristic is evaluated for each legal move in any state that occurs during the playout phase. When used within the game tree, it is evaluated and the results are stored in game nodes. The evaluation happens only after the node is fully expanded.

# Chapter 5

# Evaluation

The general game playing agent, that has been designed and implemented in previous chapters, is now compared with other solutions and GGP players. It is consecutively matched against a non-heuristic MCTS player, against well-known MAST and RAVE heuristic schemes and finally, it is compared with full-fledged GGP agents – Fluxplayer, CadiaPlayer and Sancho; renowned agents that won the GGP AAAI competition at least once. Finally, the results are evaluated with respect to the reasons that stand behind and the agent's strengths and weaknesses are identified. In the end, possibilities of future development are discussed.

## 5.1 Test Setting

Three sets of experiments were run to test behaviour of the agent under different settings. First, the agent was matched against a pure non-heuristic MCTS/UCT player using the three aforementioned concepts of the action heuristic (playout, tree and combined heuristic) with constant number of simulations per turn. Then, the playout, tree and combined heuristic were matched against MAST, RAVE and MAST+RAVE players respectively as examples of well-established control schemes. Constant time per turn was used in this experiment. Finally, the player using the combined heuristic was opposed by full-fledged GGP agents, namely by CadiaPlayer [22], Fluxplayer [18] and Sancho [29].

The values of search control parameters as used in the tests are shown in Table 5.1 and they present the best known setting of the player. The values for MAST and RAVE were set as recommended in [25].

| **Parameter** | | **Usage** | **Value** |
|---|---|---|---|
| $C$ | UCT constant | MCTS | 40 |
| $\tau$ | temperature | playout heuristic | 1 |
| $k$ | equivalence parameter | tree heuristic | calculated dynamically |
| $d$ | divisor constant for $k$ | tree heuristic | 20 |
| $\tau$ | temperature | MAST | 10 |
| $k$ | equivalence parameter | RAVE | 1000 |

Table 5.1: Parameter values used in the tests.

Each of the first two experiments consist of 300 matches per game with each scheme; 150 matches per game were used in the last test set The tests were run on Linux with multicore Intel Xeon 2.40 GHz processor with 4 GB memory limit and 1 CPU core assigned to each

| Game | MCTS vs. Playout | | MCTS vs. Tree | | MCTS vs. Combi. | |
|---|---|---|---|---|---|---|
| Battle | 85.0 × **91.5** | (±2.47) | 88.5 × **95.6** | (±1.61) | 90.9 × **97.5** | (±1.32) |
| Bidding Tic Tac Toe | 42.8 × **57.2** | (±3.21) | 40.8 × **59.2** | (±3.11) | 39.3 × **60.7** | (±3.49) |
| Blocker | 55.3 × 44.7 | (±5.63) | 52.7 × 47.3 | (±5.65) | 49.0 × 51.0 | (±5.66) |
| Breakthrough | 47.3 × 52.7 | (±5.65) | 51.7 × 48.3 | (±5.65) | 52.3 × 47.7 | (±5.65) |
| Checkers (small) | **77.3** × 22.7 | (±3.87) | 50.2 × 49.8 | (±4.37) | **79.0** × 21.0 | (±3.65) |
| Chinese Checkers | 67.2 × **82.8** | (±2.69) | 74.5 × 75.4 | (±2.84) | 69.3 × **80.7** | (±2.76) |
| Chinook | 50.7 × 59.7 | (±5.60) | 51.7 × 57.0 | (±5.62) | 51.0 × 60.7 | (±5.56) |
| Connect 4 | **65.2** × 34.8 | (±5.12) | 50.7 × 49.3 | (±5.45) | **62.0** × 38.0 | (±5.15) |
| Criss Cross | 61.3 × 63.8 | (±4.24) | 63.0 × 62.0 | (±4.24) | 62.0 × 63.0 | (±4.24) |
| Ghost Maze | 20.0 × **80.0** | (±2.77) | 19.3 × **80.7** | (±2.94) | 24.8 × **75.2** | (±3.15) |
| Nine Board Tic Tac Toe | 19.7 × **80.3** | (±4.50) | 33.7 × **66.3** | (±5.35) | 18.7 × **81.3** | (±4.41) |
| Pentago | 43.3 × **56.7** | (±5.29) | 28.8 × **71.2** | (±4.66) | 20.8 × **79.2** | (±4.27) |
| Sheep and Wolf | **58.7** × 41.3 | (±5.57) | **59.7** × 40.3 | (±5.55) | **65.0** × 35.0 | (±5.40) |
| Skirmish | **78.1** × 74.6 | (±1.42) | 79.3 × 77.0 | (±1.53) | **80.8** × 76.2 | (±1.66) |
| | 55.1 × **60.2** | (±1.30) | 53.2 × **62.8** | (±1.28) | 54.6 × **61.9** | (±1.30) |

Table 5.2: Tournament using the playout, tree and combined heuristic against pure MCTS player with fixed number of simulations.

agent. Rules for all the games tested come the Tiltyard GGP server [30] and they can be found on the CD attached. Rules for particularly interesting games are also explained in Appendix A. The opposing GGP agents were run using their default settings, however when multithreading was enabled, it was limited to one CPU core to allow the same computing power for all the players. All the agents were used in the latest version available.

MCTS/UCT, RAVE, MAST and all the three action heuristic players share the same base implementation of the MCTS algorithm, therefore the test results should give good evidence about contribution of different heuristic schemes.

## 5.2 Playing Strength

The tables 5.2 to 5.4 show the average scores reached by the agents along with a 95 % confidence interval under the three aforementioned test settings. Bold numbers indicate significant win ratio for the highlighted agent. Also note that not all the games tested are zero sum games.

### 5.2.1 Comparison with MCTS/UCT

Table 5.2 contains results for a tournament against non-heuristic MCTS player. In this experiment, all players were allowed to perform 10000 MCTS simulations per turn not limiting them by any time constraint.

The game with the strongest position of the combined scheme is Nine Board Tic Tac Toe with a score 81 (±4.4) against 19; it is also good in Pentago, Ghost Maze and Bidding Tic Tac Toe. On the other hand, it is particularly bad in Checkers and Connect4, but closer look reveals, that this is only because of the playout heuristic, while the tree heuristic has not much influence in these games. The playout heuristic follows the similar trend as the combined scheme. On the other hand, the tree heuristic scheme performs significantly better than the pure MCTS in 5 games, significantly worse only in Sheep and Wolf and it has rather no effect in the remaining games.

| Game | MAST vs. Playout | | RAVE vs. Tree | | M+R vs. Combi. | |
|---|---|---|---|---|---|---|
| Battle | 94.7 × **98.8** | (±0.80) | 85.7 × **91.2** | (±2.07) | 90.0 × 92.3 | (±1.53) |
| Bidding Tic Tac Toe | 29.2 × **70.8** | (±4.61) | 30.0 × **70.0** | (±4.43) | 22.0 × **78.0** | (±4.36) |
| Blocker | 52.0 × 48.0 | (±5.65) | 52.0 × 48.0 | (±5.65) | 54.7 × 45.3 | (±5.63) |
| Breakthrough | **67.7** × 32.3 | (±5.29) | 48.0 × 52.0 | (±5.65) | **72.3** × 27.7 | (±5.06) |
| Checkers (small) | **84.3** × 15.7 | (±3.28) | 48.5 × 51.5 | (±4.37) | **83.0** × 17.0 | (±3.42) |
| Chinese Checkers | **84.0** × 66.0 | (±2.64) | 75.8 × 74.0 | (±2.84) | **81.2** × 68.3 | (±2.75) |
| Chinook | **64.3** × 43.7 | (±5.52) | 55.0 × 59.0 | (±5.60) | **69.3** × 42.3 | (±5.40) |
| Connect 4 | **62.8** × 37.2 | (±5.28) | 54.8 × 45.2 | (±5.31) | **67.3** × 32.7 | (±5.06) |
| Criss Cross | 62.5 × 62.5 | (±4.24) | 62.5 × 62.5 | (±4.24) | 62.5 × 62.5 | (±4.24) |
| Ghost Maze | 25.2 × **74.8** | (±2.87) | 27.8 × **72.2** | (±3.30) | 30.8 × **69.2** | (±3.22) |
| Nine Board Tic Tac Toe | 35.0 × **65.0** | (±5.40) | 33.3 × **66.7** | (±5.33) | 25.0 × **75.0** | (±4.90) |
| Pentago | 37.3 × **62.7** | (±4.90) | 28.2 × **71.8** | (±4.55) | 24.0 × **76.0** | (±4.47) |
| Sheep and Wolf | **67.3** × 32.7 | (±5.31) | 53.0 × 47.0 | (±5.65) | **70.7** × 29.3 | (±5.15) |
| Skirmish | **84.8** × 80.8 | (±1.59) | **79.5** × 71.3 | (±1.49) | **87.4** × 74.7 | (±1.48) |
| | **60.8** × 56.5 | (±1.34) | 52.4 × **63.0** | (±1.30) | **60.0** × 56.5 | (±1.34) |

Table 5.3: Tournament against MAST, RAVE and MAST+RAVE players with constant time per turn.

It is also worth mentioning, how the results in the combined control scheme are connected to the playout and the tree heuristic. It seems, the influence of the playout heuristic on the overall result is much higher. Especially, when it is useless or even misleading, then the combined result is dragged down by it (Checkers, Connect4). It seems that the playout heuristic is more vulnerable, while the tree heuristic control scheme can recover when the heuristic is misleading. Thus, it is essential for the playout heuristic to be good, if it is used.

An interesting example is Bidding Tic Tac Toe – a game, where two players are bidding coins in order to mark a cell on a Tic Tac Toe board. The key property of this game is the bidding part, the game can be easily lost by doing wrong bids, even when the markers are placed in good positions on the board. The action heuristic does not help in any way with the bidding, it only helps to arrange markers in a line. In spite of this, all the heuristic agents still have significant advantage in this game. One explanation is that when a player wins a bid, it can actually use its move well which makes especially the playouts more reliable.

In general, it can be said, that the heuristic player performs very well in any Tic Tac Toe-like games, as they contain many persistently true fluents and thus the heuristic leads the player to the goal more directly.

### 5.2.2 Comparison with RAVE and MAST

Table 5.3 shows results for matches played against MAST, RAVE and MAST+RAVE. Although MAST and MAST+RAVE outperform the playout and the combined heuristic in most of the games, a good position is still held in Pentago or Nine Board Tic Tac Toe. The heuristic performs surprisingly well against RAVE with just one significant loss.

By comparing the tables 5.2 and 5.3, we see that the action heuristics perform well against MAST and RAVE in the same games (with exception of Chinese Checkers) in which they did well against a pure MCTS/UCT player. This suggests, that this approach is complementing MAST and RAVE by improving performance in games in which MAST and RAVE do not seem to have much positive effect.

| Game | Cadia vs. Combi. | | Flux vs. Combi. | | Sancho vs. Combi. | |
|---|---|---|---|---|---|---|
| Battle | 27.7 × **90.9** | (±3.02) | 68.4 × **85.1** | (±3.78) | 83.3 × **92.7** | (±2.62) |
| Bidding Tic Tac Toe | 31.3 × **68.7** | (±6.70) | 0.0 ×**100.0** | (±0.00) | **79.7** × 20.3 | (±4.14) |
| Blocker | 42.0 × **58.0** | (±7.90) | 36.7 × **63.3** | (±7.71) | 45.3 × 54.7 | (±7.97) |
| Breakthrough | 35.3 × **64.7** | (±7.65) | 0.7 × **99.3** | (±1.30) | **99.3** × 0.7 | (±1.30) |
| Checkers (small) | 44.3 × 55.7 | (±7.31) | 47.3 × 52.7 | (±7.08) | **75.3** × 24.7 | (±5.60) |
| Chinese Checkers | 42.7 × **99.3** | (±1.56) | 54.3 × **87.3** | (±3.05) | **81.3** × 55.7 | (±3.27) |
| Chinook | 33.6 × **68.5** | (±7.52) | 11.3 × **56.0** | (±6.51) | **95.3** × 6.0 | (±3.59) |
| Connect 4 | 42.7 × 57.3 | (±7.47) | 9.3 × **90.7** | (±4.37) | **93.0** × 7.0 | (±3.92) |
| Criss Cross | 57.5 × 67.5 | (±5.95) | 56.0 × **69.0** | (±5.91) | 68.0 × 57.0 | (±5.94) |
| Ghost Maze | **73.3** × 26.7 | (±3.99) | **59.0** × 41.0 | (±5.47) | **73.7** × 26.3 | (±4.10) |
| Nine Board Tic Tac Toe | 17.3 × **82.7** | (±6.06) | 0.7 × **99.3** | (±1.30) | 26.7 × **73.3** | (±7.08) |
| Pentago | 8.3 × **91.7** | (±4.17) | 3.7 × **96.3** | (±2.94) | **74.0** × 26.0 | (±6.18) |
| Sheep and Wolf | 27.3 × **72.7** | (±7.13) | 53.3 × 46.7 | (±7.98) | **93.3** × 6.7 | (±3.99) |
| Skirmish | 15.7 × **69.9** | (±2.14) | 75.7 × 72.5 | (±3.22) | **83.1** × 68.4 | (±2.15) |
| | 35.7 × **69.6** | (±1.76) | 34.0 × **75.7** | (±1.68) | **76.5** × 37.1 | (±1.64) |

Table 5.4: Tournament against CadiaPlayer, Fluxplayer and Sancho using the combined heuristic with constant time per turn.

### 5.2.3 Comparison with Other Players

The combined heuristic agent outperforms both CadiaPlayer and Fluxplayer with significant advantage. The only game, where these two agents were able to reach significant win ratio is Ghost Maze, which is ironically one of the games, where the combined heuristic agent performed well in other tests. However, the agent is dominated by Sancho with an average score 37 against 77. The combined heuristic agent seems to be completely hopeless in Breakthrough, Sheep and Wolf , Chinook and Connect 4 against Sancho, but as in other tests, it is able to win in Nine Board Tic Tac Toe with a significant gap.

This can be considered as a good result, as Sancho is currently the world's best GGP player winning the latest GGP AAAI competition in 2014; and CadiaPlayer is the only player, that won this competition three times, although it is no longer maintained. On the other hand, the results would be different, if multithreading was allowed, as both CadiaPlayer and Sancho support parallelization, however extending the agent with it should be straightforward.

During the tests, CadiaPlayer was crashing constantly in Battle, Chinese Checkers and Skirmish; while Fluxplayer was producing frequent timeouts.

## 5.3 Time Spent on Evaluating the Heuristic

Table 5.5 shows how much time during the game time was spent on evaluating the heuristic and the time needed to generate the heuristic functions for each game. These numbers do not include the time required for grounding the game description. However, most general game players use grounded game descriptions for reasoning nowadays, such that grounding needs to be done anyway.

The time required to generate the action heuristic is relatively small for the games tested, except for Checkers with almost 15 seconds and Battle with 13 seconds. However, the time spent on evaluating the heuristic is more important. While it is almost negligible for the tree heuristic, it ranges from 5 to 70 % depending on the game for the playout and combined heuristic. The table also shows ratio between the time needed to run 10000

| Game | Cost of gen. | Time spent on heuristic Playout | Tree | Combi. | Relative number of simuls. Playout | Tree | Combi. |
|---|---|---|---|---|---|---|---|
| Battle | 13.0 s | 45.8 % | 0.0 % | 45.6 % | 54.5 % | 99.5 % | 54.4 % |
| Bidding Tic Tac Toe | 0.2 s | 19.2 % | 0.3 % | 18.6 % | 97.3 % | 105.1 % | 96.8 % |
| Blocker | 0.2 s | 49.2 % | 0.1 % | 47.4 % | 51.0 % | 96.1 % | 53.0 % |
| Breakthrough | 3.5 s | 58.5 % | 0.1 % | 58.4 % | 40.9 % | 104.0 % | 41.0 % |
| Checkers (small) | 14.8 s | 27.9 % | 0.1 % | 27.7 % | 70.0 % | 103.1 % | 70.6 % |
| Chinese Checkers | 0.1 s | 8.9 % | 0.1 % | 8.8 % | 86.2 % | 101.5 % | 85.8 % |
| Chinook | 2.5 s | 9.5 % | 0.0 % | 9.5 % | 105.9 % | 102.8 % | 106.1 % |
| Connect 4 | 1.6 s | 61.9 % | 1.9 % | 60.7 % | 60.4 % | 115.2 % | 69.5 % |
| Criss Cross | 2.7 s | 5.9 % | 0.6 % | 6.3 % | 92.0 % | 102.7 % | 94.9 % |
| Ghost Maze | 0.2 s | 49.6 % | 0.8 % | 48.9 % | 55.9 % | 102.6 % | 57.9 % |
| Nine Board Tic Tac Toe | 4.0 s | 54.2 % | 0.7 % | 53.4 % | 88.9 % | 104.6 % | 93.7 % |
| Pentago | 1.2 s | 73.9 % | 0.4 % | 73.9 % | 30.8 % | 101.0 % | 30.9 % |
| Sheep and Wolf | 3.4 s | 16.3 % | 0.1 % | 16.3 % | 83.2 % | 102.2 % | 83.0 % |
| Skirmish | 6.8 s | 28.6 % | 0.1 % | 28.6 % | 68.7 % | 100.2 % | 69.0 % |

Table 5.5: Cost of generating the action heuristic, percentage of the game time spent on evaluating it and relative number of simulations compared to the non-heuristic player.

game simulations by the pure and the heuristic players. Surprisingly, these numbers do not always correspond with the percentage of the game time spent on the evaluation, because the heuristic makes the simulations effectively shorter and thus taking less time. A good example of this behaviour is in Nine Board Tic Tac Toe, where about 50 % of the game time is spent on evaluating the heuristic, but the number of simulations performed by such player is only about 10 % lower. Moreover, the combined heuristic player won 81 % of the matches against pure MCTS/UCT.

## 5.4 Testing with More Games

The agent was able to generate action heuristic for 113 games out of 127 available on the Tiltyard gaming server. Of those 14 games that failed, 5 cases failed because of inability to ground the game rules. Others games failed mostly because the goal condition was particularly complex.

A game that showed to be most problematic is Othello, because the grounded version of the goal is extremely big. Other games that failed include different versions of Chess, Amazons and Hex. On the other hand, there are some games, where the grounded game description is still rather big, but the goal condition itself is relatively simple. In this case, the action heuristic was generated successfully. Examples of such games are Breakthrough or Skirmish.

## 5.5 Summary and Future Work

The heuristic was utilized in the agent in three different search control schemes for MCTS and its effectiveness was demonstrated by comparing it with a pure MCTS/UCT, RAVE and MAST players. The combined heuristic agent outperform these players in well-known games like Nine Board Tic Tac Toe and Battle and with an exception of MAST player also in Pentago and Bidding Tic Tac Toe; however it shows significant loss ratio in

Checkers and Connect 4. When compared with other agents, the combined heuristic player can beat CadiaPlayer and Fluxplayer, but it loses against Sancho.

As it can be noted, the player performs well in any Tic Tac Toe-like games. At least partly this behaviour can be explained by the fact that these games typically contain many persistent fluents which are used to improve the heuristic. Thus, one idea for improving the quality of the heuristic in other games is to use more feature discovery techniques, such as the ones described in [16] or [18].

We can also see that the percentage of the time spent on evaluating the heuristic within the game tree is negligible, while it is significant when used in playouts. Naturally, this supports an idea to use more complex and more accurate heuristic within the tree control scheme, while less accurate and a lightweight version of it in the playout. Pruning of the heuristic formula could be used to only include the most relevant features in order to reduce the evaluation time, because it seems that some parts of the formula are triggered only in some relatively rare game states and do not contribute much to the overall result. However the question is, whether the simplified version will be good enough for playouts, as the playout heuristic seems to be prone to be misleading. As it has the most influence on the overall performance, this behaviour should be further investigated.

A possible cause for the high influence of the playout heuristic can be wrongly estimated value of $\tau$ constant. Further experiments should be therefore conveyed to investigate, whether increasing its value makes the heuristic less aggressive and thus possibly yielding better results in the combined scheme.

Another idea for future work would be to regress the goal condition by more than one step, however special care has to be taken to not increase the evaluation time too much.

# Conclusion

In this thesis, I give insights into the world of *General Game Playing* (GGP) agent systems; into the world, where agents have to understand formal description of a game in order to be able to play it. At first, it is explained, what general game players are and what are their peculiarities. Large part of the first chapter is devoted to the syntax and semantics of the *Game Description Language* (GDL), as it is crucial for such agents to understand game rules. I also define what kind of games is GDL capable to describe.

Later, I describe *Monte Carlo tree search* algorithm (MCTS) and give reasons, why it is so popular among current players. Moreover, this thesis gives a brief overview about well-known heuristic functions that can further improve its performance.

The third chapter is the core of this thesis, as I propose here a general method of creating a heuristic function. This is an action evaluation function, which means that it can be well incorporated into the MCTS algorithm plus it is based on a goal heuristic and as such it directly leads a player towards fulfilling a game goal. It can be used both withing a game tree and in playouts. The heuristic is generated from a goal condition using regression before the game starts, while it is evaluated during play time with fuzzy logic. Moreover, I give some examples of use of additional knowledge about the game that improves properties of the heuristic.

The following chapter is purely devoted to the implementation of a full-fledged GGP agent. While the heuristic generation part of the agent is written in Prolog, which is well suited for processing logic formulas; the heuristic evaluation part and the MCTS algorithm is implemented in Java with an aid of a GGP-base framework. I use a propositional network to evaluate the heuristic formula in the highest speed possible.

Finally, the agent using three heuristic schemes is tested in different settings in 14 games to demonstrate its effectiveness. First, it is compared against pure non-heuristic player; second, it is matched with well-established RAVE and MAST heuristic schemes and finally, it is opposed with other full-fledged GGP agents, namely with CadiaPlayer, Fluxplayer and Sancho. The agent performs very well in well-known games like Nine Board Tic Tac Toe (with 81 % win ratio) or Pentago, but it shows significant loss ratio in Checkers and Connect 4. In average, it performs significantly better than pure MCTS and RAVE scheme, but it is somewhat worse than MAST scheme. When compared with other agents, it clearly wins over Fluxplayer and CadiaPlayer, but loses against Sancho, which is a good result, as Sancho is currently the world's best GGP agent. The agent has also shown capability to generate the heuristic for vast majority of games that are publicly available.

Future work should investigate regression of a goal formula by more than one step. Moreover, additional feature discovery techniques can be implemented, as it was shown, that additional knowledge helps to improve the heuristic formula. However special care has to be taken not to increase the time required for evaluation and therefore some kind

of pruning of the heuristic formula should be involved. This can be possibly done by eliminating fluents that do not contribute much to the overall result.

Upon the results of this thesis, I have collaborated on the following paper that has been submitted to the GIGA15 conference:

- Michal Trutman and Stephan Schiffel. Creating action heuristics for general game playing agents. Technical report, Reykjavík University, 2015. Submitted to GIGA15.

# Bibliography


[1] Michael Genesereth and Michael Thielscher. *General Game Playing*. Morgan & Claypool, USA, 2014. ISBN 9781627052559.

[2] Jacques Pitrat. Realization of a general game-playing program. In *IFIP Congress (2)*, pages 1570–1574, 1968.

[3] Barney Pell. *Strategy Generation and Evaluation for Meta-Game Playing*. PhD thesis, University of Cambridge, 1993.

[4] Michael R. Genesereth, Nathaniel Love, and Barney Pell. General game playing: Overview of the AAAI competition. *AI Magazine*, 26(2):62–72, 2005.

[5] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*, chapter 2. Prentice-Hall, USA, 1995. ISBN 0-13-360124-2.

[6] Katie Salen and Eric Zimmerman. *Rules of Play: Game Design Fundamentals*, chapter 7. MIT Press, 2003. ISBN 0-262-24045-9.

[7] Nathaniel Love, Timothy Hinrichs, David Haley, Eric Schkufza, and Michael Genesereth. General game playing: Game description language specification. Technical report, Stanford University, March 2008.

[8] Stephan Schiffel. *Knowledge-Based General Game Playing*. PhD thesis, Technischen Universität Dresden, 2011.

[9] Michael Genesereth. Knowledge interchange format. Draft proposed American National Standard (dpANS), NCITS.T2/98-004, Stanford Logic Group, 1998.

[10] Sebastian Haufe, Stephan Schiffel, and Michael Thielscher. Automated verification of state sequence invariants in general game playing. *Artificial Intelligence*, 187-188:1–30, 2012.

[11] Krzysztof R. Apt and Elena Marchiori. Reasoning about prolog programs: From modes through types to assertions. *Formal Aspects of Computing*, 6(1):743–765, 1994. ISSN 0934-5043.

[12] Michael Thielscher. A general game description language for incomplete information games. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 994–999, Atlanta, July 2010. AAAI Press.

[13] Stephan Schiffel and Yngvi Björnsson. Efficiency of GDL reasoners. *IEEE Transactions on Computational Intelligence and AI in Games*, 2014.

[14] Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In *ECML*, page 282–293, 2006.

[15] Rémi Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. In *CG2006*, pages 72–83, 2006.

[16] Gregory Kuhlmann, Kurt Dresner, and Peter Stone. Automatic heuristic construction in a complete general game player. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence*, pages 1457–62, 2006.

[17] James Clune. Heuristic evaluation functions for general game playing. In *AAAI*, pages 1134–1139. AAAI Press, 2007.

[18] Stephan Schiffel and Michael Thielscher. Fluxplayer: A successful general game player. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence (AAAI-07)*, pages 1191–1196. AAAI Press, 2007.

[19] Sylvain Gelly and David Silver. Combining online and offline knowledge in UCT. In *Proceedings of the 24th International Conference on Machine Learning*, volume 227, pages 273–280, 2007.

[20] Shiven Sharma, Ziad Kobti, and Scott D. Goodwin. Knowledge generation for improving simulations in UCT for general game playing. In *Australasian Conference on Artificial Intelligence*, volume 5360. Springer, 2008.

[21] Hilmar Finnsson and Yngvi Björnsson. Simulation-based approach to general game playing. In *AAAI*. AAAI Press, 2008.

[22] Hilmar Finnsson and Yngvi Björnsson. Cadiaplayer: Search-control techniques. *KI*, 25(1):9–16, 2011.

[23] Mandy J. W. Tak, Mark H. M. Winands, and Yngvi Björnsson. N-grams and the last-good-reply policy applied in general game playing. In *IEEE Transactions on Computational Intelligence and AI in Games*, pages 73–83, 2012.

[24] Tom Pepels, Mandy J. W. Tak, Marc Lanctot, and Mark H. M. Winands. Quality-based rewards for Monte-Carlo tree search simulations. In *21st European Conference on Artificial Intelligence (ECAI 2014)*, pages 705–710. IOS Press, 2014.

[25] Hilmar Finnsson and Yngvi Björnsson. Learning simulation control in general game playing agents. In *AAAI*, pages 954–959. AAAI Press, 2010.

[26] Raymond Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*, pages 61–73. Massachusetts Institute of Technology, 2001.

[27] Stephan Schiffel and Michael Thielscher. Automated theorem proving for general game playing. In *Proceedings of IJCAI'09*, pages 911–916, 2009.

[28] Sam Schreiber. GGP base. `https://github.com/ggp-org/ggp-base`. Accessed: 2015-04-28.

[29] Steve Draper and Andrew Rose. Sancho GGP player. `http://sanchoggp.blogspot.com/`. Accessed: 2015-04-28.

[30] Sam Schreiber. GGP.org – Tiltyard gaming server. `http://tiltyard.ggp.org/`. Accessed: 2015-01-08.

# List of Abbreviations

| Acronym | Meaning | Explanation |
|---|---|---|
| AI | Artificial Intelligence | An antelligence exhibited by software agents. |
| AAAI | Association for the Advancement of Artificial Intelligence | A conference, where annual GGP competition is held. |
| GDL | Game Description Language | A formal language for describing game rules. |
| GGP | General Game Playing | An approach to game playing without knowing game rules beforehand. |
| KIF | Knowledge Interchange Format | An alternate syntax for GDL used for communication between players. |
| MAST | Move-Average Sampling Technique | An action heuristic based on average move value. |
| MCTS | Monte Carlo Tree Search | A simulation based state space search algorithm. |
| PAST | Predicate-Average Sampling Technique | An action heuristic based on average value of move-fluent pairs. |
| Propnet | Propositional Network | An alternate representation of game rules. |
| RAVE | Rapid Action Value Estimation | An action heuristic used within a MCTS game tree. |
| SLD | Selective Linear Definite (Resolution) | A method of finding answers to Prolog queries. |
| UCT | Upper Confidence Bounds Applied to Trees | A method of setting a trade-off between exploration and exploitation in MCTS. |

# List of Appendices

# Appendix A

# Game Rules

This appendix contains summary of rules of the games used in the thesis. Formal description of the rules in GDL can be found on the attached CD or on the Tiltyard gaming server [30]. All the following games are two player games.

## A.1 Tic Tac Toe

Tic Tac Toe is a turn-taking game played on a rectangular 3×3 board. The game starts with an empty board. A player in its turn marks any empty cell with its symbol (that is with $x$ for player $X$ and with $o$ for player $O$). A player, who first arranges 3 of its markers in a line, wins. The game results in a draw, when none manages to build a line and the board is full.

## A.2 Bidding Tic Tac Toe

The goal of this game is the same as in simple Tic Tac Toe, however players bid coins in order to mark a cell on a board. The bids occur simultaneously. Winning a bid allows the player to mark an empty cell in the next turn, however the amount of coins he bid is transferred to the other player. Players start the game with 3 coins.

## A.3 Pentago

Pentago is a variant of Tic Tac Toe, but it is played on a 6×6 board, that is split into 4 sub-boards, each having 3×3 cells. Players take turns, however each turn consists of two sub-turns. First, the player can rotate one of the sub-boards around its centre clockwise or anti-clockwise; second, it can mark any empty cell with its marker. A player, who first manages to build a line of five markers, wins.

## A.4 Nine Board Tic Tac Toe

This game, as its name suggests, consists of 9 ordinary Tic Tac Toe boards arranged in a 3×3 array. Players take turns making marks in the Tic Tac Toe boards. However, the board, where a mark can be placed must have the same coordinates as the cell, where the last mark was placed. The usual 3-in-a-row arrangement of marks in any of the 9 ordinary boards wins.

## A.5 Breakthrough

This game is played on a standard 8×8 chessboard, with each player having one side. Players start with 16 pawns on their side of a board. A player in its turn can move one of its pieces by one square forward or diagonally, where diagonal moves can capture opponent's piece. A player, whose pawn first reaches the opposite side of the board, wins.

## A.6 Skirmish

Skirmish is a Chess-like game, where players start with a king, a rook, two knights and four pawns. The pieces can move in the same way as in Chess and players get score by capturing opponent pieces. The game is over, when one of the players has no more pieces left.

## A.7 Blocker

Blocker is a simple simultaneous move game played on a rectangular 4×4 board. There are two different roles in this game: blocker and crosser. The aim of crosser is to connect the right and the left side of the board together with a line consisting of adjacent cells, while blocker tries to prevent this. Both players mark an empty cell in each turn; in case of both of them trying to mark the same cell, blocker takes priority.

# Appendix B

# CD Content

The attached CD contains the following directory structure:

- `\paper\` – source code of the paper submited to GIGA15 conference
- `\results\` – logs from the tests performed
- `\rules\` – rules in GDL of the games used
- `\src\` – source code of the agent
- `\src\ggp\` – source of the agent (in Java)
- `\src\ggp-sancho\` – source of Sancho module (including GGP-base framework)
- `\src\fluxplayer\` – source of Fluxplayer module
- `\src\fluxplayer\src\tester\michal\` – source of the agent (in Prolog)
- `\thesis\` – source code of this thesis