Simulation-Based General Game Playing

Hilmar Finnsson
Doctor of Philosophy
June 2012
School of Computer Science
Reykjavík University

Ph.D. DISSERTATION

ISSN 1670-8539

Simulation-Based General Game Playing

by

Hilmar Finnsson

Thesis submitted to the School of Computer Science
at Reykjavík University in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

June 2012

Thesis Committee:

Yngvi Björnsson, Supervisor
Associate Professor, Reykjavík University

Martin Müller
Professor, University of Alberta

Björn Þór Jónsson
Associate Professor, Reykjavík University

Mark Winands, Examiner
Assistant Professor, Maastricht University

Simulation-Based General Game Playing

Hilmar Finnsson

June 2012

Abstract

The aim of *General Game Playing (GGP)* is to create intelligent agents that automatically learn how to play many different games at an expert level without any human intervention. One of the main challenges such agents face is to automatically learn knowledge-based heuristics in real-time, whether for evaluating game positions or for search guidance.

In this thesis we approach this challenge with Monte-Carlo Tree Search (MCTS), which in recent years has become a popular and effective search method in games. For competitive play such an approach requires an effective search-control mechanism for guiding the simulation playouts. In here we describe our GGP agent, CADIAPLAYER, and introduce several schemes for automatically learning search guidance based on both statistical and reinforcement learning techniques. Providing GGP agents with the knowledge relevant to the game at hand in real time is, however, a challenging task. This thesis furthermore proposes two extensions for MCTS in the context of GGP, aimed at improving the effectiveness of the simulations in real time based on in-game statistical feedback. Also we present a way to extend MCTS solvers to handle simultaneous move games. Finally, we study how various game-tree properties affect MCTS performance.

Alhliða Leikjaspilun byggð á Hermunaraðferðum

Hilmar Finnsson

Júní 2012

Útdráttur

Markmið Alhliða Leikjaspilunar (General Game Playing, GGP) er að búa til forrit sem geta á sjálfstæðan hátt lært að spila marga mismunandi leiki og náð getu sérfræðings án þess að mannshöndin komi þar nærri. Ein aðal áskorunin við gerð slíkra forrita að geta á sjálfvirkann hátt búið til þekkingu í rauntíma sem hjálpar til við að meta stöður og leiðbeina leitarrekniritum.

Hér er reynt að nálgast þetta vandmál með leitaraðferð byggðri á hermunum sem nefnist Monte-Carlo Trjá Leit (Monte-Carlo Tree Search, MCTS). MCTS hefur á undanförnum árum náð miklum vinsældum vegna góðs árangur í spilun margs konar leikja. Til að vera keppnishæft þarf forrit sem beitir MCTS að hafa yfir að ráða öflugum aðferðum til að stýra útspilun hermana sinna. Það að sjá GGP forriti fyrir þekkingu er hentar þeim leik sem verið er að spila hverju sinni er mikil áskorun. Við kynnum forrit okkar í Alhliða Leikjaspilun, CADIAPLAYER, og ýmsar aðferðir sem við höfum þróað fyrir það sem læra að stýra leit sjálfvirkt með notkun tölfræði og styrkingarnáms. Að auki eru kynntar tvær nýjar viðbætur við MCTS í GGP sem nýta tölfræðilega endurgjöf við spilun leikja á árangursríkan hátt. Einnig er sýnd viðbót við MCTS sem gerir kleift að leysa leiki þar sem spilarar leika á sama tíma. Að lokum er kannað hvernig nokkur eigindi leikjatrjáa hafa áhrif á getu MCTS til að spila leiki.

Acknowledgements

I would especially like to thank my supervisor Yngvi Björnsson for his exceptional support throughout my research. Without his guidance and encouragement I would not be writing this thesis.

Special thanks to Martin Müller for the chance to visit the University of Alberta and his great hospitality and input while there.

Thanks go also to Abdallah Saffidine and Michael Buro for their collaboration on the work behind the MCTS solver, their input, and company during my stay at the University of Alberta.

This research was supported by grants from The Icelandic Research Fund for Graduate Students (RANNÍS), The Icelandic Centre for Research (RANNÍS) and by a Marie Curie Fellowship of the European Community programme *Structuring the ERA* under contract MIRG-CT-2005-017284.

Publications

Parts of the material in this thesis have been published in the following:

1. Hilmar Finnsson and Yngvi Björnsson. Simulation-Based Approach to General Game Playing. In Dieter Fox and Carla P. Gomes, editors, *Proceedings of the Twenty- Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 259-264. AAAI Press, 2008. (Finnsson & Björnsson, 2008)

2. Yngvi Björnsson and Hilmar Finnsson. CADIAPlayer: A Simulation-Based General Game Player. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(1):4-15, 2009. (Björnsson & Finnsson, 2009)

3. Hilmar Finnsson and Yngvi Björnsson. Simulation Control in General Game Playing Agents. In *GIGA'09 The IJCAI Workshop on General Game Playing, July 2009*. (Finnsson & Björnsson, 2009)

4. Hilmar Finnsson and Yngvi Björnsson. Learning Simulation Control in General Game-Playing Agents. In *Maria Fox and Dieter Poole, editors, Proceedings of the Twenty- Fourth AAAI Conference on Artificial Intelligence, AAAI 2008, Atlanta, Georgia, USA, July 11-15, 2010*, pages 954-959. AAAI Press, 2010. (Finnsson & Björnsson, 2010)

5. Hilmar Finnsson and Yngvi Björnsson. CadiaPlayer: Search-Control Techniques. *KI - Künstliche Intelligenz, 25(1):9-16, March 2011.* (Finnsson & Björnsson, 2011a)

6. Hilmar Finnsson and Yngvi Björnsson. Game-Tree Properties and MCTS Performance. In *GIGA'11 The IJCAI Workshop on General Game Playing, July 2011*. (Finnsson & Björnsson, 2011b)

7. Hilmar Finnsson. Generalizing Monte-Carlo Tree Search Extensions for General Game Playing. *Accepted for the Twenty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2012, Toronto, Ontario, Canada, July 22-26, 2012*. (Finnsson, 2012)

8. Abdallah Saffidine, Hilmar Finnsson and Michael Buro. Alpha-Beta Pruning for Games with Simultaneous Moves. *Accepted for the Twenty-Sixth AAAI Conference on Artificial Intelligence, Toronto, Ontario, Canada, July 22-26, 2012*. (Saffidine, Finnsson, & Buro, 2012)

All the above publications are available at:
`http://cadia.ru.is/wiki/public:cadiaplayer:main#publications`.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis contributes to the study of the *Monte-Carlo Tree Search* algorithm and the field of *General Game Playing*. Even though *Monte-Carlo Tree Search* sometimes works surprisingly well without any external knowledge, having access to any helpful knowledge can greatly improve the performance of the algorithm in practice. Applying knowledge is an especially difficult task in the field of *General Game Playing* (GGP). GGP systems must demonstrate skillful play in an wide range of games, some of which they have never encountered before. Furthermore they must do so in real time from rules that contain no explicit information on what can be considered advantageous knowledge. When confronted with such scenarios intelligent searching can bridge the gulf between being clueless and strategically intelligent.

1.1 Game-Tree Search

From the inception of the field of *Artificial Intelligence* (AI), over half a century ago, games have played an important role as a testbed and as an inspiration for advancements in the field. AI researchers have, over the decades, worked on building high-performance game-playing systems for games of various complexity capable of matching wits with the strongest humans in the world. After the realization of the first electronic computers it was not long until people started imagining how thinking could be simulated. This challenge led to *Chess* being proposed by Claude Elwood Shannon, "the father of information theory", as a starting point for unearthing the theory necessary to get us on the path towards intelligent machines. It would be an understatement to say that people were optimistic in terms of how close we were to creating computers and algorithms allowing human-like thought and the *Chess* problem of making a machine play better than any human ended up

taking almost 50 years. It was finally accomplished in 1997 with DEEP BLUE (Campbell, Hoane, Jr., & Hsu, 2002), a computer with highly specialized software and hardware for playing the game. Even though *Chess* proved to be a much bigger challenge than expected it did bring to light a cornucopia of theory for search in games. The *MiniMax* algorithm (Neumann & Morgenstern, 1944) is at the foundation of such algorithms, later refined into more efficient variants such as *Alpha-Beta* (Knuth & Moore, 1975), *NegaScout* (Reinefeld, 1983) and *Principal-Variation Search* (Marsland, 1983). These algorithms are often referred to as *Traditional Game-Tree Search*.

After *Chess* had been conquered, the game *Go* quickly became the new challenge. This is a game where programs, even though armed to the teeth with all the algorithms and lessons learned from *Chess*, were still no match for human amateurs. But with researchers now putting added effort into *Go*, the landscape of game-tree search was about to change. In 2006, a fundamentally different search method gave *Go* a much needed boost that has since then steadily pushed *Go* programs to a strong enough level to start challenging human *Go* professionals. This was the now well-known simulation-based search method *Monte-Carlo Tree Search* (Coulom, 2006). Even though *Monte-Carlo* methods had been known for some time they were nowhere close to achieving similar level of play as traditional methods had in *Chess*. The rise of Monte-Carlo methods in boardgames began with Abramson using them to estimate game-theoretical values for adversary games with complete information (Abramson, 1990) paving the way for their application to *Go* (Brügmann, 1993; Bouzy & Helmstetter, 2003). Simulation-based search then really received a renewed interest with the introduction of the UCT algorithm (Kocsis & Szepesvári, 2006) which expanded on the *Monte-Carlo* simulations making it applicable to game trees. Furthermore UCT was proven to converge to the best action of any state if given enough samples of game-tree paths.

1.2 The General Game Playing Problem

General Game Playing (GGP) (Genesereth, Love, & Pell, 2005) originated at the *Stanford Logic Group* and is an ambitious AI challenge. It revolves around having the game programs (agents) themselves be self-sufficient when coming up with successful strategies for never-before-seen puzzles and games. GGP encompasses games which allow:

- Any number of players.
- Simultaneous moves, turn-taking, or combination thereof.
- Any scoring system, be it zero-sum or not, greedy or cooperative.

The reasoning behind a challenge such as GGP is the controversy of calling a program intelligent when it mostly bases its success on the game-specific knowledge bestowed upon it by its creator. By forcing the agent to be prepared for such a vast set of problems it is futile to provide it with too game-specific information. In the best case any such knowledge must still be matched and most likely adapted to the problem at hand which pushes the agent to show intelligence of a higher level than just playing blindly from pre-programmed knowledge.

GGP uses a first-order logic language named *Game Description Language* (GDL) to describe games. It describes all games that fit the aforementioned criteria but has two constraints though: the games must be deterministic and having perfect information.

Ever since 2005 there has been an annual international GGP competition at either the *Association for the Advancement of Artificial Intelligence* (AAAI) or *International Joint Conference on Artificial Intelligence* (IJCAI) conferences. In 2011, the first *German Open GGP Competition* (GO-GGP) was held in context with the *34th German Conference on Artificial Intelligence* (KI 2011).

For interested researchers, GGP includes many AI subfields and challenges such as:

- Knowledge representation, discovery and transfer.
- Real-time and game-tree search.
- Path planning, adversary- and multi-player games.
- Reinforcement- and machine learning.

Recently a new separate GGP challenge has been put forth with the introduction of GDL-II (Thielscher, 2010; Schiffel & Thielscher, 2011) which allows non-determinism and imperfect information, but that is beyond the scope of this thesis.

1.3 CADIAPLAYER

CADIAPLAYER is our GGP agent, which has competed in all international GGP competitions since 2007. It used MCTS from the start and was the first GGP agent to do so. CADIAPLAYER won the competition in 2007 and 2008 and came second in 2011. The success CADIAPLAYER enjoyed had a big impact on the GGP community and nowadays almost all GGP agents use MCTS as their search mechanism.

In this thesis CADIAPLAYER will be our main testbed for studying and evaluating our algorithmic enhancements. It is publicly available at the CADIAPLAYER web site[1].

1.4 Contributions

The contributions of this thesis can be summarized as follows:

Building a Simulation-Based GGP Player : We show all relevant implementation details of CADIAPLAYER (Finnsson, 2007; Finnsson & Björnsson, 2008; Björnsson & Finnsson, 2009) necessary to build a GGP agent. CADIAPLAYER is also downloadable via the Internet on the webpages of the *Center for Analysis and Development of Intelligent Agents* (CADIA) at Reykjavík University.

Learning Simulation Control in GGP : Four original techniques for learning search-control are presented and evaluated in the context of GGP. Our simulation control techniques *Move-Average Sampling Technique* (MAST) (Finnsson, 2007; Finnsson & Björnsson, 2008; Björnsson & Finnsson, 2009; Finnsson & Björnsson, 2009, 2010, 2011a), *Tree-Only MAST* (TO-MAST) (Finnsson & Björnsson, 2009, 2010, 2011a) and *Predicate-Average Sampling Technique* (PAST) (Finnsson & Björnsson, 2009, 2010, 2011a) all aim at biasing simulation playout towards actions and states that show high statistical correlation with winning when they occur in simulation paths. *Features-to-Action Sampling Technique* (FAST) (Finnsson & Björnsson, 2010, 2011a) uses template matching to identify common board game features, currently detecting two such: piece types and cells (squares). We use TD(λ) (Sutton, 1988) to learn the relative importance of the detected features, e.g. the values of the different type of pieces or the value of placing a piece on a particular cell.

We also evaluate the effectiveness of *Rapid Action Value Estimation* (RAVE) (Gelly & Silver, 2007) algorithm in GGP, an effective search-control technique popularized in MCTS-based *Go* programs.

Generalized MCTS Extensions for GGP : We generalize two known search enhancements such that they do not require any a priori knowledge, thus making them applicable to GGP. The first, *Early Cutoffs*, terminates simulations prematurely that are unlikely to be useful. The second one, *Unexplored Action Urgency*, uses speculative meta-level knowledge to be more exploitative (Finnsson, 2012).

[1] `http://cadia.ru.is/wiki/public:cadiaplayer:main`

Simultaneous Moves Solver : This work is done in collaboration with Abdallah Saffidine and Michael Buro on accomplishing *Alpha-Beta* like pruning for games with simultaneous moves. As such games can be expressed in GGP we extend this pruning technique into a MCTS solver for CADIAPLAYER. This algorithm has the nice property of reducing to *Score Bounded MCTS* (Cazenave & Saffidine, 2011) in turn-taking games, which makes it at least as powerful as established MCTS solvers with the added bonus of handling simultaneous games (Saffidine et al., 2012).

Examination of Game-Tree Properties and MCTS : High-level properties that are commonly found in game trees to a varying degree are identified and their effect on the performance of MCTS is measured. As a testbed we use simple custom made games that both highlight and allow us to vary the properties of interest. This work provides insight into how MCTS behaves when faced with these different game properties. In particular, we investigate the balance between the tree's height and width, the importance of constant game progression towards a meaningful terminal state, and finally how a MCTS weakness we name *Optimistic Moves* can adversely affect simulation performance (Finnsson & Björnsson, 2011b).

1.5 Overview

The thesis is structured as follows. In Chapter 2 gives the necessary background for the work. GGP is explained in detail along with the GDL language it uses and the GGP competition setup and agent communication protocol are explained. Furthermore it describes MCTS, its structure, development and main extensions. Successful applications of the algorithm are highlighted and discussed. CADIAPLAYER is the subject of Chapter 3. The agent is described in detail down to the design and implementation level in an effort to help with and shed light on what it takes to build a GGP agent.

Chapter 4 explains and empirically evaluates our four contributions to simulation control of MCTS, as well as RAVE, in GGP. In Chapter 5 we focus on the *Early Cutoffs* and *Unexplored Action Urgency* MCTS extensions and how they can be adapted to the generality and absence of knowledge inherent to GGP.

Chapter 6 discusses the new pruning algorithm for adversary zero-sum games with simultaneous moves. This was joint work with Abdallah Saffidine and Michael Buro. In this thesis we approach this algorithm from the perspective of utilizing it as an MCTS solver and show how to extend it into CADIAPLAYER and evaluate it.

For Chapter 7 we turn our attention towards the three aforementioned game-tree properties and measure how they affect the performance of MCTS.

In Chapter 8 we discuss work done for both GGP and MCTS that is related to the contributions this thesis presents and finally we conclude in Chapter 9 where we summarize and suggest future work.

Chapter 2

Background

This chapter gives background information on the context of the thesis research. We first discuss the field of *General Game Playing* followed by explaining the *Monte-Carlo Tree Search* method for game-tree searching.

2.1 General Game Playing

Games have always been crucial to AI research as a benchmark for achieving systems capable of human-level intelligence and strategizing (Campbell et al., 2002; Schaeffer, 1997; Buro, 1999). The importance of having such an objective measure of the progress of intelligent systems cannot be overestimated, but nonetheless, this approach has led to some adverse developments. For example, the focus of the research has to some extent been driven by the quest for techniques that lead to immediate improvements to the game-playing system at hand, with less attention paid to more general concepts of human-like intelligence like acquisition, transfer, and use of knowledge. The success of game-playing systems has thus in part been because of years of relentless knowledge-engineering efforts on behalf of the program developers, manually adding game-specific knowledge to their programs. The aim of general game playing is to move beyond such dependency, making the systems themselves responsible for the knowledge they use.

In *General Game Playing (GGP)* the goal is to create intelligent agents that can automatically learn how to skillfully play a wide variety of games, given only the descriptions of the game rules. This requires that the agents learn diverse game-playing strategies without any game-specific knowledge being provided by their developers. A successful realization of this task poses interesting research challenges for artificial intelligence sub-disciplines

such as knowledge representation, agent-based reasoning, heuristic search, computational intelligence, and machine learning.

The Logic Group at Stanford University initiated the *General Game Playing Project* in 2005 to facilitate further research into the area, along with the annual GGP competition. For this purpose they provide both a well-defined language for describing games and a web-based server for running and viewing general game playing matches.

Similar ideas of general game-playing programs were introduced by Jacques Pitrat in (Pitrat, 1968) and later by Barney Pell with his program METAGAMER (Pell, 1996), which could play a wide variety of simplistic chess-like games.

2.1.1 Game Description Language

Games are specified in a *Game Description Language* (GDL) (Love, Hinrichs, Haley, Schkufza, & Genesereth, 2008), a specialization of the *Knowledge Interchange Format* (KIF) (Genesereth & Fikes, 1992), a first-order logic based language for describing and communicating knowledge. It is a variant of *Datalog* that allows function constants, negation, and recursion (in a restricted form). The expressiveness of GDL allows a large range of deterministic, perfect-information, simultaneous-move games to be described, with any number of adversary or cooperating players. Turn-based games are modeled by having the players who do not have a turn return a special no operation move (often referred to as *no-op*). A GDL game description uses keywords known as relations to specify the initial game state, as well as rules for detecting and scoring terminal states and for generating and playing legal moves. A game state is defined by the set of propositions that are true in that state. Only the relations have lexical meaning and during competitions everything else is obfuscated.

Following is a brief overview of GDL, using the partial Tic-Tac-Toe description in Figure 2.1 as a reference. A complete GDL description for Tic-Tac-Toe is provided in Appendix A.

The *role* relation lists the players participating in the game; arbitrarily many roles can be declared, that is, a game can be single-player (i.e. a puzzle), two-player, or multi-player. However, once declared the roles are fixed throughout the game. In our Tic-Tac-Toe example two players are defined, *xplayer* and *oplayer*. The *init* relation states the facts that are true in the initial state, and they are added to the knowledge-base. Here a game state is represented by the board position (initially all cells are empty) and whose turn it

```
(role xplayer)
(role oplayer)

(init (cell 1 1 b))
(init (cell 1 2 b))
...
(init (control xplayer))

(<= (legal ?w (mark ?x ?y))
    (true (cell ?x ?y b))
    (true (control ?w)))
(<= (legal oplayer noop)
    (true (control xplayer)))
...
(<= (next (cell ?m ?n x))
    (does xplayer (mark ?m ?n))
    (true (cell ?m ?n b)))
(<= (next (control oplayer))
    (true (control xplayer)))
...
(<= (row ?m ?x)
    (true (cell ?m 1 ?x))
    (true (cell ?m 2 ?x))
    (true (cell ?m 3 ?x)))
...
(<= (line ?x)
    (row ?m ?x))
(<= (line ?x)
    (column ?m ?x))
(<= (line ?x)
    (diagonal ?x))
...
(<= (goal xplayer 100)
    (line x))
(<= (goal xplayer 0)
    (line o))
...
(<= terminal
    (line x))
```

Figure 2.1: A partial Tic-Tac-Toe GDL description

is to move. The *true* relation is used in GDL to check whether a fact is in the knowledge-base.

The *legal* and *next* relations are used to determine legal moves and execute them, respectively. In Figure 2.1 the player having the turn can mark any empty cell on the board (variables in GDL are prefixed with a question mark), whereas the opponent can only play a *no-op* move. Moves are executed by temporarily adding a *does* fact to the knowledge-base stating the move played, and then call the *next* clause to determine the resulting state. Figure 2.1 shows, for one of the players, the *next* relations for updating the cell played into and the turn to move; symmetrical *next* relations are needed for the other player, as well as relations for stating that the untouched cells retain their mark (see Appendix A). The facts that the *next* call returns are added to the knowledge-base, replacing all previous state information stored there (the temporary *does* fact is also removed). This knowledge representation formalism corresponds to the closed-world assumption; that is, the only true facts are those that are known to be so, others are assumed false.

The *terminal* relation indicates when a game position is reached where the game is over, and the *goal* relation returns the value of the current game state; for terminal states this corresponds to the game outcome. The goal scores are in the range $[0 - 100]$, and if a game is coded to provide intermediate goal values for non-terminal positions, then GDL imposes the restriction that the values are monotonically non-decreasing as the game progresses. Examples of such a scoring system are games where players get points based on the number of pieces captured. In Figure 2.1 user-defined predicates are also listed, for example *line*, *row*, *column*, and *diagonal*; arbitrary many such predicates are allowed.

The official syntax and semantics of GDL are described in (Love et al., 2008).

2.1.2 GGP Communication Protocol

The *Game Master* (GM) is a server for administrating matches between GGP agents. It does so via an HTTP-based communication protocol. Before a game starts the GGP agents register with the server. Each new match game gets a unique identification string. Play begins with a start message being sent to all the agents, containing the match identifier, the GDL game description, the role of the agent, and the time limits used. After all players have responded, play commences by the GM requesting a move from all players; subsequent move requests sent by the GM contain the previous round moves of all players. This way each player can update its game state in accordance with what moves the other players made. This continues until the game reaches a terminal state. If a player sends an illegal move to the GM, a random legal move is selected for that player.

The time limits mentioned above for preparing and playing are positive integer values called *startclock* and *playclock*. The time limit values are presented in seconds; the *startclock* indicates the time from receiving the rules until the game begins and the *playclock* the time the player has for deliberating each move.

More details, including the format of the HTTP message contents are in (Love et al., 2008). A full description of the GM capabilities is given in (Genesereth et al., 2005).

2.2 Monte-Carlo Tree Search

Monte-Carlo simulations play out random sequences of actions in order to make an informed decision based on aggregation of the simulations' end results. The most appealing aspect of this approach is the absence of heuristic knowledge for evaluating game states. Monte-Carlo Tree Search (MCTS) applies Monte-Carlo simulations to tree-search problems, and has nowadays become a fully matured search method with well defined parts and many extensions. In recent years MCTS has done remarkably well in many domains. The reason for its now diverse application in games is mostly due to its successful application in the game of *Go* (Coulom, 2006; Gelly, Wang, Munos, & Teytaud, 2006; Enzenberger & Müller, 2009), a game where traditional search methods were ineffective in elevating the state-of-the-art to the level of human experts. Other triumphs of MCTS include games such as *Amazons* (Lorentz, 2008), *Lines-of-Action* (Winands, Björnsson, & Saito, 2010), *Chinese Checkers* (Sturtevant, 2008), *Kriegspiel* (Ciancarini & Favini, 2009) and *Settlers of Catan* (Szita, Chaslot, & Spronck, 2009).

Monte-Carlo Tree Search (MCTS) is at the core of CADIAPLAYER's reasoning engine and was first applied to GGP in the 2007 GGP competition (Finnsson & Björnsson, 2008). This proved to be very successful where it outplayed more traditional heuristic-based players in many types of games. However, in other type of games, such as many chess-like variants, the MCTS-based GGP agent was hopeless in comparison to its $\alpha\beta$-based counterparts.

MCTS continually runs simulations to play entire games, using the result to gradually build a game tree in memory where it keeps track of the average return of the state-action pairs played, $\mathcal{Q}(s, a)$. When the deliberation time is up, the method plays the action at the root of the tree with the highest average return value.

Figure 2.2 depicts the process of running a single simulation: the start state is denoted with S and the terminal state with T. Each simulation consists of four strategic steps: *selection*, *playout*, *expansion*, and *back-propagation*.

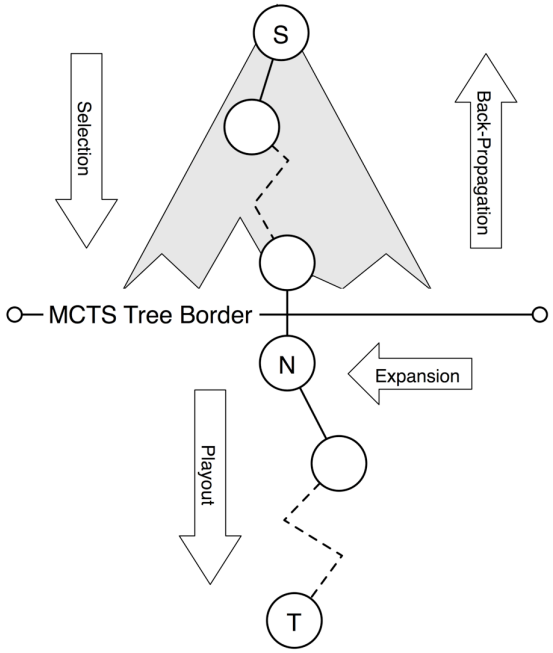

Figure 2.2: An overview of a single simulation

2.2.1 Selection

The selection step is performed at a beginning of a simulation and chooses actions while still in the tree (upper half of figure). The *Upper Confidence Bounds applied to Trees (UCT)* algorithm (Kocsis & Szepesvári, 2006) is customarily used in the selection step, as it offers an effective and a sound way to balance the exploration versus exploitation tradeoff. At each visited node in the tree the action a^* taken is selected by:

$$a^* = argmax_{a \in A(s)} \left\{ \mathcal{Q}(s,a) + C\sqrt{\frac{\ln N(s)}{N(s,a)}} \right\}$$

The $N(s)$ function returns the number of simulation visits to a state, and the $N(s,a)$ function the number of times an action in a state has been sampled. $A(s)$ is the set of possible actions in state s; if it contains an action that has never been sampled before, is selected by default as it has no estimated value. If more than one action is still without an estimate, a random tie-breaking scheme is used to select the next action. The term added to $\mathcal{Q}(s,a)$ is called the *UCT Bonus*. It is used to provide a balance between exploiting the perceived best action and exploring the less favorable ones. Every time an action is selected the bonus goes down for that action because $N(s,a)$ is incremented, while its siblings have their UCT bonuses raised as $N(s)$ is incremented. This way, when good actions have been sampled enough to give an estimate with some confidence, the bonus

of the suboptimal ones may have increased enough for them to get picked again for further exploration. If a suboptimal action is found to be good, it needs a smaller bonus boost (if any) to be picked again, but if it still looks the same or worse it will have to wait longer. The C parameter is used to tune how much influence the UCT bonus has on the action selection calculations. In Figure 2.2 the selection phase lasts from state S up to but not including the terminal state N.

2.2.2 Playout

At some point during the simulation it will fall out of the in-memory MCTS tree. This marks the start of the Playout phase. In the playout step there are no $\mathcal{Q}(s,a)$ values available for guiding the action selection, so in the most straightforward case one would choose between those available uniformly at random. However, CADIAPLAYER uses more sophisticated techniques for biasing the selection in an informed way, as discussed in later chapters. This phase includes everything from node N up to and including node T in Figure 2.2.

2.2.3 Expansion

The name of this phase refers to expanding the in-memory MCTS tree. A typical strategy is to append only one new node to the tree in each simulation: the first node encountered after stepping out of the tree (Coulom, 2006). This is done to avoid using excessive memory, in particular if simulations are fast. In Figure 2.2 the node added in this episode is labeled as N.

This way the tree grows most where the selection strategy believes it will encounter its best line of play.

2.2.4 Back-Propagation

The back-propagation phase controls how the end reward of the simulation is used to affect the traversed nodes of the current MCTS simulation path. In the basic implementation, just as with MC simulations, this keeps track of the average rewards. To distinguish node rewards at different distances from terminal state, a discounting factor can be applied to the reward as it is backed up from the terminal state to the root.

2.3 Summary

In this chapter we gave the necessary background for GGP and MCTS which make up the foundation for our research and therefore our main research tool, CADIAPLAYER. In the next chapter we examine the architecture and implementation of CADIAPLAYER.

Chapter 3

CADIAPLAYER

gggAn agent competing in the GGP competition requires at least three components: an HTTP server to interact with the GM, the ability to reason using GDL, and the AI for strategically playing the games presented to it. Our agent, CADIAPLAYER gets its name from the AI research lab at Reykjavik University, Center for Analysis and Design of Intelligent Agents (CADIA). The player was created to evaluate and showcase our research. It has participated in the international GGP competition regularly since 2007. The agent source code is publicly available at the CADIAPLAYER web site[1].

3.1 Overview

An overview of CADIAPLAYER's architecture is shown in Figure 3.1. The topmost layer of the figure is an HTTP server which runs the rest of the system as a child process and communicates with it via standard pipes. Every time a new game begins a new child process is spawned and the old one is suspended.

The game-playing engine is written in C++ and can be split up into three conceptual layers: the *Game-Agent Interface*, the *Game-Play Interface* and the *Game-Logic Interface*. The Game-Agent interface handles external communications and manages the flow of the game. It queries the Game-Play interface for all intelligent behavior regarding the game. In the Game-Logic interface the state space of the game is queried and manipulated. Because we do not have a specialized game engine, and are inferring legal moves, state transition, etc. from first-order-logic, a considerable slowdown in action and state generation is to be expected compared to specialized agents. In our experience this slowdown

[1] `http://cadia.ru.is/wiki/public:cadiaplayer:main`

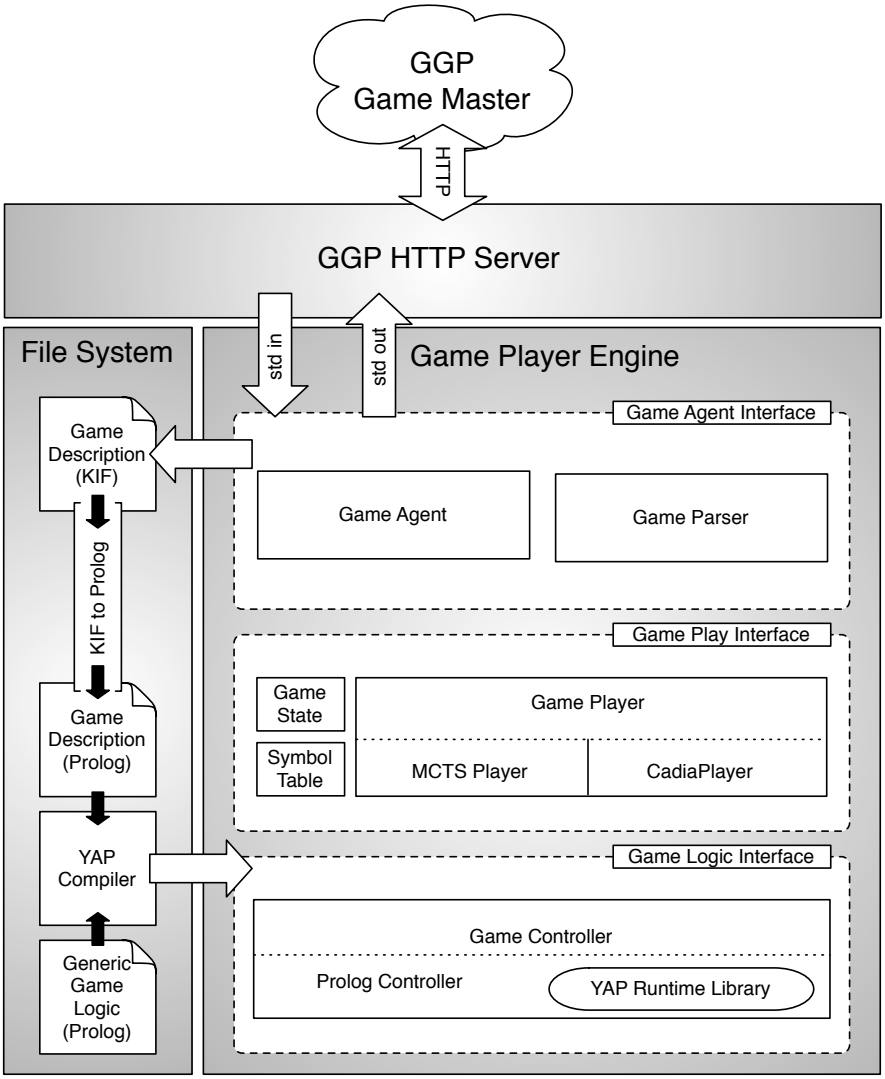

Figure 3.1: Overview of the architecture of CADIAPLAYER

can in the worst case be several orders of magnitude, depending on how the games are encoded in the GDL.

We now give an overview of CADIAPLAYER's architecture and how the agent interacts with the GM. The subsections that follow describe individual components of the agent in more detail.

A game begins when the *GGP HTTP Server* gets a message from the GM announcing a new game. The server starts a new CADIAPLAYER Child Process, extracts the content of the HTTP request, which is the GM message to the player, and relays it to the process through a standard pipe. It then waits for a reply from the process before responding to the HTTP request. On the other end of the pipe the *Game-Agent Interface* is waiting for the message and channels it into the *Game Agent* which recognizes it as an announcement

of a new game. The *Game Description* included in the message is both written to a file and sent through the *Game Parser*. The *Game Parser* initializes the *Game-Play Interface* with data from the game description and hands it back to the *Game Agent*. The *Game Agent* proceeds to selecting the type of *Game Controller* to use. CADIAPLAYER uses the *Prolog Controller* which utilizes the Prolog engine YAP (YAP, n.d.). When the *Prolog Controller* starts, it locates the game description file saved earlier and runs it through an external program that converts *KIF* to *Prolog* code and saves it to another file. Then it calls the *YAP Compiler* on the *Prolog Game Description* and a handcrafted file containing some *Generic Game Logic*. The compiled Prolog code is then loaded into memory through the *YAP Runtime Library* and is used to represent the state space for the new game. This selection varies between games. All this processing is happening on the startclock and for its remainder the *Game Player* is allowed to prepare for the game (e.g., start running simulations). When the start clock is up a message indicating that CADIAPLAYER is ready to play is returned back through the pipe to the *GGP HTTP Server* so it can notify the GM.

When CADIAPLAYER has announced that it is ready a new HTTP request can be expected. This time the GM is requesting a move decision. As before, the message is sent through the pipe and ends up in the *Game Agent*. It then queries the *Game Player* for a move decision. The *Game Player* makes its decision by using the *Game-Play Interface* it is plugged into to get information about the game. Once a decision is reached it is returned back to the *Game Agent* which relays it back to the *GGP HTTP Server* to be sent to the GM.

There is a difference between the first move and the rest of them. After the first one, all move requests from the GM contain the list of moves that were made in the last round by all players participating in the game. CADIAPLAYER uses this move list to update the state space in the *Game-Logic Interface* so it reflects the current state. The move list is parsed with the *Game Parser* into a structure the *Game-Play Interface* understands. The *Game-Logic Interface* makes a transition in its state space based on these moves. Finally the *Game Player* is queried for a move decision for the GM. This exchange of move lists and move decisions is now repeated until the game ends.

3.2 Game-Agent Interface

It is the responsibility of this layer to initialize the *Game-Play Interface* (see Section 3.3) and the *Game-Logic Interface* (see Section 3.4). This interface simply hides away all the

Table 3.1: Game Agent

Function	Description
Constructor	Takes a pointer to an instance of the Game Player interface (see Section 3.3.1) to be used and a setting indicating the type of *Game Controller* to be instanciated if the default should not be used.
setLogFile	Change the name of the log file. Defaults to "agent.log".
setPGNFile	Change the name of a resulting PGN file. Defaults to "[match id].pgn". No used for single player games.
run	Start monitoring the standard pipe for input.

nuances of understanding the text based messages from the *HTTP Server*. This is done by parsing the various messages of the GM and matching their type and data to functions of the *Game Player Interface* that will do the necessary work. The same goes for converting the returned data into messages that are automatically sent back to the *HTTP Server* which forwards it to the GM. The lexical analyzer of the parser was built using *Flex* (Flex, n.d.). This layer also logs to file any runtime information about the player and any statistical data it produces in between moves. This data is retrieved through special functions of the *Game-Play Interface*. When a game with two or more players ends, a *Portable Game Notation* (PGN) file (PGN Spec, n.d.) is generated from the game history.

In the CADIAPLAYER design framework the *Game-Agent Interface* is implemented as the *main* function of the agent. It has a concrete agent class that only needs to be provided with data on what implementations to use for the other two interfaces and the names for the log files. The interface to this agent class is in Table 3.1 and a C++ implementation sample is shown in Figure 3.2.

3.3 Game-Play Interface

The *Game-Play Interface* manages the main AI part of the player. It keeps track of the current state of the game and the history which led to it. Game Player implementations plug into this layer and use its services to run their algorithms to decide on which action to take. In order to do so this interface offers a robust class structure to represent states and actions and the ability to translate those structures back into GDL/KIF. As everything is represented by strings in KIF and it is inefficient to do string comparisons when looking up and storing, a symbol table is set up when the game description is parsed. Every atom and variable are assigned an unique unsigned integer value. As GDL demands that the names of the atoms have no lexical meaning, no information is lost by this transfor-

```
#include "agent.h"
#include "../play/players/uctplayer.h"
int main(int argc, char *argv[])
{
     cadiaplayer::play::players::UCTPlayer player;
     cadiaplayer::agent::Agent agent(&player);
     agent.setLogFile("uctagent.log");
     agent.setPGNFile("uctagent.pgn");
     exit(agent.run(argc, argv));
};
```

Figure 3.2: Sample C++ main function in the CADIAPLAYER framework

mation. Everything in the player uses this numeric representation, making comparisons and hashing more efficient. The original strings can be looked up in the symbol table for output from the player.

A game state is represented by multiple compound logical sentences, i.e., the logical facts that are true in it. The game history is stored on a stack and the state on top is the current state. The *Game-Agent Interface* is responsible for making any state changes that occur in the actual game being played, as only actions that the GM announces can be used to advance the game. This is important as even though the player sent a legal move, it is possible that it reached the GM too late and it selected a random move on behalf of the player instead. Without obeying the GM the player would not guaranteed to be in the same state as the GM.

3.3.1 Game Player

Game Player is a virtual class containing shared player logic and describes the interface a player must implement to be plugged into the framework. The *Game-Agent Interface* can then query the Game Player for move decisions given the state of the *Game-Play Interface* it is plugged into. Table 3.2 shows details of the *Game Player Interface*.

3.3.2 MCTS Game Player

Even though the CADIAPLAYER interfaces make no special distinction between traditional search or MCTS agents, the research involving CADIAPLAYER has always been focused on MCTS. The MCTS search procedure is at the heart of CADIAPLAYER and it is designed to expose certain critical injection points for inheritance rather than being

Table 3.2: Game Player

Function	Description
newGame	Should be called when the underlying game is reset or a different one is started.
setRole	Set the role of the player. Implemented in the base class.
getRole	Get the role of the player. Implemented in the base class.
prepare	Tells the player that the startclock has been triggered.
play	Queries the player for what action it would take given the current state.
postplay	Called when a play message has been sent back to the GM, in case the player wants to do any work outside the playclock before a new move request is received.
asyncplay	Called repeatedly while waiting for a new message from the GM, taking advantage of any delay that may be caused by the other participants or the GM itself. The player can e.g. run extra simulations.
stop	Called when the GM reports the game being over.
getPlayerName	For logging purposes only, to identify the player implementation being used.
getLastPlayInfo	For logging purposes, should contain description or some statistical data about what the player was thinking during the last call to the *play* function.
isSolved	Players should return true if they have constructed a solution for the game and actually require no more calculations to win the game if they are playing adversary- or multiplayer games or to get maximum points in single player games. This can be useful to the Game Agent layer to make a decision about whether it should change player implementation during runtime.
setThinkTime	Set the think time allowed for the player. Implemented in the base class.
startTimer	Starts an internal timer for the player that is used to measure for how long the player has been thinking. Implemented in the base class.
hasTime	Checks if the internal timer has exceeded the think time allowed for the player. Implemented in the base class.
getLastPlayConfidence	If multiple agents with varying types of game players are being run this function should return a move and a value indicating the confidence in this being the best move. Should only be used when selecting move to send to the GM.
parallelInfo	Get data for root parallelization (See Section 3.3.6).

Algorithm 1 search(ref qValues[])

```
1: if isTerminal() then
2:    for all r_i in getRoles() do
3:       qValues[i] ← goal(i)
4:    end for
5:    atTerminal(qValues)
6:    return
7: end if
8: playMoves ← ∅
9: for all r_i in getRoles() do
10:   moves ← getMoves(r_i)
11:   move ← selectMove(moves, gameTree[i])
12:   playMoves.insert(move)
13:   moves.clear()
14: end for
15: make(playMoves)
16: search(qValues)
17: retract()
18: for all r_i in getRoles() do
19:   qValues[i] ← γ * qValues[i]
20:   update(playMoves[i], qValues[i], gameTree[i])
21: end for
22: return
```

rewritten for each new idea. Algorithm 1 gives an overview of the search implementation used by CADIAPLAYER. Most of the functions called within this algorithm represent these injection points. The function *atTerminal* can be overwritten when something must be done at the terminal states and *selectMove* encapsulates both the selection and playout phase strategies and will be discussed in more detail in a short while. The function *make* is called any time the game advances, *search* is the algorithm's recursive call, *retract* restores the game to its previous state. Finally, inheriting players having data that needs to be updated during the back-propagation phase can do so by overwriting the *update* function. The discount factor γ is configurable, but by default set slightly less than one, or 0.999, for the algorithm to prefer earlier rather than later payoffs, as longer playouts have higher uncertainty.

The *selectMove* function controls both the selection and playout phase strategies. This function is described by Algorithm 2. It starts off with initialization (lines 1 to 5), then for each move available it checks whether there already exists a node for it in the MCTS tree (line 7) to distinguish between the selection and playout phases. If the node does not exist we are in the playout phase which is handled by querying each move for a playout-strategy value that will be fed to a Gibbs sampling routine dictating the distribution of the

random selection associated with the playout phase (lines 8 to 13). If the playout value is constant for all moves this reduces to the uniform random distribution. Alternatively, if the MCTS tree contains the node we collect the set of actions that have the highest selection value (lines 14 to 22) for tie breaking. For a simple greedy agent the average value of the looked up node can be used. Finally a decision is made if the selected move should come from the unexplored set of playout moves or the explored set of the selection moves (lines 24 to 29).

There are three key intercept points in this function. These are the functions *playoutMoveValue*, *selectionNodeValue* and *unexploredValue*. The *playoutMoveValue* can be overwritten to change the playout strategy as long as the intention is that the end result will be selected using Gibbs sampling[2]. The *selectionNodeValue* function controls the selection phase strategy, e.g., the UCT formula. By default all unexplored moves have priority in being tried at each state. This can be changed by overwriting the *unexploredValue* function. Its default implementation always returns a number representing positive infinity, but if it were to return a value lower than the selection strategy value the function will exploit the selection strategy move.

3.3.3 Game Tree

The MCTS player builds a game tree in memory storing simulation results. The game tree is modeled in the *Game-Play Interface* of the architecture.

The game tree uses 64-bit Zobrist keys (Zobrist, 1970) to identify its states and actions for fast retrieval. When the game tree is initialized against a game description, it builds multiple maps where each symbol in the symbol table is assigned a 64-bit pseudo-random number (key). The number of maps equals the highest symbol table entry id (which is zero based and incrementing) plus the highest number of predicate parameters encountered in the game description. Whenever a new state is encountered or a state lookup is needed, an identifier for it is calculated as follows: As a state is represented by multiple compound logical sentences, each containing possibly nested compound sentences, the calculations take all symbols of every sentence into account to make the identifier unique. An identifier for each of the compound sentences is retrieved by applying XOR to the Zobrist keys for the atoms. These identifiers are then combined into a single identifier for the state using XOR. The multiple maps are to counteract the chance of getting non-unique identifiers by having the map selected for fetching a key for an atom be relevant both to the preceding predicate and parameter position of the atom (if any). This would happen with only a

[2] Chapter 4 introduces a number of playout strategies that fit this criterion.

Algorithm 2 selectMove(moves[], gameTree)

1: $actionValue \leftarrow -\infty$
2: $currentValue \leftarrow -\infty$
3: $tiebreakCount \leftarrow 0$
4: $gibbsCount \leftarrow 0$
5: $gibbsDivider \leftarrow 0$
6: **for all** $move$ in $moves$ **do**
7: $node \leftarrow gameTree.get(state, move)$
8: **if** $node = \emptyset$ **then**
9: $gibbsMoves[gibbsCount] \leftarrow move$
10: $gibbsValues[gibbsCount] \leftarrow e^{(playoutMoveValue(state,move)/getTemperature())}$
11: $gibbsDivider \leftarrow gibbsDivider + gibbsValues[gibbsCount++]$
12: **continue**
13: **end if**
14: $actionValue \leftarrow selectionNodeValue(state, node)$
15: **if** $currentValue = actionValue$ **then**
16: $tiebreakMoves[tiebreakCount++] \leftarrow move$
17: **else if** $currentValue < actionValue$ **then**
18: $currentValue \leftarrow actionValue$
19: $selectedAction \leftarrow move$
20: $tiebreakCount \leftarrow 0$
21: $tiebreakMoves[tiebreakCount++] \leftarrow move$
22: **end if**
23: **end for**
24: **if** $gibbsCount > 0$ & $currentValue < unexploredValue(state, moves)$ **then**
25: $selectedMove \leftarrow gibbsMoves[selectGibbsDistribution(gibbsValues, gibbsDivider)]$
26: **else if** $tiebreakCount > 1$ **then**
27: $selectedMove \leftarrow tiebreakMoves[rand()\%tiebreakCount]$
28: **end if**
29: **return** $selectedAction$

single key map as compound sentences that should have different meaning get the same identifier when their parameters are rearranged. Algorithms 3 and 4 shows how we can get an identifier for a state and compound sentence which can recursively call itself for any nested compounds respectively. Line 6 in Algorithm 4 ensures the uniqueness concerning differently arranged parameters.

Calculating available actions in Prolog is computationally expensive so in practice, when a state is added to the MCTS tree we actually store all the available actions, not just the selected one. This way we can immediately start to bypass Prolog for fetching this state's actions when next encountered. The tree is implemented such that all state entries are references to a transposition table which keeps track of the number of references it has in the tree. Every time the GM progresses the game all the state references at the expired

Algorithm 3 getZKey(State s)

1: $key \leftarrow 0$
2: **for all** $c_i \in s.compounds$ **do**
3: $\quad key \leftarrow key \oplus getZKey(c_i, 0)$
4: **end for**
5: **return** key

tree level are deleted along with all the actual states in the transposition table they refer to if they have no more references after their reference counter has been decremented. The transposition table is a simple hash map keyed on the 64-bit state identifiers.

3.3.4 The Opponent Models

To get the best performance out of MCTS CADIAPLAYER must model not only its own role, but also the ones of the other players. So for each opponent in the game a separate game-tree model is used. Because GGP is not limited to two-player zero-sum games, the opponents cannot be modeled simply by using the negation of our return value. Any participant can have its own agenda and therefore needs its own action-value function. All these game-tree models work together when running simulations and control the UCT action selection for the player they are model. Algorithm 1 shows how the opponent models are combined with MCTS in CADIAPLAYER. The *GameTrees* array stores the different models. The functions *selectMove* and *update* use the corresponding model at position i to make move selections and updates.

3.3.5 Single-Agent Games

In the 2007 GGP competition a rudimentary implementation of the memory-enhanced IDA* (Reinefeld & Marsland, 1994) search algorithm was used as the Game Player for the single-agent games. However, if no (partial) solution was found on the startclock then the agent fell back on using the UCT algorithm on the playclock. For the 2008 GGP competition a scheme for automatically deriving search-guidance heuristics for single-agent games was developed, using a relaxed planning graph in a similar manner as heuristic search-based planners do (Hoffmann & Nebel, 2001). The heuristic (Guðmundsson, 2009) was used to guide a time-bounded A*-like algorithm (Björnsson, Bulitko, & Sturtevant, 2009) on both the startclock and the playclock. In the preliminaries we initially used this scheme alongside MCTS/UCT, picking the move from whichever method promised the higher score. However, the scheme proved neither sufficiently efficient nor robust

Algorithm 4 getZKey(Compound c, index m)

1: $key \leftarrow keymaps[m].getKey(c.symbol())$
2: **if** $c.arguments = \emptyset$ **then**
3: **return** key
4: **end if**
5: **for all** $c_i \in c.arguments$ **do**
6: $key \leftarrow key \oplus getZKey(c_i, key + i)$
7: **end for**
8: **return** key

enough across different games, and was consequently suspended. Instead, we made several small adjustments to MCTS/UCT to make it better suited to handle single-agent games.

First, the absence of an adversary makes play deterministic in the sense that the game will never take an unexpected turn because of an unforeseen move by the opponent. The length of the solution path therefore becomes irrelevant and the discount parameter unnecessary and possibly even harmful; no discounting was thus done ($\gamma = 1.0$).

Secondly, when deciding on a move to send to the GM, the best one available may not be the one with the highest average return. The average can hide a high goal if it is surrounded with low goals, while leading the player down paths littered with medium goals. We therefore also keep track of the maximum simulation score returned for each node in the MCTS tree. The average score is still used for action-selection during simulation playouts, but the move finally played at the root will be the one with the highest maximum score.

Finally, we add the entire simulation path leading to a better or equal solution than previously found to the game-tree, as opposed to only growing the tree one node at a time as the multi-player MCTS variant does. This guarantees that a good solution, once found, is never forgotten. The effect of this is clearly visible in Figure 3.3.

Overall MCTS/UCT does a reasonable job on simple single-agent puzzles, or where many (or partial) solutions are available. However, on more complex puzzles with large state spaces it is often clueless; there is really no good replacement for having a well-informed heuristic for guidance. *Nested Monte-Carlo Tree Search* (Cazenave, 2009; Rosin, 2011) is a more advanced MCTS approach for single player games which has been applied to GGP (Méhat & Cazenave, 2010b).

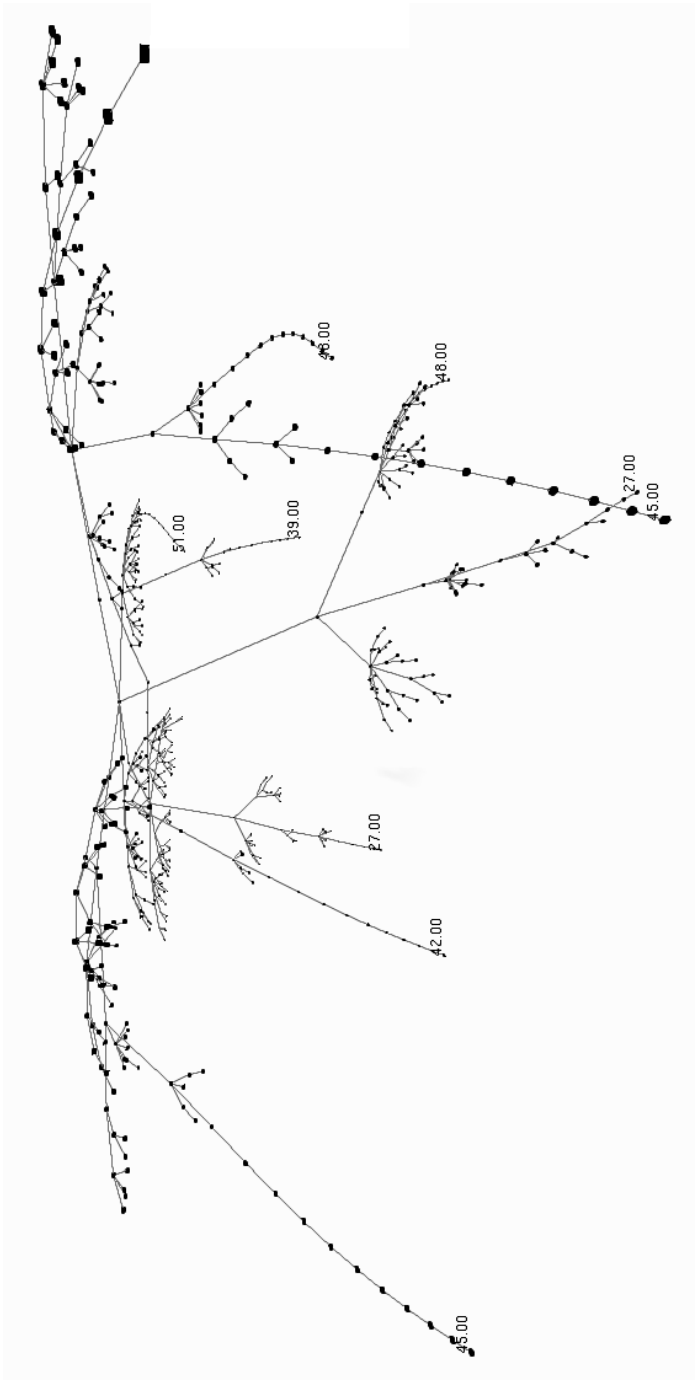

Figure 3.3: Visualization (3D) of our enhanced UCT tree for single-player games made from real data when playing Knights Tour. The numbers show goals at terminal states with better or equal score to the best one currently known at the time

3.3.6 Parallelization

One of the appeals of simulation-based searches is that they are far easier to perform in parallel than a traditional game-tree search because of fewer synchronization points. CADIAPLAYER supports two methods of running simulations in parallel. The first one is *Leaf Parallelization* where the MCTS tree is maintained by a master process and whenever crossing the MCTS border, falling out of the tree, it generates and delegates the actual playout work to a client process if available, otherwise the master does the playout. To better balance the workload a client may be asked to perform several simulations from the leaf position before returning. The second one is *Root Parallelization* where multiple instances of the MCTS agent are run completely separately and just before it is time to reply to the GM they pool all their root data to make the selection for the next move.

Parallelization for CADIAPLAYER has mostly only been used for competition purposes and is not a prominent feature of the agent. We will therefore not dwell on this subject. The above parallelization schemes are based on ideas presented in (Cazenave & Jouandeau, 2007). Further work on parallelization schemes for Monte-Carlo tree search are presented in (Chaslot, Winands, & Herik, 2008; Cazenave & Jouandeau, 2008; Chaslot, 2010).

3.4 Game-Logic Interface

The *Game-Logic Interface* encapsulates the state space of the game, provides information of available moves, how the state changes when a move is made, tells if a state is terminal and if so, what the goal reward is. It uses classes implementing an interface called Game Controller.

3.4.1 Game Controller

Game Controller is a well-defined interface which the *Game-Play Interface* connects to for the services of the *Game-Logic Interface*. It is thus easy to plug in different implementations. A detailed description of this interface is provided in Table 3.3.

The game controller implementation that CADIAPLAYER uses is called *Prolog Controller*. As the name indicates it uses Prolog as a theorem prover for the GDL logic of the game or to be more precise it uses YAP Prolog (Yet Another Prolog) (YAP, n.d.). YAP is free to use in academic environments, is a high-performance Prolog compiler and has a good C

interface. This controller takes the KIF description and converts it to a Prolog file using a home made tool (see File System box in Figure 3.1). This tool parses the KIF description, which is similar to Prolog code, so it is easy to write it back out as such. Then it calls YAP through a system command making it load the converted file plus a file containing some predefined functions (see Figure 3.1) and compile them into a YAP state. The newly compiled state is loaded into the controller using the YAP runtime library C interface. The controller uses the YAP interface to construct Prolog queries to make state transitions in the YAP state and retrieve game information from it. The controller also handles the conversion from the YAP structures into the internal structures used by the *Game-Play Interface*.

The predefined Prolog functions are always the same for all games and are used to provide easy access to extracting legal moves and play and retract them. The predefined Prolog function file is included as Appendix B.

Table 3.3: Game Controller

Function	**Description**
init	To initialize the Game Controller. Should be called before any other method.
getCurrentState	Retrieve the current state.
getMoves	Returns the list of available moves for a specific role.
make	If given moves for all roles it advances the game with those moves, but if only given a move by a single player the controller adds random moves for all other players before advancing the game.
retract	Undo the last move made by all roles.
isTerminal	Returns true if the game is over, false otherwise.
goal	Returns the value of the goal currently reached for a specific role.
gotoState	Jump to a stored or an arbitrary state that are not on the history stack. When called the history stack is unchanged except the jumped-to state has been pushed onto the top. Use with caution.
assertStateFact	Add a certain state predicate into the current state.
retractStateFact	Remove a certain state predicate into the current state.
muteRetract	Disables retracts. Can be useful when running simulations because no backtracking is needed.
syncRetract	Restore the game state that was present when *muteRetract* was called and re-enables retracts.

3.5 Summary

This Chapter described CADIAPLAYER, our GGP agent which has played a pivotal role in advancing and verifying our research. In the next chapter we move on to exploring the details of the various simulation control schemes we created and implemented into CADIAPLAYER for analysis.

Chapter 4

Learning Simulation Control

In this chapter we describe five different search-control mechanism for guiding simulation runs in MCTS. Four of them are original but the fifth one is adapted from *Go* programs. In the more recent versions of CADIAPLAYER some of these schemes are used collectively, giving an effective and robust search-control guidance on a wide range of disparate games. We also discuss the combinations we have used and which additional logic must be implemented to merge these techniques. We then finish by empirically evaluating all the simulation control techniques and selected combinations.

4.1 Simulation Control

The five different search-control mechanism for guiding simulation runs in MCTS are the following: The first one, *Move-Average Sampling Technique* (MAST), automatically learns domain knowledge for enhancing the action selection strategy in simulation playouts; the next one, *Tree-Only MAST* (TO-MAST), is identical to the first except that it is more restrictive about its learning. The third one, *Predicate-Average Sampling Technique* (PAST), is more informed about how to generalize the learned control information, that is, it affects mainly states with similar properties. The fourth method, *Features-to-Action Sampling* (FAST), uses temporal-difference learning to learn board-specific domain-knowledge for search control. The fifth method, *Rapid Action Value Estimation* (RAVE) (Gelly & Silver, 2007), is a technique originally designed to expedite search-control learning in Go programs.

4.1.1 Move-Average Sampling Technique

Move-Average Sampling Technique (MAST) (Finnsson & Björnsson, 2008) is the search-control method used by CADIAPLAYER when winning the AAAI 2008 GGP competition. It is loosely related to the *history-heuristic* (Schaeffer, 1989), which is a well-established move-ordering mechanism in *Chess*. The method learns search-control information during the back-propagation step, which it then uses in future playout steps to bias the random action selection towards choosing more promising moves. More specifically, when a return value of a simulation is backed up from T to S (see Fig. 4.1), then for each action a on the path a global (over all simulations) average for the action a, $\mathcal{Q}_h(a)$, is incrementally calculated and kept in a lookup table. Moves found to be good on average, independent of a game state, will get higher values. The rationale is that such moves are more likely to be good whenever they are available, e.g. placing a piece in the center cell in *Tic-Tac-Toe* or one of the corner cells in *Othello*. In the playout step, the action selections are biased towards selecting such moves. This is done using Gibbs sampling as below:

$$\mathcal{P}(a) = \frac{e^{\mathcal{Q}_h(a)/\tau}}{\sum_{b=1}^{n} e^{\mathcal{Q}_h(b)/\tau}}$$

where $\mathcal{P}(a)$ is the probability that action a will be chosen from the set of n available actions in the current playout state and $\mathcal{Q}_h(a)$ is the average of all values backed up in any state when action a has been selected. This results in actions with a high $\mathcal{Q}_h(a)$ value becoming more likely to be chosen. One can stretch or flatten the above distribution using the τ parameter ($\tau \rightarrow 0$ stretches the distribution, whereas higher values make it more uniform).

One of the main attractions of this scheme is its simplicity and generality, allowing useful search-control information to be efficiently computed in a game independent manner. The scheme's effectiveness has proved quite robust across a wide range of games.

4.1.2 Tree-Only MAST

Tree-Only MAST (TO-MAST) is a slight variation of MAST. Instead of updating the $\mathcal{Q}_h(a)$ for an entire simulation episode, it does so only for the part within the game tree (from state N back to S in Figure 4.1). This scheme is thus more selective about which action values to update, and because the actions in the tree are generally more informed than those in the playout part, this potentially leads to decisions based on more robust and less variance data. In short this method prefers quality of data over sample quantity for controlling the search.

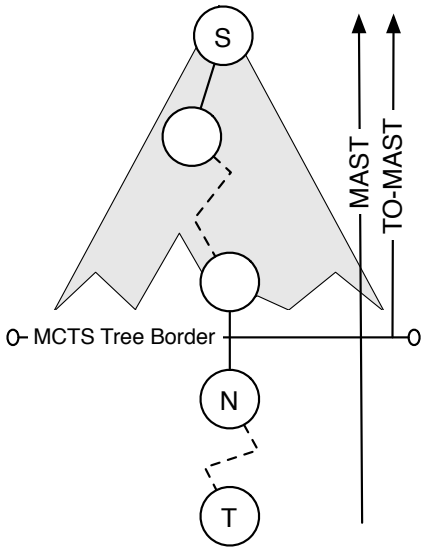

Figure 4.1: Difference between back-propagation activity of MAST and TO-MAST

4.1.3 Predicate-Average Sampling Technique

Predicate-Average Sampling Technique (PAST) has a finer granularity of its generalization than the previous schemes. As the name implies, it uses the predicates encountered in the states to discriminate how to generalize.[1]

This method works as MAST except that now average values for predicate-action pairs are maintained, $\mathcal{Q}_p(p, a)$, instead of action values $\mathcal{Q}_h(a)$. During the back-propagation, in a state s where action a was taken, $\mathcal{Q}_p(p, a)$ is updated for all $p \in P(s)$ where $P(s)$ is the set of predicates true in state s. In the playout step, an action is chosen as in MAST except that in the Gibbs sampling $\mathcal{Q}_h(a)$ is substituted with $\mathcal{Q}_p(p', a)$, where p' is the predicate in the state s with the maximum predicate-action average for a. The maximum is used for practical reasons, as it is computationally less expensive than calculating the average over all the predicate-action pairs available.

Whereas MAST concentrates on moves that are good on average, PAST can realize that a given move is good only in a given context, e.g., when there is a piece on a certain square. To ignore PAST values with unacceptably high variance, they are returned as the average game value (50) until a pre-determined minimum threshold of samples is reached.

[1] A game position, i.e. a state, is represented as a list of predicates that hold true in the state.

4.1.4 Features-to-Action Sampling Technique

The aforementioned schemes do not use any game-specific domain knowledge. Although this has the benefit of allowing effective deployment over a wide range of disparate games, this approach seems simplistic in contrast to human players, which use high-level features such as piece types and board geometry in their reasoning. The lack of understanding of such high-level game concepts does indeed severely handicap GGP players using simple search-control schemes in certain types of games, for example chess-like games where a good understanding of the different piece type values is essential for competitive play. Although GDL does not explicitly represents items such as pieces and boards such game-specific concepts can often be inferred.

With the *Features-to-Action Sampling Technique* (FAST) we use template matching to identify common board game features, currently detecting two such: piece types and cells (squares). Piece type is only judged relevant if it can take on more than one value; if not, we consider cell locations as our feature set. We use *TD*(λ) (Sutton, 1988) to learn the relative importance of the detected features, e.g. the values of the different types of pieces or the value of placing a piece on a particular cell. Each simulation, both on the start- and play-clock, generates an episode $s_1 \rightarrow s_2 \rightarrow ... \rightarrow s_n$ that the agent learns from by applying the delta rule to each state s_t in the episode:

$$\vec{\delta} = \vec{\delta} + \alpha \times [R_t^\lambda - V(s_t)] \times \bigtriangledown_{\vec{\theta}} V(s_t)$$

where R_t^λ is the λ-return (average of exponentially weighted n-step TD returns), α is the step size parameter, $V(s)$ is our value function, and $\bigtriangledown_{\vec{\theta}} V(s)$ is its gradient. A reward is given at the end of the episode, as the difference of the players' goal values. The $\vec{\delta}$ is then used in between episodes to update the weight vector $\vec{\theta}$ used by the value function to linearly weigh and combine the detected features $\vec{f}(s)$:

$$V(s) = \sum_{i=1}^{|\vec{f}|} \theta_i \times f_i(s)$$

In games with different piece types, each feature $f_i(s)$ represents the number of pieces of a given type in state s (we do not detect piece symmetry, so a white rook is considered different from a black one). In games where cell-based features (location on a board) are instead detected each feature is binary, telling whether a player has a piece in a given cell (i.e. a two-player game with N cells would result in $2N$ features). Only one set of features, pieces or locations, is used during a game and its features are referred to as the

active features. Note that for some games, no features are detected, resulting in no active features.

The value function is not used directly to evaluate states in our playouts. Although that would be possible, it would require executing not only the actions along the playout path, but also all sibling actions. This would cause a considerable slowdown as executing actions is a somewhat time consuming operation in GGP. Instead we map the value function into the same $Q_h(a)$ framework as used by MAST. This is done differently depending on type of detected features and actions. For example, for piece-type features in games where pieces move around the mapping is:

$$Q_h(a) = \begin{cases} -(2 \times \theta_{Pce(to)} + \theta_{Pce(from)}), & \text{if capture move} \\ -100 & \text{otherwise} \end{cases} \tag{4.1}$$

where $\theta_{Pce(from)}$ and $\theta_{Pce(to)}$ are the learned values of the pieces on the *from* and *to* squares, respectively. This way capture moves get added attention when available and capturing a high ranking piece with a low ranking one is preferred.

For the cell features the mapping is:

$$Q_h(a) = c \times \theta_{p,to} \tag{4.2}$$

where $\theta_{p,to}$ is the weight for the feature of player p having a piece on square to, and c is a constant. Now that we have established a way to calculate $Q_h(a)$ the $\mathcal{P}(a)$ distribution can be used to choose between actions.

We look at a concrete example in Figure 4.2 of how piece-type features are used. In *Chess* and similar board games a common encoding practice in GDL for representing a board configuration with state predicates will take on the general form (or variation thereof)

$$(< \texttt{predicate-name} >< \texttt{column} >< \texttt{row} >< \texttt{piece} >)$$

for each piece on the board. The name of the predicate may vary but a common feature is that two of its arguments indicate the cell location and the remaining one the piece type currently occupying the cell. For example, $(cell\ h\ 1\ wr)$ indicates a white rook on square `h1`. Our template does not care in what order the column and row are, but it needs the piece to be the last argument. Similarly, actions in games where pieces move around are commonly encoded using the recurring format of indicating the from and to cell locations of the moving piece, e.g. $(move\ h\ 1\ h\ 8)$. This way both different

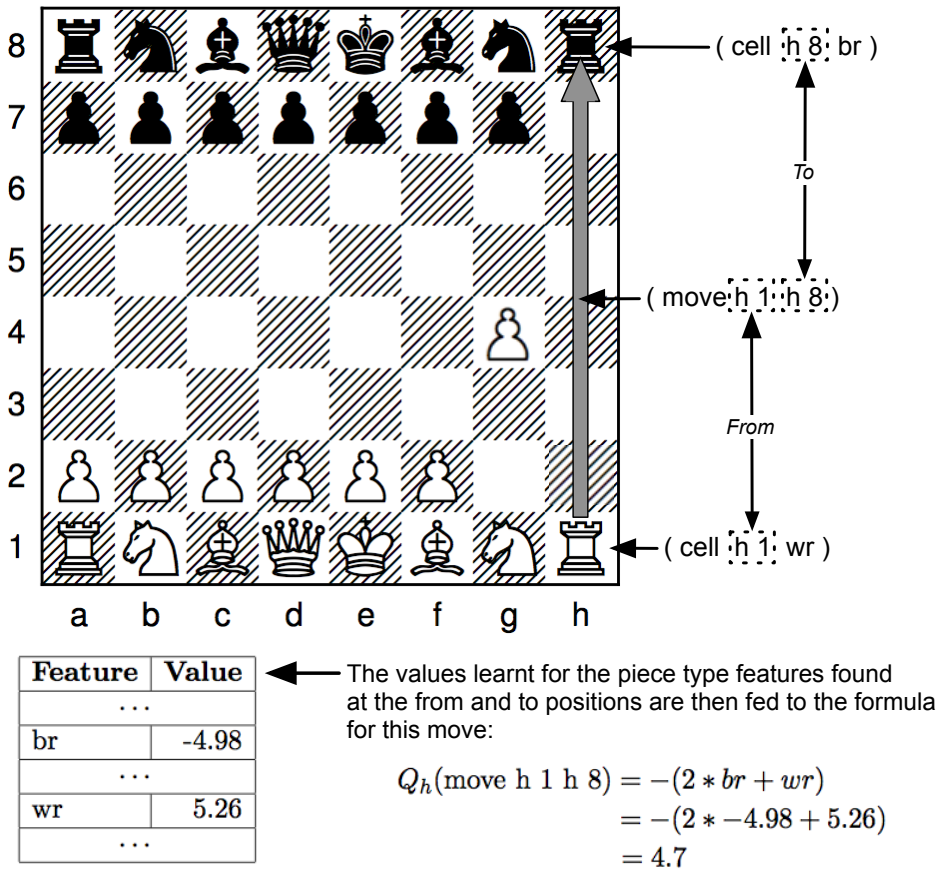

Figure 4.2: FAST capture calculations in *Chess* for (move h 1 h 8) in a state containing (cell h 1 wr) and (cell h 8 br).

piece types and piece captures can be identified. As soon as the program starts running its simulations, the TD-learning episodes provide learned values for the different piece types. For example, in *Chess* we would expect the winning side to be material up more often and our own pieces thus getting positive values and the opponent's pieces negative ones. The more powerful pieces —such as queens and rooks— have higher absolute values (e.g., in Skirmish, a simplified chess-like game used in our experiments, the values typically learned for a pawn, knight, bishop, and rook were approximately 5, 10, 10, and 13, respectively). The learned piece values θ are stored in a lookup table and consulted when captures moves are made, as depicted in Figure 4.2 for the move *rook h1 captures on h8*. The more powerful the captured opponent's piece is, especially when captured with a low ranked piece, the higher the $Q_h(a)$ becomes and thus the likelihood that the move will be played. The learned piece values are constantly updated throughout the game, however, as a precaution, they are not used unless having a value far enough from zero.

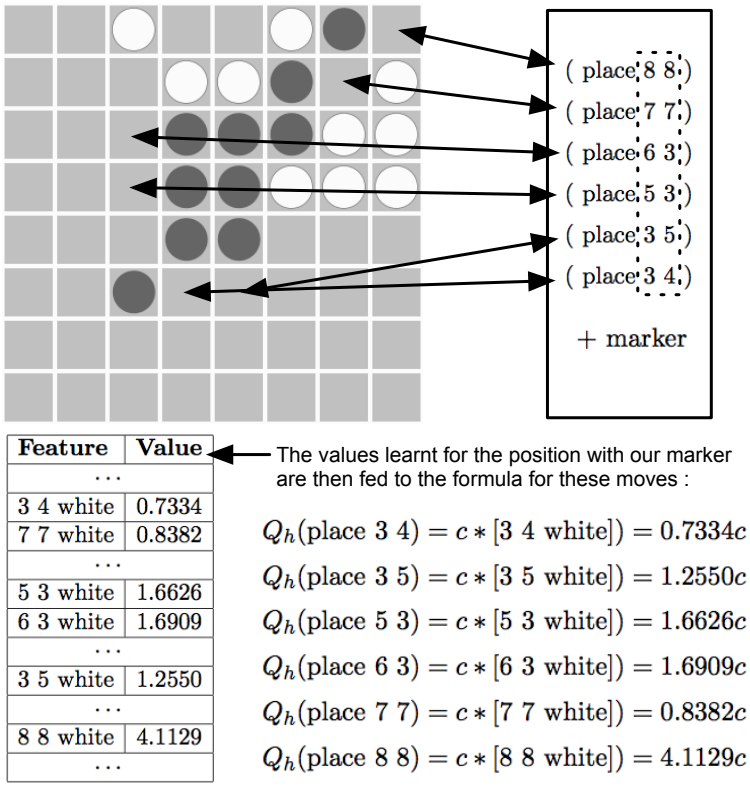

Figure 4.3: FAST calculations in *Othello* for available moves.

In games with only one type of piece for each role, the piece locations become active features instead of piece types. This is shown for the game *Othello* in Figure 4.3, where $(3\ 4\ white)$ is an example feature (i.e., a white disk on cell 3 4). During the playout step, in each state, we look up the $Q_h(a)$ values for all available actions in the feature table by their associated role and location, and bias play towards placing disks onto cells with high learned values, such as corners and edges.

4.1.5 Rapid Action Value Estimation

Rapid Action Value Estimation (RAVE) (Gelly & Silver, 2007) is a method to speed up the learning process inside the game tree. In *Go* this method is known as *All-Moves-As-First* (AMAF) heuristic because it uses returns associated with moves further down the simulation path to get more samples for duplicate moves available, but not selected, in the root state. When this method is applied to a tree structure as in MCTS the same is done for all levels of the tree. When backing up the value of a simulation, we update in the tree not only the value for the action taken, $\mathcal{Q}(s,a)$, but also sibling action values, $\mathcal{Q}_{RAVE}(s,a')$, if and only if action a' occurs further down the path being backed up (S to T in Figure 4.1).

As this presents bias into the average values, which is mainly good initially when the sampled data is still unreliable, these rapidly learned estimates should only be used for high variance state-action values. With more simulations the state-action averages $\mathcal{Q}(s,a)$ become more reliable, and should be trusted more than the RAVE value $\mathcal{Q}_{RAVE}(s,a)$. To accomplish this the method stores the RAVE value separately from the actual state-action values, and then weights them linearly as:

$$\beta(s) \times \mathcal{Q}_{RAVE}(s,a) + (1 - \beta(s)) \times \mathcal{Q}_{UCT}(s,a)$$

where $\mathcal{Q}_{UCT}(s,a)$ is the UCT value of the state-action pair (see Section 2.2.1) and

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

The parameter k is called the *equivalence parameter* and controls how many state visits are needed for both estimates to be weighted equal. The function $n(s)$ tells how many times state s has been visited.

4.2 Combining Schemes

In CADIAPLAYER you always use the same player throughout a single game, so in order to apply different techniques at the same time, they must be combined into a single player implementation.

The RAVE scheme can be easily combined with all the other simulation control schemes as it operates in the selection step of a simulation, as opposed to in the playout step as the other four. Due to this straight forwardness in implementation a RAVE/MAST combination was built and used in the 2009 version of CADIAPLAYER.

The later devised FAST scheme was not as easily integrated into the already existing combined RAVE/MAST scheme. The problem is that MAST and FAST both operate on the playout step and possibly bias the action selection differently. We opted for the following MAST/FAST integration in the playout step:

$$Q_h(a) = \begin{cases} Q_{\text{MAST}}(a) + w \times Q_{\text{FAST}}(a) & \text{if any features active in } A(s) \\ Q_{\text{MAST}}(a) & \text{otherwise} \end{cases} \tag{4.3}$$

where $Q_{\text{MAST}}(a)$ and $Q_{\text{FAST}}(a)$ are the $Q_h(a)$ values as calculated by the MAST and FAST schemes respectively, and the w parameter is a weighing constant deciding on their relative importance. If no features are active in the current state, meaning that either no features were detected in the game description or that no capture moves are available in such a game, the $Q_{\text{FAST}}(a)$ distribution becomes uniform and is omitted (as it would shift the final distribution without adding any information).

For maximum efficiency the influence of each scheme must be carefully weighted, possibly varying depending on the game type. A RAVE/MAST/FAST combination scheme was used in the 2010 competition, however, we simply fixed $w = 1.0$ because of a lack of time for thoroughly tuning a more appropriate value.

Since MAST, TO-MAST and PAST are closely related and really just a different take on a similar logic they are not viable to be mixed. They could, however, be selected between if some criteria for such selection was invented. The above combination method can be used to swap in both TO-MAST and PAST instead of MAST.

4.3 Empirical Evaluation

The aforementioned search-control schemes were developed and added to CADIAPLAYER at different time points, often in a response to an observed inefficiency of the agent in particular types of games. The driving force behind combining these schemes has in part been the GGP competition and therefore only a few number of combinations have been used. Because of this we will evaluate all simulation-control schemes individually, but only the subset of possible combinations that have been used for the GGP competitions. The main difference in the agent between subsequent GGP competitions has thus been the sophistication level of the search control. Although there have been other more minor improvements made to the agent from year to year, e.g., in the form of a more effective transposition table, implementation efficiency improvements, and better tuned parameter settings, in here we are interested in quantifying the effects of the search-control schemes. We nullify the effect of other changes by having all agent versions share the newest and most effective code base, thus differing only in the type of search-control scheme used.

4.3.1 Setup

We empirically evaluate eight versions of CADIAPLAYER, one is a basic MCTS version, five representing a certain type of search-control and the remaining two are RAVE/MAST and RAVE/MAST/FAST combinations. The MCTS and MAST versions are used as baseline players that all other versions are matched against. The MCTS baseline agent chooses actions uniformly at random in the playout phase. We use MAST as a baseline of a player with an augmented playout strategy as it is the most general and was the search-control scheme used when CADIAPLAYER won the 2008 competition. In the tables that follow, each data point represents the result of a 300-game match, showing both a win percentage and a 95% confidence interval. The matches were conducted on CentOS Linux running on Intel(R) Xeon(R) 3.00GHz and 3.20GHz CPU computers. Each agent used a single processor and competing agents always used the same speed processors.

The value of the UCT parameter C is set to 40 (for perspective, possible game outcomes are in the range 0-100). The τ parameter of the $\mathcal{P}(a)$ distribution is set to 10 for all agents, with the exception of PAST where we have it set to 8. The sample threshold for a PAST value to be used was set to 3. In FAST the λ parameter is set to 0.99, the step-size parameter α to 0.01, c to 5, and w to 1. The *equivalence parameter* for RAVE is set to 500. These parameters are the best known settings for each scheme, based on trial and error testing. The startclock and the playclock were both set to 10 seconds.

Our experiments used:

- Eight two-player turn-taking games: *3D Tic-Tac-Toe, Breakthrough, Checkers, Connect 5, Knightthrough, Othello, Skirmish and TCCC4*.
- Two two-player simultaneous-move games: *Battle and Chinook*.
- Two three-player turn-taking games: *3-player Chinese Checkers and 3-player TCCC4*.

Full game descriptions can be found on the Dresden GGP server (Dresden GGP, n.d.) and descriptions of the games used in these experiments are found in Appendix C.

4.3.2 Individual Schemes Result

Tables 4.1 and 4.2 show the result of the matches of all individual schemes. In addition to an MCTS baseline player (Table 4.1), we also compare to the MAST scheme (Table 4.2).

Table 4.1: Tournament: Simulation Control Agents versus MCTS

Game	**Roles**	**Move**	**MAST** win %	**TO-MAST** win %	**PAST** win %	**FAST** win %	**RAVE** win %
3D Tic-Tac-Toe	2	Turn	62.33 (± 5.49)	55.33 (± 5.64)	87.00 (± 3.81)	54.00 (± 5.65)	77.67 (± 4.72)
Battle	2	Sim.	58.50 (± 5.28)	59.33 (± 5.31)	63.83 (± 5.19)	58.00 (± 5.32)	89.67 (± 3.36)
Breakthrough	2	Turn	88.33 (± 3.64)	86.33 (± 3.89)	87.00 (± 3.81)	82.33 (± 4.32)	63.33 (± 5.46)
Checkers	2	Turn	56.50 (± 5.38)	81.83 (± 4.13)	52.17 (± 5.44)	49.00 (± 5.40)	80.50 (± 4.31)
Chinese Checkers 3P	3	Turn	64.33 (± 3.89)	53.33 (± 5.65)	70.67 (± 5.16)	47.00 (± 5.66)	50.33 (± 5.67)
Chinook	2	Sim.	69.67 (± 5.11)	71.00 (± 5.10)	71.00 (± 5.06)	64.17 (± 5.35)	68.83 (± 5.12)
Connect 5	2	Turn	64.67 (± 5.42)	58.00 (± 5.59)	83.00 (± 4.21)	72.33 (± 5.07)	92.00 (± 3.08)
Knightthrough	2	Turn	94.00 (± 2.69)	92.00 (± 3.08)	95.33 (± 2.39)	84.67 (± 4.08)	69.00 (± 5.24)
Othello	2	Turn	63.67 (± 5.33)	51.83 (± 5.58)	64.00 (± 5.34)	68.67 (± 5.15)	69.17 (± 5.16)
Skirmish	2	Turn	43.33 (± 5.24)	48.50 (± 5.30)	38.00 (± 5.16)	92.00 (± 2.82)	51.50 (± 5.24)
TCCC4	2	Turn	77.33 (± 4.63)	43.50 (± 5.57)	80.33 (± 4.31)	46.50 (± 5.55)	51.17 (± 5.50)
TCCC4 3P	3	Turn	63.17 (± 5.42)	48.00 (± 5.64)	61.50 (± 5.47)	36.33 (± 5.41)	49.33 (± 5.63)
Overall			**67.15**	**62.42**	**71.15**	**62.92**	**67.71**

Table 4.2: Tournament: Simulation Control Agents versus MAST

Game	**Roles**	**Move**	**TO-MAST** win %	**PAST** win %	**FAST** win %	**RAVE** win %
3D Tic-Tac-Toe	2	Turn	53.67 (± 5.65)	77.33 (± 4.75)	30.00 (± 5.19)	69.67 (± 5.21)
Battle	2	Sim.	54.17 (± 5.37)	54.17 (± 5.39)	54.33 (± 5.33)	87.83 (± 3.60)
Breakthrough	2	Turn	53.33 (± 5.65)	52.00 (± 5.66)	41.33 (± 5.58)	13.00 (± 3.81)
Checkers	2	Turn	76.50 (± 4.61)	53.67 (± 5.32)	45.50 (± 5.44)	75.67 (± 4.59)
Chinese Checkers 3P	3	Turn	37.00 (± 5.47)	47.00 (± 5.66)	34.33 (± 3.93)	40.00 (± 5.55)
Chinook	2	Sim.	42.33 (± 5.54)	48.50 (± 5.54)	40.83 (± 5.48)	45.17 (± 5.55)
Connect 5	2	Turn	44.67 (± 5.62)	68.83 (± 5.09)	48.67 (± 5.67)	85.67 (± 3.97)
Knightthrough	2	Turn	50.00 (± 5.67)	44.33 (± 5.63)	25.33 (± 4.93)	11.67 (± 3.64)
Othello	2	Turn	34.67 (± 5.29)	56.67 (± 5.50)	57.83 (± 5.49)	57.50 (± 5.54)
Skirmish	2	Turn	56.83 (± 5.29)	45.00 (± 5.20)	93.83 (± 2.59)	57.83 (± 5.29)
TCCC4	2	Turn	25.00 (± 4.75)	61.00 (± 5.27)	21.83 (± 4.58)	28.83 (± 4.91)
TCCC4 3P	3	Turn	40.00 (± 5.49)	53.83 (± 5.62)	30.60 (± 3.79)	42.50 (± 5.56)
Overall			**47.35**	**55.19**	**43.70**	**51.28**

All five schemes show a significant improvement over the MCTS base player overall where PAST does the best with an over 70% average improvement on this set of games. MAST and RAVE tie for second best ($67\% - 68\%$) but the games they individually excel in do not have any noticeable overlap. TO-MAST and FAST also do similar overall ($62\% - 63\%$). TO-MAST sets itself apart from its related schemes with its great performance boost in *Checkers* while the most noticeable result of FAST is its excellent performance in *Skirmish*. We can see from Table 4.1 that the MAST and PAST schemes offer a genuine improvement over the MCTS player in almost all the games, particularly in *Breakthrough* and related games along with TO-MAST. This is not surprising as the schemes were originally created to improve the agent's performance in such type of games. The only game where the MAST and PAST schemes have non-positive effects on performance is in *Skirmish*. Even though PAST performs somewhat better than MAST we have preferred to use MAST in our competition player as it is in some ways more robust. For example PAST might suffer from scaling problems in games with unnaturally large state representation.

The addition of RAVE does not cause MCTS to suffer any adverse effects with this set of games and adds a lot to the performance in *3D Tic-Tac-Toe*, *Battle*, *Checkers*, and *Connect 5*, be it against MCTS or MAST.

Regarding Table 4.2, there are several points of interest. First of all RAVE offers improvement over MAST in half of the games (and vice-versa) indicating that a lot can be gained by combining the two such that they compensate for each other weaknesses. PAST still does better than MAST when going head to head against it with 55% average win. It only does statistically significantly worse in *Knightthrough*.

Also of interest is to contrast TO-MAST's performance on different games. The only difference between MAST and TO-MAST is that the former updates action values in the entire episode, whereas the latter only updates action values when back-propagating values in the top part of the episode, that is, when in the tree. TO-MAST significantly improves upon MAST in the game of *Checkers*, whereas it has a detrimental effect in the game of *Othello*. A possible explanation is that actions generalize better between states in different game phases in *Othello* than in *Checkers*, that is, an action judged good towards the end of the game is more often also good early on if available. For example, placing a piece on the edge of the board is typically always good and such actions, although not available early on in the game, start to accumulate credit right away. FAST might offer some improvements in *Othello*, and again so does RAVE, but FAST outperforms all others in *Skirmish*. The problem that MCTS-based players have with chess-like games such as *Skirmish* is that they do not realize fast enough that the value of the pieces are radically

Table 4.3: Tournament: Combined Agents versus MCTS

Game	Roles	Move	RAVE/MAST win %	RAVE/MAST/FAST win %
3D Tic-Tac-Toe	2	Turn	87.33 (± 3.77)	85.33 (± 4.01)
Battle	2	Sim.	94.67 (± 2.37)	26.50 (± 4.70)
Breakthrough	2	Turn	89.33 (± 3.50)	78.67 (± 4.64)
Checkers	2	Turn	79.67 (± 4.37)	69.00 (± 5.01)
Chinese Checkers 3P	3	Turn	68.33 (± 5.27)	63.33 (± 5.46)
Chinook	2	Sim.	74.33 (± 4.89)	56.33 (± 5.55)
Connect 5	2	Turn	95.83 (± 2.24)	84.00 (± 4.16)
Knightthrough	2	Turn	93.00 (± 2.89)	91.00 (± 3.24)
Othello	2	Turn	78.17 (± 4.63)	72.17 (± 4.96)
Skirmish	2	Turn	51.67 (± 5.39)	89.50 (± 3.17)
TCCC4	2	Turn	75.33 (± 4.66)	49.33 (± 5.59)
TCCC4 3P	3	Turn	60.00 (± 5.49)	44.33 (± 5.59)
Overall			**78.97**	**67.46**

different, e.g. that a rook is worth more than a pawn. Even though MCTS might towards the end of the game start to realize this then it is far to late to save the game. The reason for FAST's performance is that it is not only able to detect the capture moves in *Skimish*, it can also evaluate the loss or gain in material. FAST is therefore very helpful in focusing the simulations towards the objective of the game which is to capture all the opponent's pieces.

4.3.3 Combined Schemes Result

In CADIAPLAYER 2009 the combined RAVE/MAST scheme was introduced. The new version is though still inferior to the baseline player in the game of *Skimish* so we specifically targeted this inefficiency in the 2010 version of the agent by incorporating FAST. The 2011 competition version of CADIAPLAYER was also a RAVE/MAST/FAST combination, but this time it was parallelized over 40 cores using *Root Parallelization*. To measure how CADIAPLAYER has improved through the years as its search control mechanism has been improved we also pitch these two versions against the MCTS and MAST baseline players which are actually the 2007 and 2008 versions of CADIAPLAYER respectively. The results of these tournaments are presented in Tables 4.3 and 4.4

From the tournament data we see that the new combined RAVE/MAST/FAST scheme is very effective in *Skirmish* against the baseline players as expected. Still the new combina-

Table 4.4: Tournament: Combined Agents versus MAST

Game	Roles	Move	RAVE/MAST win %	RAVE/MAST/FAST win %
3D Tic-Tac-Toe	2	Turn	81.67 (± 4.39)	75.00 (± 4.91)
Battle	2	Sim.	86.33 (± 3.75)	25.50 (± 4.64)
Breakthrough	2	Turn	53.00 (± 5.66)	40.67 (± 5.57)
Checkers	2	Turn	79.50 (± 4.27)	65.00 (± 5.23)
Chinese Checkers 3P	3	Turn	49.67 (± 5.67)	53.33 (± 5.65)
Chinook	2	Sim.	55.50 (± 5.55)	35.33 (± 5.32)
Connect 5	2	Turn	91.67 (± 3.13)	68.67 (± 5.22)
Knightthrough	2	Turn	55.00 (± 5.64)	26.67 (± 5.01)
Othello	2	Turn	63.33 (± 5.34)	57.67 (± 5.52)
Skirmish	2	Turn	54.50 (± 5.22)	92.50 (± 2.74)
TCCC4	2	Turn	53.50 (± 5.45)	27.50 (± 4.90)
TCCC4 3P	3	Turn	46.50 (± 5.63)	31.17 (± 5.20)
Overall			**64.18**	**49.92**

tion does incur reduced performance in multiple games with this set. First of all FAST did not activate in *3D Tic-Tac-Toe* and *Chinese Checkers 3P* so the small dip in performance from RAVE/MAST is explained by the overhead of coming to that conclusion.

Battle, *Checkers*, *Skirmish*, *TCCC4*, and *TCCC4 3P* triggered the piece-type feature of FAST. Even though triggered in *Checkers*, it was rather ineffective as the captures there are achieved by jumping over pieces. *Battle* seems to confuse FAST as playing a capture move does not guarantee a capture. Since the game uses simultaneous moves, the opponent can move out of the way or choose to defend a piece so if attacked the attacker is captured. The problem with *TCCC4* and *TCCC4 3P* is that the number of pieces vary only for disks as the other pieces re-spawn on capture, and the agent thus learned a significant value for that piece only, making our agent too aggressive in adding such pieces to the board.

Breakthrough, *Chinook*, *Connect5*, *Knightthrough*, and *Othello* all triggered the location feature of FAST. For *Breakthrough*, *Chinook*, and *Knightthrough* piece formations are more important than placing pieces on particular locations, and the FAST scheme thus adds noise which may reduce the effectiveness of MAST. In *Othello* and especially *Connect5* FAST and MAST appear have competing strategies resulting in loss of focus on both sides.

From these experiments, it is clear that a deeper analysis is needed into which game properties must be present to apply the FAST scheme in the most effective way. Using FAST

in the competition build of CADIAPLAYER is still important as it offers great performance boost in chess-like games, e.g. *Skirmish*, which none of the other schemes handle effectively.

4.4 Summary

Informed search-control for guiding simulations playouts is an essential core component of CADIAPLAYER. In here we described the various search-control schemes we have created and/or tried and empirically evaluated the performance gains achieved over the years as it evolved.

Generic search-control schemes using only implicit domain knowledge, such as MAST and RAVE, are able to provide impressive performance across a large collection of games. However, such a generic approach fails to take advantage of higher-level game concepts that are important for skillful play in many game domains. By using search-control methods that incorporate such concepts, like FAST, we were able to improve the performance of CADIAPLAYER in chess-like games, which had previously proved problematic for our agent. Challenges in using such game-type specific approaches include knowing when to apply them and how to balance them with existing search-control approaches. We made some initial steps in that direction with the RAVE/MAST/FAST combination of the CADIAPLAYER agent.

Chapter 5

Generalizing MCTS Extensions

GGP agents must learn in real time. This means that various MCTS search enhancements that rely on a priori knowledge, either handcrafted or learned offline, are not applicable as is in GGP. The contributions of this chapter are extensions to two existing techniques that generalize their knowledge acquisition such that no domain specific knowledge is required a priori; real time information is gathered from the MCTS simulations rather than making assumptions from the game rules. These extensions are *Early Cutoffs* and *Unexplored Action Urgency*. They deal with terminating simulations early if they are good enough or useless and keeping the focus on good actions in the presence of completely unexplored actions, respectively.

5.1 MCTS Extensions for GGP

In GGP it is a difficult task to generate reliable knowledge which can even backfire, causing undesirable behavior in the agent. Extending MCTS with knowledge can produce good results but the improvement gain relies heavily on the quality of the knowledge used. Any attempt of injecting such extensions into an MCTS GGP agent must take into consideration at least two possible scenarios. The extension may be harmful in some games if the knowledge it utilizes is not general enough and in some cases it may not be applicable at all and should be ignored.

Following are generalizations of two extensions for MCTS. They make use of statistical information gathered in real time from the simulations rather than making static assumptions from the game rules.

5.1.1 Early Cutoffs

Without knowledge to guide the MCTS simulations huge amounts of computation time may be wasted on irrelevant continuations in the playout phase. There is a danger of simulations running for a long time, missing a lot of obvious good moves or even just going around in circles in the state space. When they finally end up in a terminal state the information backed up in the tree may be just noise. Since GGP has no prior knowledge to help avoid useless paths, it can waste a large portion of the node expansions traversing such paths. Also, to ensure that GGP games are finite a counter artificially terminated too long games is often encoded into the game rules. This may lead to uninformative scores as such games are typically as a draw without any regards to which player is closer to winning. If it would be possible to predict that a simulation is headed nowhere it would be better to terminate it immediately and use the time saved to run another simulation that has a better chance of generating more relevant information.

In other cases simulations may have entered a part of the state space where one player is in such a good position that it is not necessary to continue and better to score the game immediately so another simulation can run. It may even be enough to know when in the playout phase the simulation discovers a better or worse position for the agent and return back this information.

This extension covers two cases when simulations can be terminated early. On one hand is the case when a simulation can be evaluated to an informative score before reaching a terminal state and on the other hand is the case when the simulation has gone on longer than any rational play of the game would without reaching a terminal state.

Before cutoffs can be made we first gather data that helps us decide if the extension is applicable to the game being played. This is done by observing the "normal" simulations of the agent to get information on whether the extension should be activated and what settings to use. This data includes the maximum and minimum depths of the terminal states and therefore needs naturally terminated simulations.

To see if it is possible to evaluate the states we use the scoring mechanism of GDL, the *Goal* relation. The trouble here is that it makes no promises of returning useful information, if any at all, in non-terminal states. Still in some games it can give a good estimation on the state as a result of how the game rules are written, To make sure that the *Goal* relation can be used we check it for a property called *Stability*. The notion of stability in GGP was invented for the agent CLUNEPLAYER (Clune, 2008) and was used to determine if a feature extracted from the game description would be useful in evaluating game positions. Stability is based on the assumption that if a value calculated from a state changes gradu-

Algorithm 5 Pseudo-code for deciding cuts for the Early Cutoff extension

```
if not useEarlyCutoff then
  return false
end if
if playoutSteps < minimumSteps then
  return false
end if
if IsGoalStable() then
  // Cutoff point has been calculated as:
  // cut ← firstGoalChange + numPlayers
  return playoutSteps ≥ cut
end if
if hasTerminalInterval() then
  // Cutoff point has been calculated as:
  // cut ← firstTerminal + 0.33 ∗ terminalInterval
  return playoutSteps ≥ cut
end if
```

ally throughout the course of the game it is in some way correlated with what is actually happening in the game as it plays out. If this value jumps back and forth continuously or never changes it is deemed unrelated to the game progression. To calculate the stability value the variance of the changes in the score obtained from the *Goal* relation is observed throughout the simulations and then averaged over them. If the resulting stability value is lower than a predetermined stability threshold but not zero this score is believed to be stable and the extension is activated.

Algorithm 5 outlines the decision-making process for whether an early cutoff should be made; we use it as a reference throughout this section. It is called right after checking if the current state is terminal and before any actions retrieval or selection is made to transition to the next state. The variable *useEarlyCutoff* is true if the extension is active and *playoutSteps* always contains the current number of states traversed since the current simulation left the MCTS tree.

The *Goal* stability part of the extension has precedence and once it has been activated it terminates simulations and scores them with the *Goal* relation as shown with the conditional function *IsGoalStable* in Algorithm 5. The cutoff point is the distance from the initial state of the game to the first state where the goal changes with the number of players participating added. It is then checked relative to how many playout steps have been taken. This way the simulations should immediately start to draw attention towards the parts of the state space where something is to be gained. The player count addition is made to allow at least one round to be played beyond the goal point if there is an immediate neutralizing response, not to mention if it is a forced one. These variables are only

set during the initial simulations at the same time the stability is determined and therefore keep the same distance from the MCTS tree fringe throughout the game.

In order to predict if a simulation is a waste of time due to its length the agent collects data on at what depth terminal states are encountered and uses it to estimate the depth interval in which they occur. If the interval is very narrow or always at the same depth, there is no benefit in making early cuts as little or no time can be saved by doing so. The minimum interval size is set equal to the variable *minimumSteps* seen in Algorithm 5 where it is also used as a failsafe for a minimum number of playout steps that must be made for a cut to be considered. If it is found that the interval is sufficiently big the extension becomes active through the conditional function *hasTerminalInterval()* in Algorithm 5.

As before a cutoff depth is set and then checked relative to the MCTS tree fringe by keeping track of the number of steps taken in the playout phase. The cutoff depth is set relative to the initial state and matches traversing the third of the interval where terminal states have been observed. As an example let us say we have a game where terminal states have been reached at depths 20 to 50 relative to the initial state. The cutoff depth is then:

$$20 + \frac{1}{3} * (50 - 20) = 20 + 10 = 30$$

If we then exit the MCTS tree at depth 8 from the initial state the simulation will be terminated at depth 38. A terminal interval cutoff defaults to a draw (GGP score of 50).

The reason for one third being the fraction of the terminal state interval traversed as a minimum for a cut to be made was based on results from Chapter 7. They have shown that pushing back an artificial termination depth to give a UCT agent more chance to find a real terminal state does not result in better play through better information. In the example game there the agent reached its peak performance as early as only finding a real terminal state 30% of the time. From those results, traversing the top one third of the terminal state interval as an absolute minimum is a reasonable setting for this parameter.

5.1.2 Unexplored Action Urgency

In GGP it is commonplace to use UCT in the selection phase to control the exploration/exploitation balance in MCTS. By default UCT never exploits on the fringe of the MCTS tree, always selecting amongst the unexplored actions available. It is not until all unexplored actions have been exhausted that attention is given to exploitation in the state. When domain-specific knowledge is available unexplored actions can be evaluated and bad ones ignored. This may sound unreasonable but if the knowledge is accurate, the

Algorithm 6 Pseudo-code for using the Unexplored Action Urgency extension

if $state.explored = \emptyset$ **then**
  // We are in the Playout phase
  return $playoutStrategy(state.unexplored)$
end if
$action \leftarrow selectionStrategy(state.explored)$
if $state.unexplored = \emptyset$ **then**
  // Fringe not reached in the Selection phase
  return $action$
end if
// Fringe reached in the Selection phase
$exploit \leftarrow action.uctValue$
$discount \leftarrow state.unexplored.size()/state.actions.size()$
// Calculate the urgency of the unexplored actions
$urgency \leftarrow 50 + C_p * \sqrt{\ln state.visits()} * discount$
// Check if it is more urgent to exploit or explore
if $exploit \geq urgency$ **then**
  return $action$
else
  return $playoutStrategy(state.unexplored)$
end if

agent would never have selected the bad action anyway. Although in GGP we may not have domain-specific knowledge at our disposal, on the fringe of the MCTS tree we have some knowledge about the actions available, the explored actions, which motivates us to use this knowledge to do something smarter than the default handling.

The main idea with the extension is to make it possible for the agent to exploit actions on the MCTS tree fringe if there is reason to do so. Even though no domain specific knowledge is available we can observe how the rewards of an action accumulate. Basically if the reward is consistently very good the action estimated average will stay high no matter how often we select. So if an action fits this description we are going to prefer it over any unexplored action found in the state while the number of simulations traversing this state is low. Instead of the default rule with unexplored actions taking precedence a quick decision point is added when the agent is on the MCTS fringe. The decision of selecting an unexplored action is given a value that can be compared with the highest UCT value of the already explored actions and the higher value wins. Basically the agent is selecting between using the selection strategy on the subset of explored actions in the state or the playout strategy on the subset of the unexplored actions. We call the value assigned to the unexplored actions their *urgency*, which is a phrase used when knowledge is used to assign values to actions in (Bouzy, 2005). This is similar to the *First Play Urgency* (FPU) in MoGo (Gelly et al., 2006).

Pseudo-code for the urgency calculations is shown in Algorithm 6. The variable $state$ accesses the current simulation state and from it we get three action sets, $unexplored$ actions, $explored$ actions, and all $actions$. The $exploit$ variable stores the highest UCT value of the already explored actions in the state and the $urgency$ variable holds the urgency of the unexplored actions. The formula for the urgency is actually the UCT formula as it would look for an action that has been tried once before in this state and resulted in an immediate draw (GGP scores range from 0 to 100). The $discount$ variable has the purpose of lowering the urgency as more and more actions get explored. We have the discount calculated as the ratio between the number of unexplored actions left and the total number of actions, because with fewer and fewer actions left unexplored, the more likely we are to have already come across the best available action. By incorporating the UCT formula into this decision we keep the guarantee of all actions getting explored in the limit.

This extension relies on the agent actually selecting a good action early on when exhausting the unexplored actions in a state on the MCTS fringe. It is likely that given a uniform random playout strategy this extension will simply get balanced out and no improvements made. This extension must be paired with a playout strategy that shifts the action selection in favor of selecting good actions early. Given such a playout strategy this extension should boost its effect while not causing harm if it turns out only as good as the uniform random strategy in some games. An example of such an extension is the *Move-Average Sampling Technique* (MAST) (Finnsson & Björnsson, 2008) (see Section 4.1.1) which we will use in order to evaluate this extension in the experiments section of this chapter.

5.2 Empirical Evaluation

Both extensions were implemented into CADIAPLAYER. The game set is the same as in Chapter 4. Descriptions of them are in Appendix C.

5.2.1 Setup

The agents ran on a single Intel Xeon 2.66GHz, 3.00GHz or 3.20GHZ CPU (competing agents always on matching CPUs) and every game type was run as a tournament between two agents with 300 game instances played, having the agents switch roles half way through. In every game the agents were given 10 seconds both for the startclock and the playclock. The number of sample simulations for the *Early Cutoff* extensions was

set as 100 and its stability threshold as 1000 (in this setup the measured stability value of Checkers is around 250 and Othello around 3000). The results are displayed in Tables 5.1 and 5.2. The cells marked as *-N/A-* under the *Early Cutoff* extension are games where the extension decided that it should not activate. In every tournament an extended agent was pitched against the base agent it was derived from. All cell data is the winning percentage of the extended agent along with a 95% confidence interval.

The base agents used were MCTS and MAST CADIAPLAYER agents. MAST has the ability to bias the playout action selection in many games towards good actions, allowing the hypothesized consequences of Unexplored Action Urgency extension to be checked. In Table 5.2 the base agents are extended with both extensions to reveal the overall gain and test for any improvement overlap. The third base player in that table, the RAVE/MAST/FAST agent is the extension combination used by the GGP agent CADIAPLAYER (Finnsson & Björnsson, 2008) in the GGP competition in 2011. The failsafe variable for minimum steps in the Early Cutoff extension was set to 10.

5.2.2 Results

The results for the *Early Cutoff* extension (Table 5.1) show significant improvement in the games *3D Tic-Tac-Toe*, *Battle*, *Checkers*, *Skirmish* and *TCCC* both with and without the MAST extension plus in *Connect 5* when using only vanilla MCTS. Also it is important to point out that in other games where this extension is active it shows little or no sign of having adverse effects on the agent. Regarding the games that did not activate the extension, let us take a closer look at *Othello* as an example. It has only extremely rare instances of the game ending before the board has been filled, making the terminal interval just a single value when sampled. The player who has more disks wins and this is the information the *Goal* relation gives at every state of the game. This information goes back and forth in declaring which player is winning and therefore gets evaluated as unstable by the extension. This matches the generally known fact that the difference in number of disks is a bad heuristic for *Othello*.

Table 5.1: Isolated Extension Tournaments

Game	Roles	Move	**Early Cutoff + UCT** vs. **UCT** win %	**Early Cutoff + MAST** vs. **MAST** win %	**Unexplored Action Urgency + UCT** vs. **UCT** win %	**Unexplored Action Urgency + MAST** vs. **MAST** win %
3D Tic-Tac-Toe	2	Turn	62.67 (± 5.48)	57.67 (± 5.60)	48.00 (± 5.66)	57.67 (± 5.60)
Battle	2	Sim.	87.50 (± 3.62)	73.33 (± 4.97)	46.33 (± 5.52)	54.00 (± 5.50)
Breakthrough	2	Turn	53.00 (± 5.66)	51.67 (± 5.66)	44.33 (± 5.63)	70.67 (± 5.16)
Checkers	2	Turn	72.83 (± 4.79)	74.33 (± 4.71)	48.33 (± 5.41)	52.00 (± 5.33)
Chinese Checkers 3P	3	Turn	48.33 (± 5.66)	55.33 (± 5.64)	42.33 (± 5.60)	47.33 (± 5.66)
Chinook	2	Sim.	46.17 (± 5.49)	52.17 (± 5.53)	50.17 (± 5.56)	63.67 (± 5.33)
Connect 5	2	Turn	61.33 (± 5.52)	44.50 (± 5.53)	51.00 (± 5.67)	58.00 (± 5.46)
Knightthrough	2	Turn	54.67 (± 5.64)	47.67 (± 5.66)	51.33 (± 5.67)	61.00 (± 5.53)
Othello	2	Turn	*-N/A-*	*-N/A-*	47.83 (± 5.54)	58.50 (± 5.46)
Skirmish	2	Turn	63.67 (± 5.13)	67.67 (± 4.97)	41.50 (± 5.26)	45.17 (± 5.38)
TCCC4	2	Turn	66.67 (± 4.99)	63.33 (± 5.18)	37.33 (± 5.32)	53.33 (± 5.32)
TCCC4 3P	3	Turn	49.50 (± 5.62)	43.67 (± 5.56)	51.33 (± 5.65)	49.83 (± 5.62)
Overall			**60.58**	**57.39**	**46.65**	**55.93**

Table 5.2: Combined Extensions Tournaments

Game	Roles	Move	**Early Cutoff + Unexplored Action Urgency + UCT** vs. **UCT** win %	**Early Cutoff + Unexplored Action Urgency + MAST** vs. **MAST** win %	**Early Cutoff + Unexplored Action Urgency + RAVE/MAST/FAST** vs. **RAVE/MAST/FAST** win %
3D Tic-Tac-Toe	2	Turn	64.67 (± 5.42)	61.33 (± 5.52)	56.67 (± 5.62)
Battle	2	Sim.	74.67 (± 4.80)	74.67 (± 4.82)	84.33 (± 3.99)
Breakthrough	2	Turn	52.33 (± 5.66)	68.33 (± 5.27)	73.33 (± 5.01)
Checkers	2	Turn	71.00 (± 4.87)	71.33 (± 4.82)	65.83 (± 5.12)
Chinese Checkers 3P	3	Turn	50.67 (± 5.67)	53.67 (± 5.65)	62.00 (± 5.50)
Chinook	2	Sim.	44.33 (± 5.48)	63.83 (± 5.32)	54.33 (± 5.47)
Connect 5	2	Turn	53.00 (± 5.66)	51.17 (± 5.54)	69.33 (± 5.21)
Knightthrough	2	Turn	48.00 (± 5.66)	65.67 (± 5.38)	85.00 (± 4.05)
Othello	2	Turn	45.83 (± 5.58)	54.67 (± 5.57)	47.83 (± 5.52)
Skirmish	2	Turn	58.50 (± 5.28)	61.67 (± 5.29)	54.33 (± 5.41)
TCCC4	2	Turn	54.33 (± 5.45)	60.17 (± 5.26)	50.83 (± 5.54)
TCCC4 3P	3	Turn	47.33 (± 5.62)	52.17 (± 5.61)	65.00 (± 5.31)
Overall			**55.39**	**61.56**	**64.07**

The *Unexplored Action Urgency* extension yields significant improvements on many of the games and as expected it needs an informed playout strategy to work. This is confirmed by the fact that no improvement is gained when paired with the random playouts of UCT while the MAST pairing shows significant improvements in six out of the twelve games. Using plain UCT seem even to be harmful to *Chinese Checkers 3P*, *Skirmish* and *TCCC*. Note that this combination should never be used in practice as the extension demands an informed playout strategy and with MAST the results of these games go back to being within neutral significance limits. Because the development of MAST was inspired by problems UCT encountered when playing *Breakthrough*, it is not surprising to see the biggest improvements there.

When the extensions have been joined together (Table 5.2), the improvements they provide do not seem to overlap except in the case of *Battle* and *Connect 5*. *Battle*, even though still at approximately 75%, does worse than without the *Unexplored Action Urgency* in the UCT case. *Early Cutoffs* improves *Connect 5* with UCT and *Unexplored Action Urgency* with MAST, but combined they seem to cancel out the improvements of the other. With MAST we have significant improvements on eight of the twelve games plus *Othello* being close as it did go over the significance threshold in Table 5.1 and does not activate the *Early Cutoff* extension.

Both extensions, *Early Cutoffs* and *Unexplored Action Urgency*, apart or together did not improve upon either of the three player games, no matter what base agent they were combined with.

Regarding the question if these extensions can improve upon the state-of-the-art in GGP we see that there are undeniable significant improvements for 8 of the 12 games resulting in approximately 14% overall improvement on this set of games.

5.3 Summary

In this chapter we described how two MCTS extensions can be generalized for practical use in GGP agents. The two extensions show genuine improvements to basic agents without, as far as we can tell, displaying any severe adverse effects when they do not help. When merged with a GGP agent that is armed with extensions that allow it to reach competition level strength, both these extensions still offer overall improvements.

Chapter 6

MCTS Simultaneous Moves Solver

When searching game trees significant benefits can be achieved by pruning away branches which under no circumstances can produce information beneficial to the decision being made at the root. The most fundamental of all such pruning methods is *Alpha-Beta* pruning (Knuth & Moore, 1975; Russell & Norvig, 2010). It applies to zero-sum sequential two-player games such as *Chess* and *Checkers*. Extending the kind of safe pruning alpha-beta does beyond the constraints of sequential (turn-taking) zero-sum two-player games has also been achieved for sequential games without zero-sum rewards and more than two players (Sturtevant & Korf, 2000; Sturtevant, 2005). The first MCTS solver (Winands, Björnsson, & Saito, 2008) backing up *MiniMax* values was made for the game *Lines of Action* and recently a generalized MCTS solver for zero-sum games which allows for alpha-beta style pruning under the name Score Bounded MCTS (Cazenave & Saffidine, 2011). This method allows safe pruning of proven irrelevant parts of the search tree.

In this chapter we focus on how to implement a generalized MCTS solver with safe pruning that also works on game trees without the sequential constraint, but are still confined to two players and zero-sum rewards. We do this by treating the game tree nodes as stacked matrix games (normal-form games) which propagate back up the tree their *Nash Equilibria* values.

We first briefly explain the *Nash Equilibria*, *Normal-Form* games, and *Score Bounded MCTS* before describing the proposed pruning method and how MCTS can be extended with it. This is continuation on a joint work with Abdallah Saffidine and Michael Buro on *Alpha-Beta* pruning for games with simultaneous moves (Saffidine et al., 2012).

6.1 Nash Equilibrium and Normal-Form Games

A Nash equilibrium is a strategy profile for all players for which no player can increase his payoff by deviating unilaterally from his strategy. In the case of zero-sum two-player games, all Nash equilibria result in the same payoff, called the *value* of the game. *MiniMax* values are a special case of the Nash equilibrium where only one of the two players decides how the game progresses in each state. When faced with simultaneous actions, Nash equilibrium strategies are often mixed strategies in which actions are performed with certain probabilities (e.g., the only Nash equilibrium strategy for *Rock-Paper-Scissors* is playing rock, paper, and scissors randomly with probability $1/3$ each).

Two-player zero-sum games presented in normal-form lists all action combination payoffs in a matrix for player MAX where the rows and columns represent the separate action sets of the players. When working with normal-form games it is sometimes possible to simplify them based on action domination. This happens when no matter how the opponent acts, the payoff for some action a is always less or equal to the payoff for some other action b or a mixed strategy not containing a. In this situation there is no incentive to play action a and it can be ignored. Action domination is the basis for our pruning.

6.2 Score Bounded MCTS

An MCTS solver which backs up exact *MiniMax* values of the sequential zero-sum two-outcome game *Lines of Action* was introduced in (Winands et al., 2008). *Score bounded MCTS* (Cazenave & Saffidine, 2011) expands on this idea and generalizes the MCTS solver concept to any sequential zero-sum game. Score bounded search allows for pruning in the absence of exact *MiniMax* values as long as there is some information available to establish upper and lower bounds on the *MiniMax* values.

Because simulations do not usually methodically explore the game tree, it is to be expected that we cannot easily assign *MiniMax* values to the states when we explore them as we are only sampling the subtree below. Even though we may not have explored every reachable state, the sampling information builds up and can be used to get tighter and tighter bounds on state values. These bounds are called *Pessimistic* and *Optimistic*, referring to the payoff MAX believes can be achieved in the worst and best case, respectively. Such pessimistic and optimistic bounds were also used in the *B** algorithm (Berliner, 1979). The default bounds are the minimum and maximum achievable values. Instead of backing up a *MiniMax* value, the pessimistic and optimistic bounds of a state

are deduced via the *MiniMax* rule from the matching bounds of subsequent states. If the pessimistic value and optimistic value become equal the real *MiniMax* value of the state has been found and it has been proved. Having these bounds also allows pruning in a shallow *Alpha-Beta* fashion if at a MAX node there is a child node with an optimistic value equal or lower than the pessimistic value of the current node and vice versa at the MIN nodes.

6.3 Score Bounded Simultaneous MCTS

To adapt the *Score Bounded MCTS* to simultaneous games we still keep the pessimistic and optimistic values in the game tree nodes, but now we view the bounds of the child states as a normal-form game for which we can calculate the game theoretical *Nash Equilibrium* value which is then used as the node's bounds. For node n the pessimistic value is the Nash value of the normal-form game assembled from the pessimistic values of its children and the same for the optimistic value of node n.

With these bounds serving as the *Pessimistic* and *Optimistic* bounds of the *Score Bounded MCTS* we obtain the real game theoretical values of the game tree nodes. For the remainder of the chapter we use the terms pessimistic and optimistic values for the lower and upper bounds respectively, the minimum (0) and maximum (100) scores in GGP as the score limits so if MAX gets a score of x, MIN will always get $100 - x$. P will stand for the pessimistic matrix and O for the optimistic one. Furthermore we will also from now on assume play from the perspective of MAX who is the row player.

Let q be a position in the game tree with m actions for MAX and n actions for MIN. When q is created it sets up two $m \times n$ matrices, P for the pessimistic values of all child states and O for the optimistic values. P is initialized with 0 and O with 100, which also are the Nash values of such initialized matrices in the children. Terminal states only get a single value, the actual game outcome in that state which is backed up to its corresponding cell in both P and O of the parent. At some point these true outcomes will affect the Nash values of the P and O in the parent which in turn will be propagated to its parent, and so on. Calculation of the Nash values for zero-sum normal-form game is done via the Linear Program (LP) in Figure 6.1 which uses the pessimistic value (p) calculation as an example. To work, this method needs all the elements of the equation matrix to be positive and it does not affect the outcome if all the matrix elements are offset by a constant value which is then subtracted from the outcome. As the lowest score in GGP is 0 it is enough to use 1 as this constant. In Figure 6.1, P^{+1} represents P offset by $+1$. This is why p is

$$x = \begin{pmatrix} x_1 \\ \vdots \\ x_{a-1} \\ x_{a+1} \\ \vdots \\ x_m \end{pmatrix}, P = \begin{pmatrix} p_{1,1} & \cdots & p_{1,n} \\ \vdots & & \vdots \\ p_{m,1} & \cdots & p_{m,n} \end{pmatrix}, e = \begin{pmatrix} 1 \\ \vdots \\ 1 \end{pmatrix}$$

$$\min x, \text{subject to } x^t P^{+1} \geq e, x \geq 0, p = \frac{1}{\sum x} - 1$$

Figure 6.1: Calculating Nash Equilibrium value in a zero-sum Normal-Form game

G:

	b_1
a_1	50
a_2	value(H)

H:

	b_2	b_3
a_3	40	?
a_4	30	?

Figure 6.2: Example Normal-Form games

subtracted by 1. The optimistic value is done exactly the same using the optimistic matrix also offset by $+1$.

6.4 Simultaneous Move Pruning

Consider game G in Figure 6.2. The row player will only select action a_2 if the value of subgame H is greater than 50. Now consider subgame H: no matter what the values of cells (a_3, b_3) and (a_4, b_3) are, the best value the row player can hope for at this point is 40. As a result, we do not even need to compute the exact value for H and it can be pruned.

Figure 6.2 is an example of when the opponent can lower an already attained proven value if given the opportunity and therefore the action leading to it should never be played. Also it cannot contribute to the game value. Even if we assume that the unknown values of game H are equal to the maximum score of the game the row player's Nash equilibrium value will still be 40 and will not change unless the unknown values cause it to lower. With this limited information we can deduce that 40 is an upper bound on the game value of H.

As mentioned above we will use action domination as the pruning criterion because, similar to that of the *Alpha-Beta* algorithm, such sub-tree pruning indicates that we have proof that they will under no circumstances improve upon the current guaranteed payoff assuming rational players.

$$x = \begin{pmatrix} x_1 \\ \vdots \\ x_{a-1} \\ x_{a+1} \\ \vdots \\ x_m \end{pmatrix}, P = \begin{pmatrix} p_{1,1} & \cdots & p_{1,n} \\ \vdots & & \vdots \\ p_{a-1,1} & \cdots & p_{a-1,n} \\ p_{a+1,1} & \cdots & p_{a+1,n} \\ \vdots & & \vdots \\ p_{m,1} & \cdots & p_{m,n} \end{pmatrix}, f = \begin{pmatrix} o_{a,1} & \cdots & o_{a,n} \end{pmatrix}$$

$$x^t P \geq f, 0 \leq x \leq 1, \textstyle\sum_i x_i = 1$$

Figure 6.3: System of inequalities for deciding whether row action a is dominated

$$x = \begin{pmatrix} x_1 & \cdots & x_{b-1} & x_{b+1} & \cdots & x_m \end{pmatrix},$$

$$O = \begin{pmatrix} o_{1,1} & \cdots & o_{1,b-1} & o_{1,b+1} & \cdots & o_{1,n} \\ \vdots & & \vdots & \vdots & & \vdots \\ o_{m,1} & \cdots & o_{m,b-1} & o_{m,b+1} & \cdots & o_{m,n} \end{pmatrix}, f = \begin{pmatrix} p_{1,b} \\ \vdots \\ p_{m,b} \end{pmatrix}$$

$$Ox^t \leq f, 0 \leq x \leq 1, \textstyle\sum_i x_i = 1$$

Figure 6.4: System of inequalities to decide if a column action b is dominated

To determine if a MAX action is dominated we must establish if there exists a strategy among the other MAX actions that even in the worst case is still better than or equal to the best case of the action selected for domination checking. Similarly for the MIN actions, an action may be pruned if the worst case bounds of using said action as a pure strategy for MIN is equal or higher than a strategy MIN can assemble from the other available actions.

To calculate the domination we use P, O and LP. Figures 6.3 and 6.4 describe the inequalities used for this. If the problem in Figure 6.3 has a feasible solution then the MAX action a is dominated because there exists a strategy using the other pessimistic estimates, that pays more than the optimistic estimate of a no matter what action MIN chooses. In other words, if MAX takes action a MIN can make sure that the game value will stay fixed or go down from there on. Likewise if the problem in Figure 6.4 has a feasible solution then the MIN action b is dominated because there exists a strategy using the other actions optimistic estimates, that pays less than the pessimistic estimate of b no matter what action MAX chooses.

6.5 Implementation

Having to solve LPs we can expect some overhead once the solver has been joined with the MCTS agent. It is therefore important to try to keep the solver calculations to a minimum. In CADIAPLAYER the solver was implemented using the *GNU Linear Programming Kit* (GLPK) and a player inheriting the MCTS player. Every state that needs to calculate its bounds and Nash values is issued a special solver node which contains all LP's and last known bounds of each child. All solver nodes are stored in a hash map keyed on the state id. This map also keeps track of the deepest ply the state has occurred so those nodes no longer needed may be deleted to free up memory. The solver nodes are also able to map action ids to the 1-based index used by GLPK for the LP problem matrices and back.

Solver nodes get created only if the state they represent is a terminal state or if during the back-propagation step its child state on the simulation path already has a solver node that does not still only have 0 and 100 as its bounds. Solver states are updated during the back-propagation step if their child state has a solver state which bounds have changed. If a state needs to be updated, the new value is inserted into all LP's and the pruning check is run. At the moment all unpruned actions of both participants are checked, alternating between the action sets while something gets pruned. Now the new pessimistic and optimistic values of the state are calculated. Because both the pruning and the bounds will not change unless at least one complete row or column has changed, no calculation is done until the solver node has incurred at least as many changes as the length of the shorter dimension of its bounds matrices.

In (Winands et al., 2008) the authors identified a problem with how solved loss nodes can affect the MCTS averages. If you ignore them their parent may become too biased towards only a subset of its children. We therefore added an option into the solver such that it can either backup the values of the solved nodes if selected during the agent simulation, or to avoid the solved nodes and always try one of the unsolved children. The latter approach, although maybe not ideal for competitive play, may be helpful for solving games. If the game gets solved as a win in the root, the agent stops running simulations and plays out the guaranteed win as a solved win can only happen through pure strategies.

Pruning is achieved by overwriting the value calculations of available moves in the selection step such that pruned moves are given a large negative value instead of their normal UCT value.

6.6 Empirical Evaluation

We set up a four game tournament between normal UCT and UCT augmented with the each of the two solver variants. In the following section the variant also selecting solved nodes is called S-UCT and the one that explores only unsolved nodes is called E-UCT.

6.6.1 Setup

In all games between the solver agents and the UCT agent, time controls were set to 10 seconds and each agent played 150 games as either side, 300 in all for each data point. All games used are described in Appendix C. All agents and experiments ran on a single Intel Xeon 2.66GHz CPU. The *Goofspiel* games were encoded such that they contain no transpositions.

6.6.2 Results

Tables 6.1 and 6.2 show the results of the tournaments. The first column shows the game played, and the numbers in parenthesis in the *Goofspiel* rows indicate the number of cards dealt to each player. The *Node Expansions* column shows how many nodes the agent expanded on average during each turn in all 300 games before reaching a solved position. The accompanying percentage is the ratio of average node expansions relative to those of the UCT opponent during that tournament. The third column *Simulation Count*, displays the same information as the previous one, except the numbers are for complete simulations per turn. The last column shows the winning percentage of the MCTS solver agent with a 95% confidence interval.

Table 6.1: E-UCT vs. UCT

Game	Node Expansions	Simulation Count	Win%
Bidding TTT	88,629 (60.01%)	6,383 (46.11%)	9.50 ($\pm$ 3.24)
Goofspiel(6)	29,466 (28.38%)	4,538 (27.66%)	35.33 ($\pm$ 4.49)
Goofspiel(9)	21,531 (65.89%)	2,699 (59.57%)	37.00 ($\pm$ 4.72)
Pawn Whopping	35,613 (81.31%)	2,443 (43.67%)	46.67 ($\pm$ 4.88)
Average	**58.90%**	**44.25%**	**32.13**

Table 6.2: S-UCT vs. UCT

Game	Node Expansions	Simulation Count	Win%
Bidding TTT	102,256 (91.68%)	9,259 (102.98%)	45.33 (± 5.31)
Goofspiel(6)	110,743 (74.18%)	3,4671 (123.96%)	47.50 (± 3.53)
Goofspiel(9)	47,793 (81.17%)	13,464 (120.81%)	51.17 (± 4.52)
Pawn Whopping	40,415 (94.31%)	5,663 (106.45%)	48.33 (± 4.85)
Average	**85.33%**	**113.55%**	**48.08**

Clearly, avoiding simulating solved nodes has adverse effects on the agent as the E-UCT agent performs relatively much worse than S-UCT. Added information, if any, does not outweigh the cost of the extra exploration. When it comes to the average winning percentage, S-UCT seems to be on par with UCT. The solver overhead of S-UCT in node expansions is very game specific, averaging around 85%, but more interestingly the simulation count goes beyond that of the normal UCT. The solved nodes shorten the simulations enough to increase their number even with the decrease in node expansions. The increase in number of simulations does, however, not translate into a stronger player. If anything, the solver-based player might even be slightly worse.

So the question becomes: If we could remove some of the overhead, will we get a stronger agent? The overhead of solving the LP problems may be reduced simply by switching to one of the commercial LP kits available that is faster than GLPK. To examine how the S-UCT solver would do if the overhead were absent we set up an additional 300 game tournament where the node expansions per turn were fixed for both agents. These tournament results are shown in Table 6.3. Now the simulation count rises even more, but still the strength of S-UCT is not much affected. Apparently, UCT is already picking good moves. Also, it is one thing to prove a game value, but in game positions like *Rock-Paper-Scissors* knowing the game value does not help you to win.

To see if it is a trait of the S-UCT that the extra simulations do not help, we set up one final 300 game tournament. This time we used normal UCT in place of S-UCT and allowed it to have the same simulation count advantage as S-UCT had managed in Table 6.3. The results are in Table 6.4. The first result column shows the number of simulations the first player was limited to (see the Simulation Count column in Table 6.3). The second result column contains the second player's simulation limit which equals 100% of the Simulation Count column in Table 6.3. The last column gives the winning percentage of the first player, the one which was allowed more simulations. The extra simulations do not seem to help UCT to improve much, so it is not just an artifact of having S-UCT in

Table 6.3: Fixed Node Expansions: S-UCT vs. UCT

Game	Node Expansions	Simulation Count	Win%
Bidding TTT	100,000 (100.00%)	8,341 (113.56%)	47.67 (± 5.17)
Goofspiel(6)	100,000 (100.00%)	27,350 (160.85%)	47.17 (± 3.64)
Goofspiel(9)	50,000 (100.00%)	10,079 (135.25%)	52.33 (± 4.45)
Pawn Whopping	50,000 (100.00%)	4,838 (113.36%)	49.67 (± 4.56)
Average	**100.00%**	**130.76%**	**49.21**

Table 6.4: Uneven Simulation Count: UCT vs. UCT

Game	P1 Simulation Limit	P2 Simulation Limit	Simulation Ratio	P1 Win %
Bidding TTT	8,341	7,345	113.56%	53.00 (± 5.12)
Goofspiel(6)	27,350	17,003	160.85%	49.83 (± 2.51)
Goofspiel(9)	10,079	7,452	135.25%	48.33 (± 4.41)
Pawn Whopping	4,838	4,268	113.36%	53.33 (± 4.54)
Average			**130.76%**	**51.12**

control of those extra simulations. It seems that if might take more than just boosting UCT to significantly improve its playing strength in these games.

To get an idea of the amount of pruning this kind of UCT solver may be able to provide we had the E-UCT agent solve the root node of *Goofspiel* with 4, 5, or 6 six cards per player. The reason for not using the S-UCT agent was simply that the focus of UCT keeps it from exploring enough for the root to get solved within any reasonable time limits. The results are in Table 6.5 and the columns from left to right show the time the solution took in seconds, the number of solver nodes created, and the ratio of solver nodes relative to the full game tree. All numbers are an average of 10 runs.

Table 6.5 shows that there is much that even MCTS can prune away in these simultaneous game as in the best case here E-UCT has only created solver nodes for about quarter of the tree to have solved the game value at the root.

Table 6.5: Solving *Goofspiel* on the Startclock with E-UCT

Game	Seconds	Nodes	Ratio
Goofspiel(4)	1.074	669	50.95%
Goofspiel(5)	11.314	8,934	27.22%
Goofspiel(6)	510.055	380,142	32.17%

6.7 Summary

In the chapter we have presented a method to incorporate a solver in MCTS which can handle simultaneous-move zero-sum games. This preliminary look into the subject indicates that the solver can be incorporated without much overhead, however, the benefits in terms of an improved playing strength are unclear, at least for games we tried. We also showed that large sections of the game trees in simultaneous-move games can be pruned by a MCTS solver.

Chapter 7

Game-Tree Properties

Some games are more problematic for MCTS than others. Often, more game-specific knowledge is needed to make the simulations effective. In the previous chapters we have aimed at developing search-control methods that are robust across a large range of games. However using more game specific properties tailored towards certain game types may be beneficial. In this chapter we identify high-level properties that are commonly found in game trees and measure how they affect the performance of MCTS. By doing so we hope to lay some groundwork for searching smarter rather than relying on brute force. This work helped for example in developing the *Early Cutoff* extension in Chapter 5 and the magnitude of a problem we have called *Optimistic Moves* which was the inspiration of MAST in Chapter 4. In this section we will not be using CADIAPLAYER as a testbed. Instead we use simple custom-made games that allow us to vary the properties of interest and obtain large datasets much quicker than using GGP.

7.1 Properties

In the following we describe tree properties we identified as being particularly important for MCTS performance that are general enough to be found in a wide variety of games. It is by no means intended as a complete list.

7.1.1 Tree Depth vs. Branching Factor

The most general and distinct properties of game trees are their depth and width, so the first property we investigate is the balance between the depth and the branching factor.

These are properties that can quickly be estimated using simulations at runtime. With increasing depth the simulations become longer and therefore decrease the number of samples that make up the aggregated values at the root. Also, longer simulations are more likely to result in improbable lines of simulation play. Increasing the branching factor results in a wider tree, decreasing the proportion of lines of play investigated. The depth and width relative to the number of nodes in the trees can be varied, allowing us to answer the question if MCTS favors one over the other.

7.1.2 Progression

Some games progress towards a natural termination with every move made while other allow moves that only maintain a status quo. Examples of naturally progressive games are *Connect 4*, *Othello* and *Quarto*, while on the other end of the spectrum we have games like *Skirmish*, *Chess* and *Bomberman*. Games that can go on infinitely in practice have some maximum length imposed on them. When reaching this length limit the game either results in a draw or is scored based on the current board position. This is especially common in GGP games. When such artificial termination is applied, progression is affected because some percentage of simulations do not yield useful results. This is especially true when all artificially terminated positions are scored as a draw.

7.1.3 Optimistic Moves

Optimistic moves is a name we have given to moves that achieve very good result for its player assuming that the opponent does not realize that this seemingly excellent move can be refuted right away. The refutation is for example accomplished by capturing a piece just moved, which MCTS thinks is on its way to ensure victory for the player. This situation arises when the opponent's best response gets lost among the other moves available to the simulation action selection policies. In the worst case this causes the player to play the optimistic move and lose the piece for nothing. Given enough simulations MCTS eventually becomes wice to the fact that this move is not a good idea, but at the cost of running many simulations to rule out this move as an interesting one.

Figure 7.1 is an example of an optimistic move. It is Black's turn to move and an MCTS agent would initially find it most attractive to move the far advanced black piece one square forward (*b4-b3*). However, this is obviously a bad move because White can capture the piece with *a2-b3*; this is actually the only good reply for White as all the others lead to a forced win for Black (*b3-a2* followed by *a2-b1*). This can work both ways as the

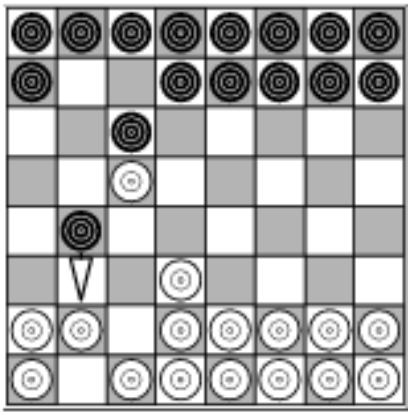

Figure 7.1: Breakthrough game position

simulations can also detect such a move for the opponent and thus waste simulations on a preventive move when one is not needed.

7.2 Empirical Evaluation

We used custom-made games for evaluating the aforementioned properties, as described in the following setup subsection. This is followed by subsections detailing the individual game property experiments and their results.

7.2.1 Setup

All games have players named White and Black and are turn-taking with White going first. The experiments were run on Linux based dual processor Intel(R) Xeon(TM) 3GHz and 3.20GHz CPU computers with 2GB of RAM. Each experiment used a single processor.

The scoring for each game is in the interval of $[0, 1]$ and MCTS uses $C_p = 1/\sqrt{2}$ with a uniform random playout strategy. The node expansion strategy adds only the first new node encountered to the MCTS tree and neither a discount factor nor other modifiers are used in the back-propagation step. The players only deliberate during their own turn. A custom-made tool is used to create all games and agents. This tool allows games to be set up as FEN strings[1] for boards of any size and by extending the notation one can select from custom predefined piece types. Additional parameters are used to set game options like goals (capture all opponents or reach the back rank of the opponent), artificial termination depth and scoring policy, and whether squares can inflict penalty points.

[1] Forsyth-Edwards Notation. http://en.wikipedia.org/wiki/FEN

7.2.2 Tree Depth vs. Branching Factor

The games created for this experiment can be thought of as navigating runners through an obstacle course where the obstacles inflict penalty points. We experimented with three different setups for the penalties as shown in Figure 7.2. The pawns are the runners, the corresponding colored flags their goal and the big X's walls that the runners cannot go through. The numbered squares indicate the penalty inflicted when stepped on. White and Black each control a single runner that can take one step forward each turn. The board is divided by the walls so the runners will never collide with each other. Every time the runner takes a step forward, additionally a new lane may be selected on their side of the wall. For example, on its first move in the setups in Figure 7.2 White could choose from the moves *a1-a2*, *a1-b2*, *a1-c2* and *a1-d2*. All but one of the lanes available to each player incur one or more penalty points. The game is set up as a turn taking game but both players must make an equal number of moves and therefore both will have reached the goal before the game terminates. This helps in keeping the size of the tree more constant. The winner is the one that has fewer penalty points upon game termination. The optimal play for White is to always move on lane *a*, resulting in finishing with no penalty points, while for Black the optimal lane is always lane *i*. This game setup allows us to control the properties of interest by varying the board size. The depth of the tree to be tuned by setting the lanes to a different length. The branching factor is tuned through the number of lanes per player. To ensure that the amount of tree nodes does not collapse with all the transpositions possible in this game, the game engine produces state ids that depend on the path taken to the state it represents. Therefore states that are identical will be perceived as different ones by the MCTS algorithm if reached through different paths. This state id scheme was used only for the experiments in this subsection.

The first game we call *Penalties* and can be seen in Figure 7.2 (a). Here all lanes except for the safe one have all steps giving a penalty of one. The second one we call *Shock*

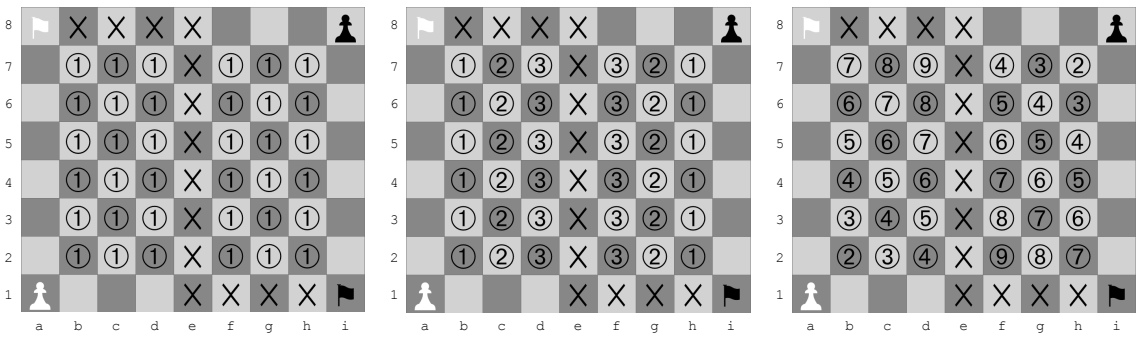

Figure 7.2: (a) Penalties Game, (b) Shock Step Game, and (c) Punishment Game

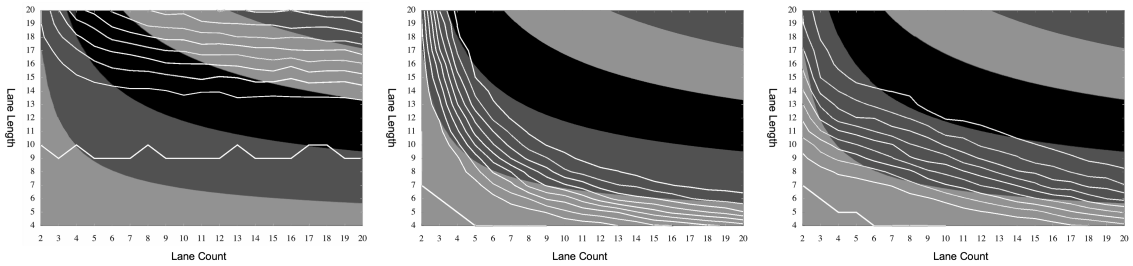

Figure 7.3: (a) Penalties Results, (b) Shock Step Results, and (c) Punishment results

Step and is depicted in Figure 7.2 (b). Here the squares of each non-safe lane have the same amount of penalty, but the penalty by the lane's distance from the safe lane. The third game called *Punishment* is shown in Figure 7.2 (c). The penalty amount is as in the *Shock Step* game except now it also gets progressively larger the further the runner has advanced.

We set up races for the three games with all combinations of lanes of length 4 to 20 squares and number of lanes from 2 to 20. We ran 1000 games for each data point. MCTS runs all races as White against an optimal opponent that always selects the move that will traverse the course without any penalties. MCTS was allowed 5000 node expansions per move for all setups. The results from these experiments are shown in Figure 7.3. The background depicts the trend in how many nodes there are in the game trees related to number of lanes and their length. The borders where the shaded areas meet are node equivalent lines, that is, along each border all points represent the same node count. When moving from the bottom left corner towards the top right one we are increasing the node count exponentially. The overlaid lines, called win lines, are the data points gathered from running the MCTS experiments. The line closest to the bottom left corner represent the 50% win border (remember the opponent is perfect and a draw is the best MCTS can get). Each borderline after that shows a 5% lower win ratio from the previous one. This means that if MCTS only cares how many nodes there are in the game tree and its depth or width has no bearing on the outcome, then the win lines should follow the trend of the background plot exactly.

The three game setups all show different behaviors related to how depth and branching factor influence the strength of MCTS. When the penalties of any of the sub-optimal moves are minimal as in the first setup, bigger branching factor seems to have almost no effect on how well the player does. This is seen by the fact that when the number of nodes in the game tree increases due to more lanes, the win lines do not follow the trend of the node count which moves down. They stay almost stationary at the same depth.

As soon as the moves can do more damage as in the second game setup we start to see quite a different trend. Not only does the branching factor drag the performance down, it does so at a faster rate than the node count in the game tree is maintained. This means that MCTS is now preferring more depth over bigger branching factor. Note that as the branching factor goes up so does the maximum possible penalty.

In the third game the change in branching factor keeps on having the same effect as in the second one. In addition, now that more depth also raises the penalties, MCTS also declines in strength if the depth becomes responsible for the majority of game tree nodes. This is like allowing the players to make bigger and bigger mistakes the closer they get to the goal. This gives us the third trend where MCTS seems to favor a balance between the tree depth and the branching factor.

To summarize, MCTS does not have a definite preference when it comes to depth and branching factor and its strength cannot be predicted from those properties only. It appears to be dependent on the rules of the game being played. We show that games can have big branching factors that pose no problem for MCTS and vice versa. Still with very simple alterations to our abstract game we can see how MCTS does worse with increasing branching factor and can even prefer a balance between it and the tree depth.

7.2.3 Progression

For experimenting with the progression property we created a racing game similar to the one used in the tree depth vs. width experiments. Here, however, the size of the board is kept constant (20 lanes of length 10) and the runners are confined to their original lane by not being allowed to move sideways. Each player, White and Black, has two types of runners, ten in total, initially set up as shown in Figure 7.4. The former type, named *active* runner and depicted as a pawn, moves one step forward when played whereas the second, named *inactive* runner and depicted by circular arrows, stays on its original square when played. In the example shown in the figure each player has 6 active and 4 inactive runners. In the context of GGP each inactive runner has only a single *noop* move available for play. By changing the ratio between runner types a player has, one can alter the progression property of the game: the more active runners there are, the faster the game progresses (given sub-optimal play). The game terminates with a win once a player's runner reaches a goal square (a square with the same colored flag).

We also impose an upper limit on the number of moves a game can last. A game is terminated artificially and scored as a tie if neither player has reached a goal within the upper limit of moves. By changing the limit one can affect the progression property of

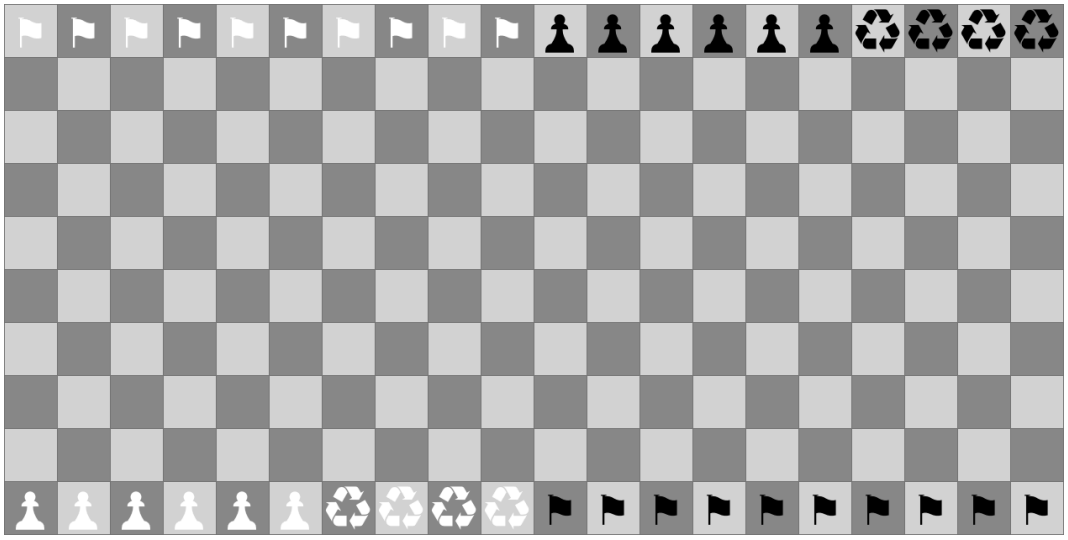

Figure 7.4: Progression game

the game: the longer a game is allowed to last the more likely it is to end in a naturally resulting goal rather than being depth terminated, thus progressing better. We modify this upper limit of moves in fixed step sizes of 18, which is the minimum number of moves it takes Black to reach a goal (Black can first reach a flag on its 9th move, which is the 18th move of the game as White goes first). A *depth factor* of one thus represents an upper limit of 18 moves, depth factor of two 36 moves, etc.

In the experiments that follow we run multiple matches of different progression, one for each combination of the number of active runners ([1-10]) and the depth factor ([1-16]). Each match consists of 2000 games where MCTS plays White against an optimal Black player always moving the same active runner. The computing resources of MCTS are restricted to 100,000 node expansions per move.

The result is shown in Figure 7.5, with the winning percentage of MCTS plotted against both the depth factor (left) and percentage of simulations ending naturally (right). Each curve represents a game setup using a different number of active runners.[2] The overall shape of both plots shows the same trend, reinforcing that changing the depth factor is a good model for indirectly altering the number of simulations that terminate naturally (which is not easy to change directly in our game setup). When looking at each curve in an isolation we see that as the depth factor increases, so does MCTS's performance initially, but then it starts to decrease again. Increasing the depth factor means longer, and thus fewer, simulations because the number of node expansions per move is fixed. The detrimental effect can thus be explained by fewer simulations. This is better seen

[2] We omit the 5, 7, and 9 active runners' curves from the plots to make them less cluttered; the omitted curves follow the same trend as the neighboring ones.

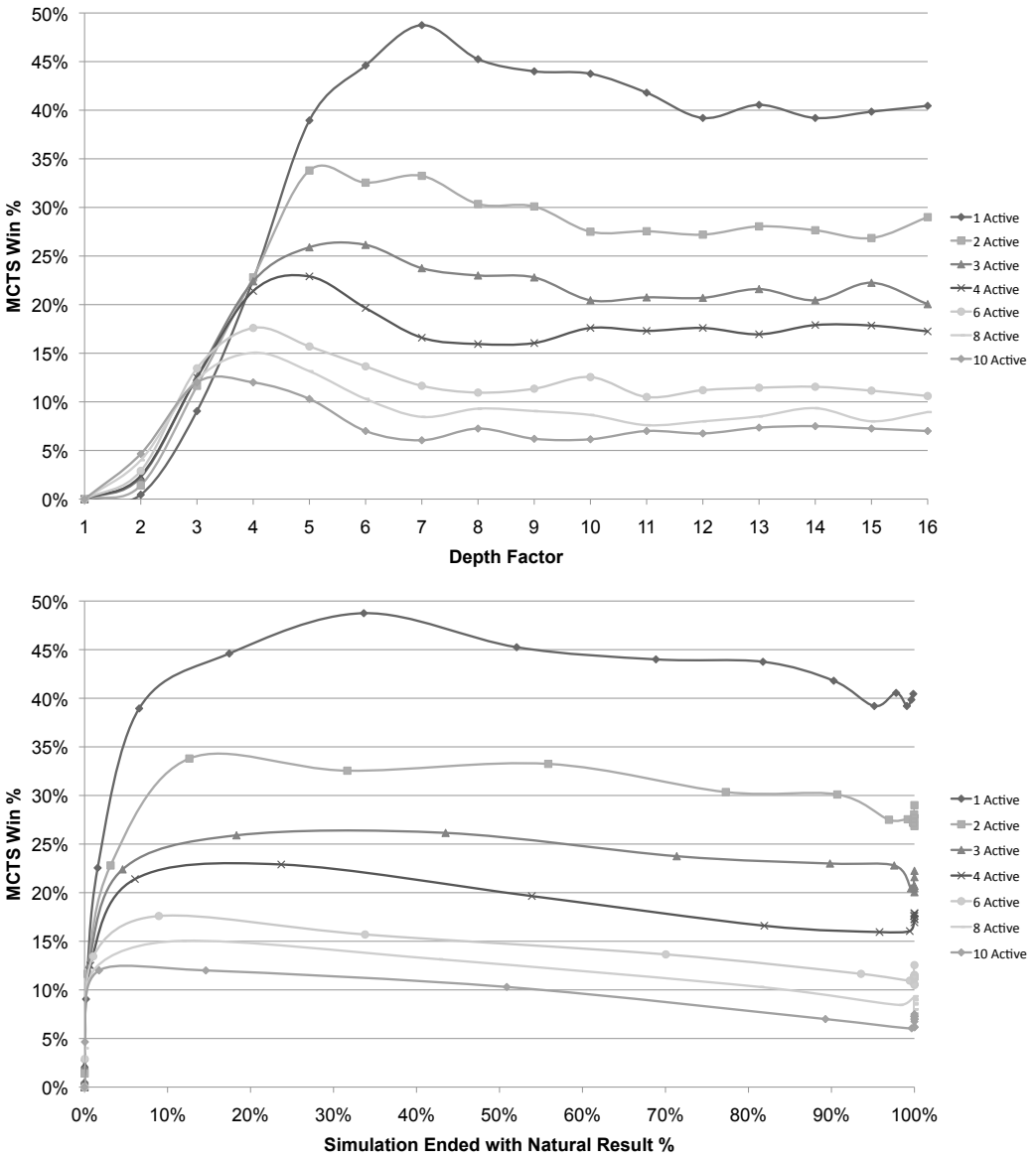

Figure 7.5: Progression Depth Factor: Fixed node expansion count

in Figure 7.6 where the result of identical experiments as in the previous figure is given, except now the number of simulations —as opposed to node expansions— is kept fixed (at 1000).

The above results show that progression is an important property for MCTS. What is somewhat surprising, however, is how quickly MCTS's performance improves as the percentage of simulations ending at true terminal states goes up. In our testbed it already reaches close to peak performance as early as 30%. This shows promise for MCTS even in games where most paths may be non-progressive, as long as a somewhat healthy ratio of the simulations terminate in useful game outcomes. Additionally, in GGP one could take advantage of this in games where many lines end with the step counter reaching the

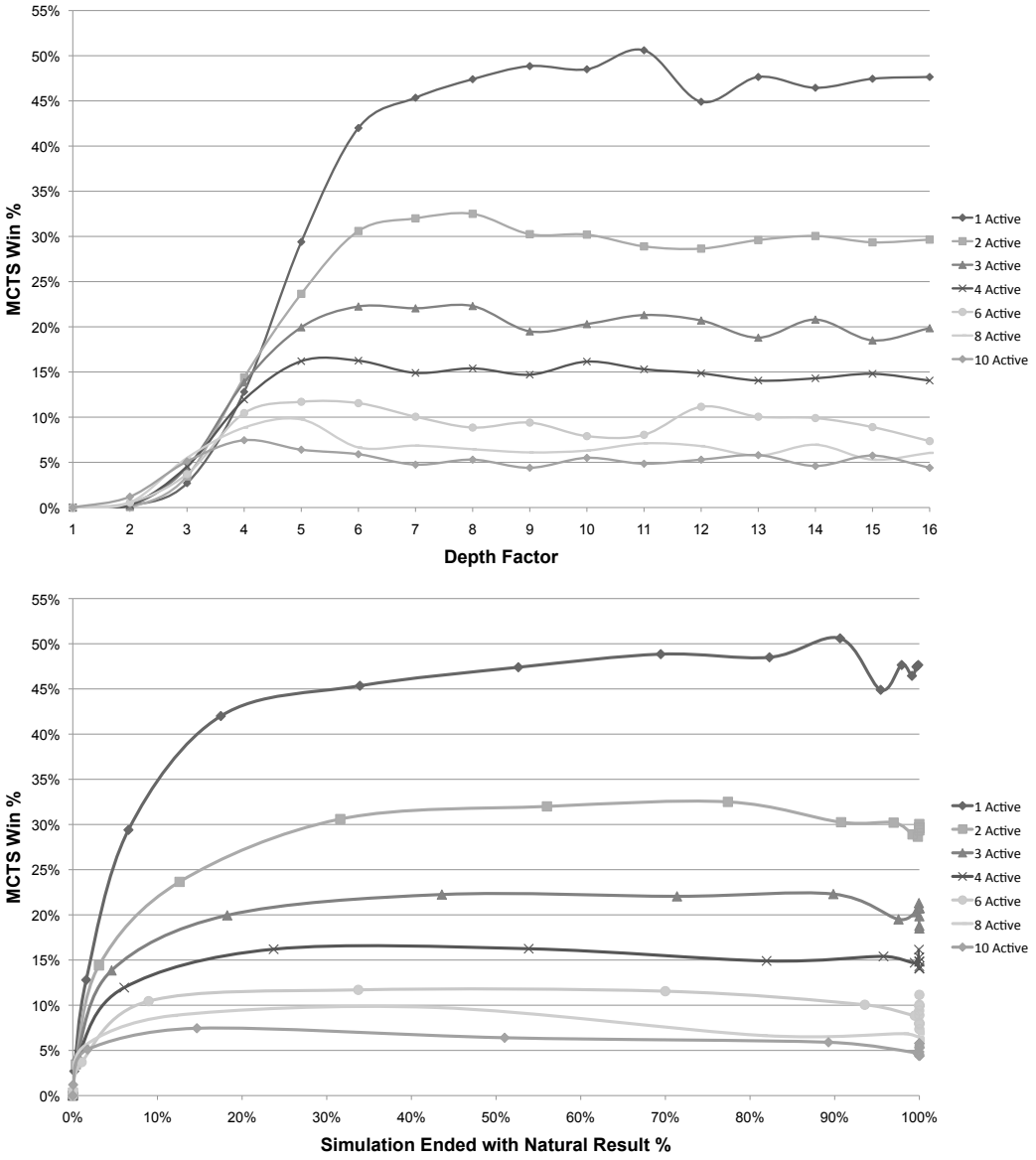

Figure 7.6: Progression Depth Factor: Fixed simulation count

upper limit, by curtailing the simulations even earlier. Although this would result in a somewhat lower ratio of simulations returning useful game outcomes, it would result in more simulations, thus potentially giving a better quality tradeoff (as in Figure 7.5).

We can see the effects of changing the other dimension —number of active runners a player has— by contrasting the different curves in the plots. As the number of active runners increases, so does the percentage of simulations ending in true terminal game outcomes, however, instead of resulting in an improved performance, it decreases sharply. This performance drop is seen clearly in Figure 7.7 when plotted against the number of active runners (for demonstration, only a single depth factor curve is shown). This behavior, however, instead of being a counter argument against progression, is an artifact

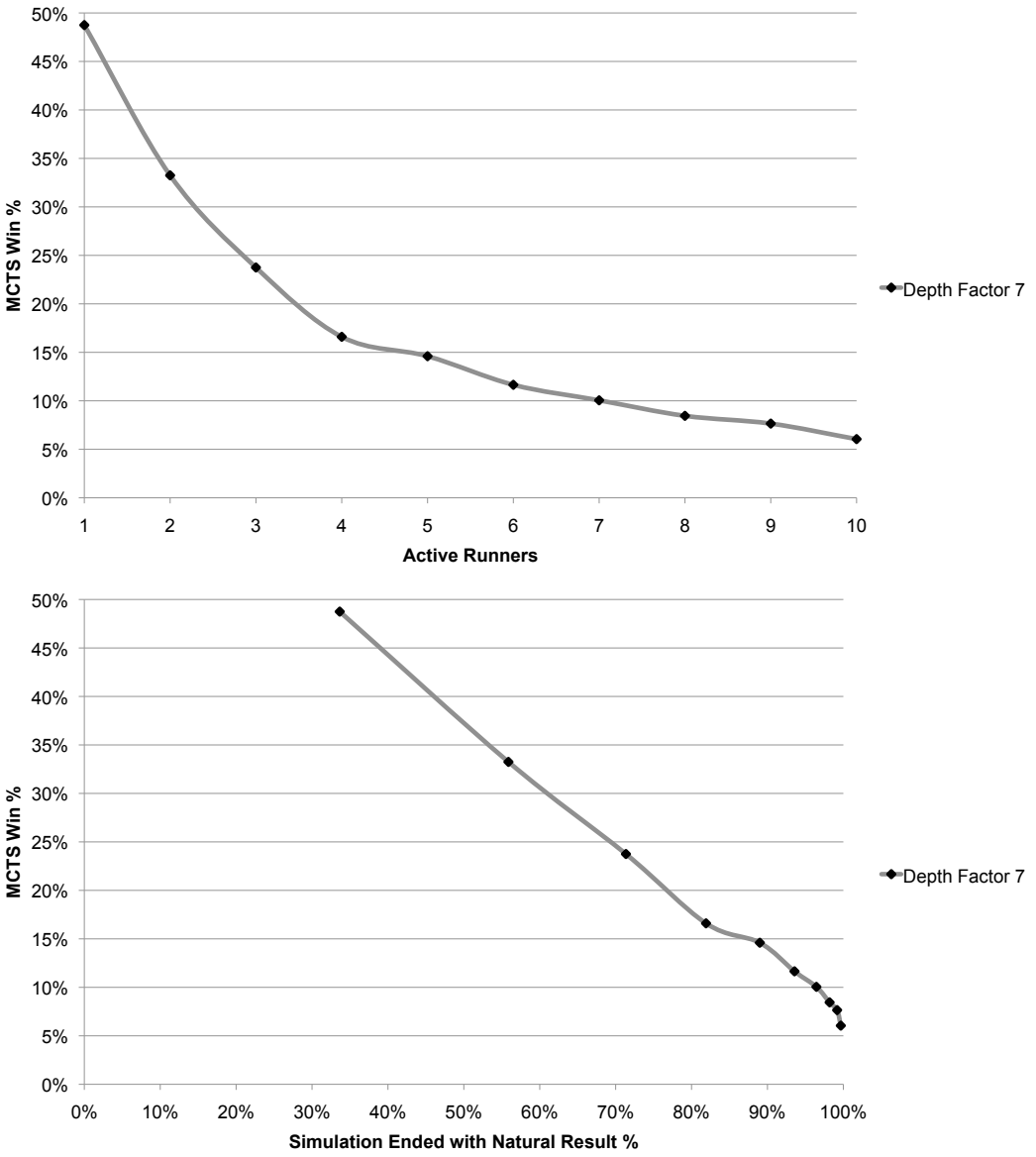

Figure 7.7: Progression Active Runners: Fixed node expansion count

of our experimental setup. In the game setup, if White makes even a single mistake, i.e., not moving the most advanced runner, the game is lost. When there are more good runners to choose from, as happens when the number of active runners go up, so does the likelihood of inadvertently picking a wrong runner to move. This game property of winning only by committing to any single out of many possible good strategies, is clearly important in the context of MCTS. We suspect that in games with this property MCTS might be more prone to switching strategies than traditional $\alpha\beta$ search, because of the inherent variance in simulation-based move evaluation. Although we did not set out to investigate this now apparently important game property, it clearly deserves further investigation in future work.

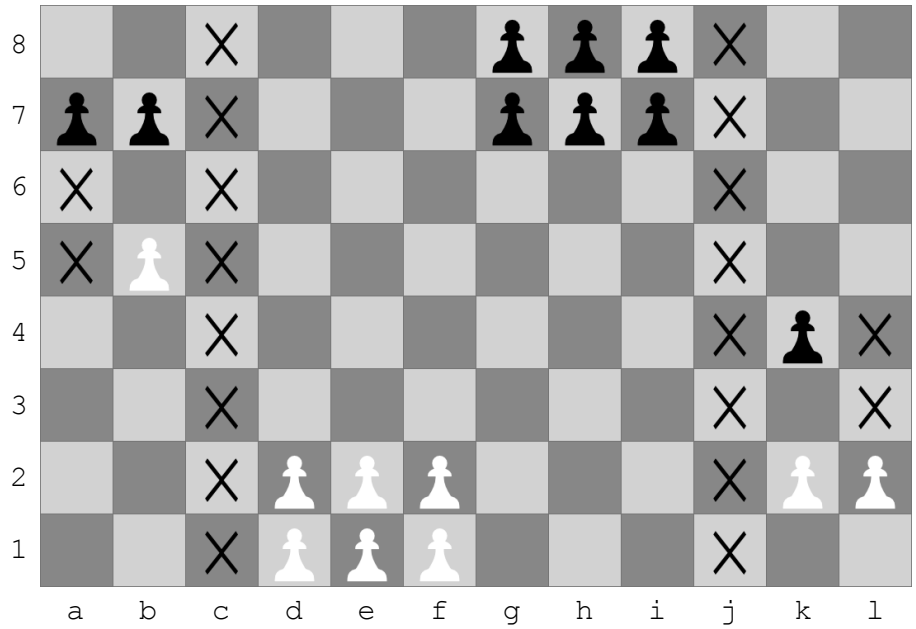

Figure 7.8: Optimistic Moves game

7.2.4 Optimistic Moves

For this experiment we observe how MCTS handles a position in a special variation of *Breakthrough* which accentuates the optimistic moves property. *Breakthrough* is a turn-taking game played with pawns that can only move one square at a time either straight or diagonally forward. When moving diagonally they are allowed to capture an opponent pawn, should one reside on the square they are moving onto. The player who is first to move a pawn onto the opponent's back rank wins. The variation and the position we set up is shown in Figure 7.8. The big X's are walls that the pawns cannot move onto. There is a clear winning strategy for White on the board, namely moving any of the pawns in the midfield on the second rank along the wall to their left. The opponent only has enough moves to intercept with a single pawn which is not enough to prevent losing. This position has also built-in pitfalls presented by an optimistic move, for both White and Black, because of the setups on the *a* and *b* files and *k* and *l* files, respectively. For example, if White moves the *b* pawn forward he threatens to win against all but one Black reply. That is, capturing the pawn on *a7* and then win by stepping on the opponent's back rank. This move is optimistic because naturally the opponent responds right away by capturing the pawn and in addition, the opponent now has a guaranteed win if he keeps moving the capturing pawn forward from now on. Similar setup exists on the *k* file for Black. Still since it is one ply deeper in the tree it should not influence White before he deals with his own optimistic move. Yet it is much closer in the game tree than the actual best moves on the board.

Table 7.1: Optimistic Moves Results

Nodes	**500,000**		**1,000,000**		**2,500,000**		**5,000,000**		**10,000,000**		**25,000,000**		**50,000,000**	
	Move	*Chosen*	*Move*	*Chosen*	*Move*	*Chosen*	*Move*	*Chosen*	*Move*	*Chosen*	*Move*	*Chosen*	*Move*	*Chosen*
Six Pawns	b5-b6	1000	b5-b6	1000	b5-b6	926	b5-b6	734	b5-b6	945	l2-k3	519	k2-k3	507
					l2-k3	44	k2-k3	153	l2-k3	37	k2-k3	481	l2-k3	484
					k2-k3	30	l2-k3	113	k2-k3	18			f2-e3	9
Four Pawns	b5-b6	1000	b5-b6	1000	b5-b6	1000	b5-b6	996	l2-k3	441	**e2-d3**	535	**e2-d3**	546
							k2-k3	3	k2-k3	438	**e2-e3**	407	**e2-e3**	449
							l2-k3	1	b5-b6	121	b5-b6	46	e2-f3	5
Two Pawns	b5-b6	980	b5-b6	989	**d2-d3**	562	**d2-d3**	570	**d2-d3**	574	**d2-d3**	526	**d2-d3**	553
	l2-k3	13	k2-k3	6	**d2-e3**	437	**d2-e3**	430	**d2-e3**	426	**d2-e3**	474	**d2-e3**	447
	k2-k3	7	l2-k3	5	b5-b6	1								
One Pawn	**d2-d3**	768	**d2-d3**	768	**d2-d3**	781	**d2-d3**	761	**d2-d3**	791	**d2-d3**	750	**d2-d3**	791
	d2-e3	232	**d2-e3**	232	**d2-e3**	219	**d2-e3**	239	**d2-e3**	209	**d2-e3**	250	**d2-e3**	209

We ran experiments showing what MCTS considered the best move after various amount of node expansions. We combined this with four setups with decreased branching factor. The branching factor was decreased by removing pawns from the middle section. The pawn setups used were the ones shown in Figure 7.8, one with the all pawns removed from files *f* and *g*, one by additionally removing all pawns from files *e* and *h* and finally one where the midfield only contained the pawns on *d2* and *i7*. The results are in Table 7.1 and the row named "Six Pawns" refers to the setup in Figure 7.8, that is, each player has six pawns in the midfield and so on. The columns then show the three most frequently chosen moves after 1000 tries and how often they were chosen by MCTS at the end of deliberation. The headers show the expansion counts given for move deliberation. The true optimal moves are printed in bold.

The setup showcases that optimistic moves are indeed a big problem for MCTS. Even at 50,000,000 node expansions the player faced with the biggest branching factor still erroneously believes that he must block the opponent's piece on the right wing before it is moved forward (the opponent's optimistic move). Taking away two pawns from each player thus lowering the branching factor makes it possible for the player to figure out the true best move (moving any of the front pawns in the midfield forward) in the end, but at the 10,000,000 node expansion mark the player is still also clueless. The setup when each player only has two pawns each and only one that can make a best move, MCTS makes this realization somewhere between the 1,000,000 and 2,500,000 mark. Finally, in the setup which only has a single pawn per player in the midfield, MCTS has realized the correct course of action before the lowest node expansion count measured.

Clearly the bigger branching factors aggravate this problem. The simulations can be put to much better use if this problem could be avoided by pruning these optimistic moves early on. The discovery process of avoiding these moves can be sped up by more greedy simulations or biasing the playouts towards the (seemingly) winning moves when they are first discovered. Two general method of doing so are the MAST (see Section 4.1.1) and RAVE (Gelly & Silver, 2007) techniques, but much bigger improvements could be made if these moves could be identified when they are first encountered and from then on completely ignored. MCTS solvers can also detect optimistic moves, but only in endgame scenarios.

7.3 Summary

In this chapter we ran experiments to gain insight into game tree properties that influence MCTS performance. We found that it depends on the game itself whether MCTS prefers deep trees, big branching factor, or a balance between the two. Apparently small nuances in game rules and scoring systems may alter the preferred game-tree structure. Consequently it is hard to generalize much about MCTS performance based on game tree depth and width. Progression is important to MCTS. However, our results suggests that MCTS may also be applied successfully in slowly progressing games, as long as a relatively small percentage of the simulations provides useful outcomes. In GGP games one could potentially take advantage of this low ratio by curtailing potentially fruitless simulations early, thus increasing simulation throughput. Hints of MCTS having difficulty in committing to a strategy when faced with many good ones were also discovered. Optimistic Moves are a real problem for MCTS that escalates with an increased branching factor.

Chapter 8

Related Work

8.1 General Game Playing Agents

One of the first general game-playing systems was Pell's METAGAMER (Pell, 1996), which played a wide variety of simplified chess-like games, but the introduction of the AAAI GGP competition (Genesereth et al., 2005) really brought about an ongoing interest in general game-playing systems.

CLUNEPLAYER (Clune, 2007) and FLUXPLAYER (Schiffel & Thielscher, 2007) were the winners of the 2005 and 2006 GGP competitions, respectively. CADIAPLAYER won the competition in 2007 and 2008, and the agent ARY (Méhat & Cazenave, 2010a) won in 2009 and 2010. Turbo Turtle (Sam Schreiber, Independent Researcher) proved victorious in the 2011 competition. An overview of all GGP Competition results can be found in Appendix D. The first two agents employed traditional game-tree search, whereas the all the later winners are MCTS based.

The CLUNEPLAYER (Clune, 2007) agent creates abstract models from the game descriptions that incorporate essential aspects of the original game, such as payoff, control, and termination. The agent then identifies stable features through sampling, which are then used for fitting the models using regression. Since winning the GGP competition in 2005, the agent has finished second in all subsequent GGP competitions until its retirement in 2008. In the 2008 competition the agent had dual capabilities such that it could choose between using either mimimax-based or Monte-Carlo simulation search, based on game properties.

The FLUXPLAYER agent (Schiffel & Thielscher, 2007; Haufe, Michulke, Schiffel, & Thielscher, 2011) uses fluent calculus (an extension of situated calculus) for reasoning

about actions and for generating game states. Standard game-tree search techniques are used for the planning phase, including non-uniform depth-first search, iterative deepening, transposition tables, and history heuristic. The heuristic function evaluation is based on fuzzy logic where semantic properties of game predicates are used for detecting static structures in the game descriptions. In addition to winning the 2006 GGP competition, this system has frequently been at the top and is still competing.

The ARY agent (Méhat & Cazenave, 2010a) uses MCTS and was developed at the Paris 8 University. It has successfully used root parallelization (Chaslot et al., 2008) to increase its playing strength (Méhat & Cazenave, 2011; Méhat & Cazenave, 2011). ARY has also been used to research single player MCTS in GGP with regards to the use of transposition tables and *Nested Monte-Carlo Search* (Méhat & Cazenave, 2010b).

The GAMER agent (Kissmann & Edelkamp, 2011) instantiates the game rules upon receiving them and proceeds with trying to solve the game (Kissmann & Edelkamp, 2010). Their solver is based on symbolic search using *Binary Decision Diagrams* solving in a backwards direction and can handle single- and two-player turn-taking games. As this approach can be time consuming, depending on the game, GAMER runs a UCT/MCTS player until the solver reaches the current game state. From there on the agent plays optimally.

The CENTURIO agent (Möller, Schneider, Wegner, & Schaub, 2011) uses MCTS, but with different strategies depending on if the game type is single-player, turn-taking or simultaneous move multiplayer. Additionally *Answer Set Programming* is also used with single-player games. This agent can translate GDL into Java code using techniques in (Waugh, 2009) adapted to Java. CENTURIO supports both thread parallelizing a single MCTS tree on a multicore computer and multiple MCTS trees over a cluster of computers.

8.2 Work on General Game Playing

In (Banerjee, Kuhlmann, & Stone, 2006) it is shown how knowledge transfer can be obtained with games of the same genre when using reinforcement learning. First a small game that can be learned quickly and captures the genre is selected. Then a number of features for it are identified and handcrafted. In this case the features are recognized from the structure of the game tree, but can be expanded to other types of feature recognition. After learning the small game, each feature F_i is assigned the average value of the $\mathcal{Q}(s, a)$ values where $F_i \in (s, a)$. Now, with the aid of these feature values we can initialize states

in other games of the same genre. When a state-action pair is initialized we examine which features it contains and use the highest value of these features. This way the value function of the new game does not have to start from scratch. Two of the authors, Gregory Kuhlmann and Peter Stone, participated in the first three GGP competitions with their player UTEXAS LARG (G. Kuhlmann, Dresner, & Stone, 2006). To read more on the subject of knowledge transfer we refer to (Sherstov & Stone, 2005), (Taylor, Whiteson, & Stone, 2006), (Banerjee & Stone, 2007) and (G. J. Kuhlmann, 2010).

Search control in GGP has also been studied in (Sharma, Kobti, & Goodwin, 2008) where states and move patterns are used to generate domain-independent knowledge, and in (Kirci, Schaeffer, & Sturtevant, 2009) differences in state predicates are combined with the causing actions to detect both offensive and defensive features. Automatic generation of state evaluation function has also been addressed in (Michulke, 2011) which aims at mitigating the problems of expensive feature generation and cost of learning weights by use of the GDL *Goal* function. A co-evolution approach that allows algorithm designers to both minimize the amount of domain knowledge built into the system and model opponent strategies more efficiently (Reisinger, Bahceci, Karpov, & Miikkulainen, 2007) and a logic program approach where the game descriptions are translated into specialized evaluator that works with decomposed logical features for improving accuracy and efficiency (Kaneko, Yamaguchi, & Kawai, 2001).

Other important GGP research topics include: representations (G. Kuhlmann & Stone, 2007), efficient GDL reasoning engines (Waugh, 2009; Saffidine & Cazenave, 2011), detecting state-space properties such as symmetries (Schiffel, 2010) and factorability (Cox, Schkufza, Madsen, & Genesereth, 2009; Günther, Schiffel, & Thielscher, 2009), proving game properties (Schiffel & Thielscher, 2009; Thielscher & Voigt, 2010), and automatic opening book generation (Chaslot et al., 2009). There is also interest in extending the expressive power of GDL (Love et al., 2008) to what had been dubbed GDL-II (Thielscher, 2010), adding support for both non-deterministic and incomplete information games.

We would also like to refer the reader to (Thielscher, 2011), which is an overview of GGP research, for more detail.

8.3 Monte-Carlo Tree Search

Monte Carlo Tree Search (MCTS) was pioneered in computer *Go*, and is now used by several of the strongest *Go* programs, including MOGO (Gelly et al., 2006),CRAZYSTONE

(Coulom, 2006), and FUEGO (Enzenberger & Müller, 2009). Nowadays many other game programs have also adopted MCTS for games such as *Amazons* (Lorentz, 2008), *Lines-of-Action* (Winands et al., 2010), *Chinese Checkers* (Sturtevant, 2008), *Kriegspiel* (Ciancarini & Favini, 2009), and *Settlers of Catan* (Szita et al., 2009).

Experiments in *Go* showing how simulations can benefit from using an informed playout policy are presented in (Gelly & Silver, 2007). This, however, requires game-specific knowledge which makes it difficult to apply in GGP. The paper also introduced RAVE. A RAVE variant called *poolRave* (Rimmel, Teytaud, & Teytaud, 2010) for the playout phase of MCTS has been shown to have some general benefits as it is applicable in both the game *Go* and *Havannah*. Progressive Strategies (Chaslot, Winands, Herik, Uiterwijk, & Bouzy, 2007) are also used by Go programs to improve simulation guidance in the MCTS's selection step.

Comparison between *Monte-Carlo* and *AlphaBeta* methods was done in (Clune, 2008). There the author conjectures that *AlphaBeta* methods do best compared to MCTS when: (1) The heuristic evaluation function is both stable and accurate, (2) The game is two-player, (3) The game is turn-taking, (4) The game is zero-sum and (5) The branching factor is relatively low. Experiments using both real and randomly generated synthetic games are then administered to show that the further you deviate from these settings the better *Monte-Carlo* does in relation to *AlphaBeta*.

Early Cutoff has been used before in MCTS game playing agents in an effort to get better information through more simulations. INVADERMC (Lorentz, 2008) is an MCTS agent made for the game *Amazons* that utilizes this kind of extension. It terminates a simulation when it has reached a fixed number of playout-steps, returning a heuristic evaluation of the reached state. This extension is also used in the game *Lines of Action* (Winands & Björnsson, 2009), where it has the more appropriate name *Evaluation Cut-offs*. There the cutoff is not made until the evaluation function has reached a certain score instead of having the length of the simulation dictate when to cut. However, both approaches rely on pre-coded evaluation knowledge. The *Unexplored Action Urgency* extension resembles extensions that include domain specific knowledge in selecting between actions in the playout phase. The notion of move urgency was used in (Bouzy, 2005) when playing *Go*. There knowledge about *Go* is used to bias the selection probability distribution of playout moves from being uniform to reflect what is known, that is moves that are known to be good become more urgent. Move urgency is widely used in *Go* programs. The *Unexplored Action Urgency* is similar to the *First Play Urgency* (FPU) in MOGO (Gelly et al., 2006), except there the unexplored urgency value is a constant. We extended early cutoffs and action urgency to also be applicable in GGP.

An MCTS solver for the game *Lines of Action* was introduced in (Winands et al., 2008). *Lines of Action* is turn-taking and has only win or loss outcomes. A generalized version of the MCTS solver was presented under the name *Score Bounded MCTS* in (Cazenave & Saffidine, 2011) that is applicable to any outcome turn-taking zero-sum game. This method keeps track of *Pessimistic* and *Optimistic* bounds (Berliner, 1979) for each tree node, updated based on the same type of bounds in the child nodes, so that pruning may be done if the bounds cross before the actual *MiniMax* value of the node is discovered. *Score Bounded MCTS* solver has been applied to the simultaneous game *Tron* (Teuling, 2011), but the simultaneous aspect is bypassed by treating the game as a turn-taking one.

In (Ramanujan, Sabharwal, & Selman, 2010) the authors identify *Shallow Traps*, i.e. when MCTS agent fails to realize that taking a certain action leads to a winning strategy for the opponent. Instead of this action getting a low ranking score, it looks like being close to or even as good as the best action available. The paper examines MCTS behavior faced with such traps 1, 3, 5 and 7 plies away. We believe there is some overlapping between our Optimistic Moves and these Shallow Traps.

MCTS performance in imperfect information games is studied in (Long, Sturtevant, Buro, & Furtak, 2010). For their experiments the authors use game trees where they can tune three properties: (1) *Leaf Correlation* - the probability of all siblings of a terminal node having the same payoff value, (2) *Bias* - the probability of one player winning the other and (3) *Disambiguation factor* - how quickly the information sets shrink. They then show how any combination of these three properties affect the strength of MCTS.

Chapter 9

Conclusions and Future Work

In this thesis we have taken a close look at MCTS in the context of GGP. MCTS has over recent years proven to be most resilient in many domains. The combination of the two produced our successful and trendsetting GGP agent CADIAPLAYER, the subject of Chapter 3, which has been at the forefront of the GGP competition since its debut in 2007.

9.1 Summary of Results

One of the most essential part of simulation-based GGP agents is the search-control methods they use for guiding the simulations. Forming a strategy in real-time having only the game description is challenging, but in this task lie the potentials of generating intelligent behavior. It seems clear that integrating knowledge into the search process is necessary to achieve an expert level of play. This task may be approached using simulation-based methods, giving access to a world of statistical data or we may try to impose meaning onto the rules of the game. Even though the problem of finding useful data is achieved for a certain game, not all data is applicable to every game and can even be damaging in some cases. In competition settings, as in any real-time use of the GGP agents, robustness is vital to the success of this challenge and must be taken into account. In Chapter 4 we introduced several method to address this task, utilizing both statistical domain-independent and game-specific approached to improve the playing strength of CADIAPLAYER while making sure robustness is maintained, showing that knowledge truly is power. These methods form the state-of-the-art in simulation search-control in GGP.

Nowadays, game-specific programs have embraced MCTS and while adapting it to their game, some have come up with beneficial extensions. These advances are, however, more often than not targeted towards something very problem-specific and often rely heavily on some domain-dependent knowledge. This makes many such extensions hard to directly apply to GGP. In Chapter 5 we generalize two such extensions to be applicable in GGP, which when applied to CADIAPLAYER, improve its playing strength by a significant margin.

Recently methods that allow MCTS to deploy a general solver have started to surface. In Chapter 6 we add to the evolution of such solvers by extending the current two-player turn-taking zero-sum method to also include simultaneous move games.

What makes MCTS outperform the more traditional search methods in some domains and not others is still a relatively open question. Is there any way to predict the effectiveness of MCTS and if so can we bias some aspect or property to our advantage? In order to come up with methods that better control MCTS it seems necessary to understand what influences its behavior in game trees. In Chapter 7 we pinpoint three properties common in MCTS and through extensive experiments were able to give insight into how much and at what level they affect MCTS to guide further development.

9.2 Future Work

For future work there are still many interesting research questions to explore for further improving simulation-based search control in GGP.

- There is some specific work that can be done to further improve the schemes discussed, for example, improving PAST so that it reaches a competition level of robustness. Also, we do not fully understand which game properties determine why MAST is so much better than TO-MAST on certain games but clearly worse on others.
- FAST is just the beginning for the research needed on how best to incorporate higher-level game concepts like material and captures into simulation control, e.g. to understand a much broader range of games and game concepts. We believe that to take GGP simulation-based agents to the next level such reasoning is essential.
- The combination of FAST with MAST was just an initial step in that direction and needs to be better explored with regards to how their application overlaps in some

games, so measures can be taken against decreasing the performance boost of the better suited method.

- The MCTS simultaneous moves solver has still some way to go. Another research task regarding the solver is to examine the feasibility of dropping the zero-sum constraint by treating each of the bounds as a bimatrix game, for which a Nash value can be calculated with known algorithms.
- Regarding the game tree properties in Chapter 7 methods may be devised for MCTS that help identifying these properties on the fly and take measures that either exploit or counteract what is discovered. As of now we have only used the insight into slow game progression to motivate the design of the *Early Cutoff* extension in Chapter 5. This could be in the form of new extensions, pruning techniques or even parameter tuning of known extension. Also more research needs to be done regarding the possible MCTS strategy commitment issues.

Bibliography

Abramson, B. (1990). Expected-Outcome: A General Model of Static Evaluation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *12*, 182–193.

Banerjee, B., Kuhlmann, G., & Stone, P. (2006). Value Function Transfer for General Game Playing. In *ICML Workshop on Structural Knowledge Transfer for ML.*

Banerjee, B., & Stone, P. (2007). General Game Learning using Knowledge Transfer. In *The 20th International Joint Conference on Artificial Intelligence* (pp. 672–677).

Berliner, H. (1979). The B* Tree Search Algorithm: A Best-First Proof Procedure. *Artificial Intelligence*, *12*(1), 23–40.

Björnsson, Y., Bulitko, V., & Sturtevant, N. (2009). TBA*: Time-bounded A*. In *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009.*

Björnsson, Y., & Finnsson, H. (2009). CADIAPlayer: A Simulation-Based General Game Player. *IEEE Transactions on Computational Intelligence and AI in Games*, *1*(1), 4–15.

Bouzy, B. (2005). Associating domain-dependent knowledge and Monte Carlo approaches within a Go program. *Information Sciences*, *Heuristic Search and Computer Game Playing IV, 175*(4), 247–257.

Bouzy, B., & Helmstetter, B. (2003). Monte-carlo go developments. In H. J. van den Herik, H. Iida, & E. A. Heinz (Eds.), *Acg* (Vol. 263, pp. 159–174). Kluwer.

Brügmann, B. (1993). *Monte Carlo Go* (Tech. Rep.). Max Planck Institute of Physics.

Buro, M. (1999). How Machines Have Learned to Play Othello. *IEEE Intelligent Systems*, *14*(6), 12–14. (Research Note)

Campbell, M., Hoane, Jr., A. J., & Hsu, F.-H. (2002). Deep Blue. *Artificial Intelligence*, *134*(1–2), 57–83.

Cazenave, T. (2009). Nested Monte-Carlo Search. In C. Boutilier (Ed.), *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009* (pp. 456–461).

Cazenave, T., & Jouandeau, N. (2007). On the Parallelization of UCT. In *Proc. of the Computer Games Workshop (CGW2007)* (pp. 93–101).

Cazenave, T., & Jouandeau, N. (2008). A Parallel Monte-Carlo Tree Search Algorithm. In *Computers and Games* (pp. 72–80).

Cazenave, T., & Saffidine, A. (2011). Score Bounded Monte-Carlo Tree Search. In H. van den Herik, H. Iida, & A. Plaat (Eds.), *Computers and Games* (Vol. 6515, pp. 93–104). Springer Berlin / Heidelberg.

Chaslot, G. (2010). *Monte-Carlo Tree Search*. PhD Dissertation, Maastricht University, The Netherlands, Department of Knowledge Engineering.

Chaslot, G., Hoock, J.-B., Perez, J., Rimmel, A., Teytaud, O., & Winands, M. H. M. (2009). Meta Monte-Carlo Tree Search for Automatic Opening Book Generation. In *Proceedings of the IJCAI-09 Workshop on General Game Playing (GIGA'09).*

Chaslot, G., Winands, M. H. M., & Herik, H. J. van den. (2008). Parallel Monte-Carlo Tree Search. In *Computers and Games* (pp. 60–71).

Chaslot, G., Winands, M. H. M., Herik, H. J. van den, Uiterwijk, J., & Bouzy, B. (2007). Progressive Strategies for Monte-Carlo Tree Search. In *Joint Conference on Information Sciences, Salt Lake City 2007, Heuristic Search and Computer Game Playing Session.*

Ciancarini, P., & Favini, G. P. (2009). Monte Carlo tree search techniques in the game of Kriegspiel. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence* (pp. 474–479). San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.

Clune, J. E. (2007). Heuristic Evaluation Functions for General Game Playing. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, AAAI 2007, Vancouver, British Columbia, Canada, July 22-26, 2007* (pp. 1134–1139).

Clune, J. E. (2008). *Heuristic Evaluation Functions for General Game Playing*. PhD Dissertation, University of California, Los Angeles, Department of Computer Science.

Coulom, R. (2006). Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In *The 5th International Conference on Computers and Games (CG2006)* (pp. 72–83).

Cox, E., Schkufza, E., Madsen, R., & Genesereth, M. R. (2009). Factoring general games using propositional automata. In *Proceedings of the IJCAI-09 Workshop on General Game Playing (GIGA'09).*

Dresden GGP Server. (n.d.). Dresden GGP server Web site: http://euklid.inf.tu-dresden.de:8180/ggpserver.

Enzenberger, M., & Müller, M. (2009). *Fuego - an open-source framework for board games and Go engine based on Monte-Carlo tree search* (Tech. Rep. No. 09-08). Department of Computing Science, University of Alberta.

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

Finnsson, H. (2012). Generalizing Monte-Carlo Tree Search Extensions for General Game Playing. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence (Accepted), AAAI 2012, Toronto, Ontario, Canada, July 22-26, 2012*. AAAI Press.

Finnsson, H., & Björnsson, Y. (2008). Simulation-Based Approach to General Game Playing. In D. Fox & C. P. Gomes (Eds.), *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008* (pp. 259–264). AAAI Press.

Finnsson, H., & Björnsson, Y. (2009). Simulation Control in General Game Playing Agents. In *GIGA'09 The IJCAI Workshop on General Game Playing*.

Finnsson, H., & Björnsson, Y. (2010). Learning Simulation Control in General Game-Playing Agents. In M. Fox & D. Poole (Eds.), *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010* (pp. 954–959). AAAI Press.

Finnsson, H., & Björnsson, Y. (2011a). CadiaPlayer: Search-Control Techniques. *KI - Künstliche Intelligenz*, *25*(1), 9–16.

Finnsson, H., & Björnsson, Y. (2011b). Game-Tree Properties and MCTS Performance. In *GIGA'11 The IJCAI Workshop on General Game Playing*.

flex: The Fast Lexical Analyzer. (n.d.). The Flex Web site: http://flex.sourceforge.net/.

Fox, M., & Poole, D. (Eds.). (2010). *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*. AAAI Press.

Gelly, S., & Silver, D. (2007). Combining online and offline knowledge in UCT. In Z. Ghahramani (Ed.), *Proceedings of the 2007 International Conference on Machine Learning* (Vol. 227, pp. 273–280). ACM.

Gelly, S., Wang, Y., Munos, R., & Teytaud, O. (2006). *Modification of UCT with patterns in Monte-Carlo Go* (Technical Report No. 6062). INRIA.

Genesereth, M. R., & Fikes, R. E. (1992). *Knowledge Interchange Format, Version 3.0 Reference Manual* (Tech. Rep. No. Technical Report Logic-92-1). Stanford University.

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

Guðmundsson, G. Þ. (2009). *Solving general game playing puzzles using heuristics search*. Unpublished master's thesis, Reykjavík University.

Günther, M., Schiffel, S., & Thielscher, M. (2009). Factoring General games. In *Proceedings of the IJCAI-09 Workshop on General Game Playing (GIGA'09).*

Haufe, S., Michulke, D., Schiffel, S., & Thielscher, M. (2011). Knowledge-Based General Game Playing. *KI - Künstliche Intelligenz*, *25*(1), 25–33.

Herik, H. J. van den, & Spronck, P. (Eds.). (2010). *Advances in Computer Games, 12th International Conference, ACG 2009, Pamplona, Spain, May 11-13, 2009. Revised Papers.* Springer.

Hoffmann, J., & Nebel, B. (2001). Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, *14*, 253–302.

ICC Help: PGN-spec. (n.d.). The Internet Chess Club Web site: http://www.chessclub.com/help/PGN-spec.

Kaneko, T., Yamaguchi, K., & Kawai, S. (2001). Automatic Feature Construction and Optimization for General Game Player. In *Proceedings of Game Programming Workshop 2001 (GPW2001)* (pp. 25–32).

Kirci, M., Schaeffer, J., & Sturtevant, N. (2009). Feature Learning Using State Differences. In *GIGA'09 The IJCAI Workshop on General Game Playing.*

Kissmann, P., & Edelkamp, S. (2010). Instantiating general games using prolog or dependency graphs. In *Proceedings of the 33rd annual German conference on Advances in artificial intelligence* (pp. 255–262). Berlin, Heidelberg: Springer-Verlag.

Kissmann, P., & Edelkamp, S. (2011). Gamer, a General Game Playing Agent. *KI - Künstliche Intelligenz*, *25*(1), 49–52.

Knuth, D. E., & Moore, R. W. (1975). An Analysis of Alpha-beta Pruning. *Artificial Intelligence*, *6*(4), 293–326.

Kocsis, L., & Szepesvári, C. (2006). Bandit Based Monte-Carlo Planning. In *European Conference on Machine Learning (ECML)* (pp. 282–293).

Kuhlmann, G., Dresner, K., & Stone, P. (2006). Automatic Heuristic Construction in a Complete General Game Player. In *Proceedings of the Twenty-first AAAI Conference on Artificial Intelligence* (pp. 1457–1462).

Kuhlmann, G., & Stone, P. (2007). Graph-Based Domain Mapping for Transfer Learning in General Games. In *Proceedings of The Eighteenth European Conference on Machine Learning.*

Kuhlmann, G. J. (2010). *Automated Domain Analysis and Transfer Learning in General Game Playing*. Unpublished doctoral dissertation, University of Texas at Austin.

Long, J. R., Sturtevant, N. R., Buro, M., & Furtak, T. (2010). Understanding the Success of Perfect Information Monte Carlo Sampling in Game Tree Search. In M. Fox &

D. Poole (Eds.), *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010.* AAAI Press.

Lorentz, R. J. (2008). Amazons discover Monte-Carlo. In *Proceedings of the 6th international conference on Computers and Games* (pp. 13–24). Berlin, Heidelberg: Springer-Verlag.

Love, N., Hinrichs, T., Haley, D., Schkufza, E., & Genesereth, M. (2008). *General Game Playing: Game Description Language Specification* (Technical Report No. March 4 2008). Stanford University.

Marsland, T. A. (1983). Relative Efficiency of Alpha-Beta Implementations. In *Ijcai* (pp. 763–766).

Méhat, J., & Cazenave, T. (2010a). Ary, a general game playing program. In *Board Games Studies Colloquium.*

Méhat, J., & Cazenave, T. (2010b). Combining UCT and Nested Monte Carlo Search for Single-Player General Game Playing. *IEEE Transactions on Computational Intelligence and AI in Games*, *2*(4), 271–277.

Méhat, J., & Cazenave, T. (2011). A Parallel General Game Player. *KI - Künstliche Intelligenz*, *25*, 43–47. (10.1007/s13218-010-0083-6)

Méhat, J., & Cazenave, T. (2011). Tree Parallelization of Ary on a Cluster. In *GIGA'11 The IJCAI Workshop on General Game Playing*.

Michulke, D. (2011). Heuristic Interpretation of Predicate Logic Expressions in General Game Playing. In *Proceedings of the IJCAI-11 Workshop on General Game Playing (GIGA'11).*

Möller, M., Schneider, M., Wegner, M., & Schaub, T. (2011). Centurio, a General Game Player: Parallel, Java- and ASP-based. *KI - Künstliche Intelligenz*, *25*(1), 17–24.

Neumann, J. V., & Morgenstern, O. (1944). *Theory of Games and Economic Behavior*. Princeton University Press.

Pell, B. (1996). A Strategic Metagame Player for General Chess-Like Games. *Computational Intelligence*, *12*, 177–198.

Pitrat, J. (1968). Realization of a general game-playing program. In *IFIP Congress (2)* (pp. 1570–1574).

Ramanujan, R., Sabharwal, A., & Selman, B. (2010). On Adversarial Search Spaces and Sampling-Based Planning. In *ICAPS'10* (pp. 242–245).

Reinefeld, A. (1983). An Improvement to the Scout Tree-Search Algorithm. *International Computer Chess Association Journal*, *6*(4), 4–14.

Reinefeld, A., & Marsland, T. A. (1994). Enhanced Iterative-Deepening Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *16*(7), 701–710.

Reisinger, J., Bahceci, E., Karpov, I., & Miikkulainen, R. (2007). Coevolving Strategies for General Game Playing. In *IEEE Symposium on Computational Intelligence and Games* (pp. 320–327).

Rimmel, A., Teytaud, F., & Teytaud, O. (2010, May). Biasing Monte-Carlo Simulations through RAVE Values. In *The International Conference on Computers and Games 2010.* Kanazawa, Japan.

Rosin, C. D. (2011). Nested Rollout Policy Adaptation for Monte Carlo Tree Search. In T. Walsh (Ed.), *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011* (pp. 649–654). IJCAI/AAAI.

Russell, S. J., & Norvig, P. (2010). *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. Pearson Education.

Saffidine, A., & Cazenave, T. (2011). A Forward Chaining Based Game Description Language Compiler. In *Proceedings of the IJCAI-11 Workshop on General Game Playing (GIGA'11).*

Saffidine, A., Finnsson, H., & Buro, M. (2012). Alpha-Beta Pruning for Games with Simultaneous Moves. In *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence (Accepted), AAAI 2012, Toronto, Ontario, Canada, July 22-26, 2012.* AAAI Press.

Schaeffer, J. (1989). The History Heuristic and Alpha-Beta Search Enhancements in Practice. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, *PAMI-11*(11), 1203–1212.

Schaeffer, J. (1997). *One Jump Ahead: Challenging Human Supremacy in Checkers*. Springer-Verlag New York, Inc.

Schiffel, S. (2010). Symmetry Detection in General Game Playing. In M. Fox & D. Poole (Eds.), *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010.* AAAI Press.

Schiffel, S., & Thielscher, M. (2007). Fluxplayer: A Successful General Game Player. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence, AAAI 2007, Vancouver, British Columbia, Canada, July 22-26, 2007* (pp. 1191–1196).

Schiffel, S., & Thielscher, M. (2009). Automated Theorem Proving for General Game Playing. In C. Boutilier (Ed.), *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009* (pp. 911–916).

Schiffel, S., & Thielscher, M. (2011). Reasoning About General Games Described in GDL-II. In *Proceedings of the AAAI Conference on Artificial Intelligence* (pp.

846–851). San Francisco: AAAI Press.

Sharma, S., Kobti, Z., & Goodwin, S. (2008). Knowledge Generation for Improving Simulations in UCT for General Game Playing. In *AI 2008: Advances in Artificial Intelligence* (pp. 49–55). Springer.

Sherstov, A. A., & Stone, P. (2005). Improving Action Selection in MDP's via Knowledge Transfer. In *Proceedings of the Twentieth National Conference on Artificial Intelligence* (pp. 1024–1029).

Sturtevant, N. R. (2005). Leaf-Value Tables for Pruning Non-Zero-Sum Games. In *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30-August 5, 2005* (pp. 317–323). Professional Book Center.

Sturtevant, N. R. (2008). An Analysis of UCT in Multi-player Games. In H. J. van den Herik, X. Xu, Z. Ma, & M. H. M. Winands (Eds.), *Computers and Games, 6th International Conference, CG 2008, Beijing, China, September 29 - October 1, 2008. Proceedings* (Vol. 5131, pp. 37–49). Springer.

Sturtevant, N. R., & Korf, R. E. (2000). On Pruning Techniques for Multi-Player Games. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on on Innovative Applications of Artificial Intelligence* (pp. 201–207).

Sutton, R. S. (1988). Learning to Predict by the Methods of Temporal Differences. *Machine Learning*, *3*, 9–44.

Szita, I., Chaslot, G., & Spronck, P. (2009). Monte-Carlo Tree Search in Settlers of Catan. In H. J. van den Herik & P. Spronck (Eds.), *Advances in Computers and Games* (Vol. 6048, pp. 21–32). Springer.

Taylor, M. E., Whiteson, S., & Stone, P. (2006). Transfer Learning for Policy Search Methods. In *ICML workshop on Structural Knowledge Transfer for Machine Learning.*

Teuling, N. G. P. D. (2011). *Monte-Carlo Tree Search for the Simultaneous Move Game Tron* (Technical Report). University of Maastricht, Netherlands.

Thielscher, M. (2010). A General Game Description Language for Incomplete Information Games. In *Proceedings of the AAAI Conference on Artificial Intelligence* (pp. 994–999). Atlanta: AAAI Press.

Thielscher, M. (2011). General Game Playing in AI Research and Education. In J. Bach & S. Edelkamp (Eds.), *Proceedings of the German Annual Conference on Artificial Intelligence (KI)* (Vol. 7006, pp. 26–37). Berlin, Germany: Springer.

Thielscher, M., & Voigt, S. (2010). A Temporal Proof System for General Game Playing. In M. Fox & D. Poole (Eds.), *AAAI.* AAAI Press.

Waugh, K. (2009). Faster State Manipulation in General Games using Generated Code. In *Proceedings of the IJCAI-09 Workshop on General Game Playing (GIGA'09).*

Winands, M. H. M., & Björnsson, Y. (2009). Evaluation Function Based Monte-Carlo LOA. In H. J. van den Herik & P. Spronck (Eds.), *Advances in Computers and Games* (Vol. 6048). Springer. (33–44)

Winands, M. H. M., Björnsson, Y., & Saito, J.-T. (2008). Monte-Carlo tree search solver. In *Proceedings of the 6th international conference on Computers and Games* (pp. 25–36). Berlin, Heidelberg: Springer-Verlag.

Winands, M. H. M., Björnsson, Y., & Saito, J.-T. (2010). Monte Carlo Tree Search in Lines of Action. *IEEE Transactions on Computational Intelligence and AI in Games*, *2*(4), 239–250.

YAP Prolog. (n.d.). YAP Prolog Web site: http://www.ncc.up.pt/ vsc/Yap.

Zobrist, A. L. (1970). *A New Hashing Method With Application for Game Playing* (Technical Report No. TR88). Madison: University of Wisconsin.

Appendix A

GDL for Tic-Tac-Toe

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; Tictactoe
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Roles
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

  (role xplayer)
  (role oplayer)

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

  (init (cell 1 1 b))
  (init (cell 1 2 b))
  (init (cell 1 3 b))
  (init (cell 2 1 b))
  (init (cell 2 2 b))
  (init (cell 2 3 b))
  (init (cell 3 1 b))
  (init (cell 3 2 b))
  (init (cell 3 3 b))
  (init (control xplayer))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Dynamic Components
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Cell

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

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

```
      (true (cell ?m ?n b)))

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

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

  (<= (next (control xplayer))
      (true (control oplayer)))

  (<= (next (control oplayer))
      (true (control xplayer)))


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

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

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

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


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


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

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

  (<= (legal ?w (mark ?x ?y))
      (true (cell ?x ?y b))
      (true (control ?w)))

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

  (<= (legal oplayer noop)
```

```
      (true (control xplayer)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

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

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

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

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

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

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

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

  (<= terminal
      (line x))

  (<= terminal
      (line o))

  (<= terminal
      (not open))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

Appendix B

Prolog GGP Functions

```
:- use_module( library( lists ) ).
:- use_module( library( ordsets ) ).

distinct( _x, _y ) :- _x \= _y.

or( _x, _y ) :- _x ; _y.
or( _x, _y, _z ) :- _x ; _y ; _z.
or( _x, _y, _z, _w ) :- _x ; _y ; _z ; _w.
or( _x, _y, _z, _w, _v ) :-
    _x ; _y ; _z ; _w ; _v.
or( _x, _y, _z, _w, _v, _q ) :-
    _x ; _y ; _z ; _w ; _v ; _q.
or( _x, _y, _z, _w, _v, _q, _r ) :-
    _x ; _y ; _z ; _w ; _v ; _q ; _r.
or( _x, _y, _z, _w, _v, _q, _r, _s ) :-
    _x ; _y ; _z ; _w ; _v ; _q ; _r ; _s.
or( _x, _y, _z, _w, _v, _q, _r, _s, _t ) :-
    _x ; _y ; _z ; _w ; _v ; _q ; _r ; _s ; _t.
or( _x, _y, _z, _w, _v, _q, _r, _s, _t, _u ) :-
    _x ; _y ; _z ; _w ; _v ; _q ; _r ; _s ; _t ; _u.

:- dynamic state/1.

add_state_clauses( [] ).

add_state_clauses( [_x | _l] ) :-
    assert( state( _x ) ),
    add_state_clauses( _l ).

add_does_clauses( [] ).

add_does_clauses( [ [_p,_m] | _l] ) :-
    assert( does( _p, _m ) ),
    add_does_clauses( _l ).

add_does_clause( _p, _m ) :-
    assert( does( _p, _m ) ).
```

```
get_does_clause(_p, _m) :-
does(_p, _m).

state_init( _r ) :-
    role( _r ).

state_retract_info( _l ) :-
    bagof( _c, state( _c ), _l ).

state_gen_moves( _p, _m ) :-
    legal( _p, _m ).

state_make_move( _p, _m ) :-
    assert( does( _p, _m ) ),
    bagof( A, next( A ), _l ),
    retract( does( _p, _m ) ),
    retractall( state( _ ) ),
    add_state_clauses( _l ).

state_make_sim_moves( _ml ) :-
    add_does_clauses( _ml ),
    bagof( A, next( A ), _l ),
    retractall( does( _, _ ) ),
    retractall( state( _ ) ),
    remove_duplicates( _l, _ll ),
    add_state_clauses( _ll ).

state_make_rel_moves( _ml ) :-
    add_does_clauses( _ml ),
    bagof( A, next( A ), _l ),
    bagof( B, state( B ), _ll ),
    retractall( does( _, _ ) ),
    retractall( state( _ ) ),
    append(_l, _ll, _lll),
    remove_duplicates( _lll, _llll ),
    add_state_clauses( _llll ).

state_make_exist_moves :-
    bagof( A, next( A ), _l ),
    retractall( does( _, _ ) ),
    retractall( state( _ ) ),
    remove_duplicates( _l, _ll ),
    add_state_clauses( _ll ).

state_retract_move( _ri ) :-
    retractall( state( _ ) ),
    add_state_clauses( _ri ).

state_append_move( _ri ) :-
    add_state_clauses( _ri ).

state_remove_duplicates :-
    state_retract_info( _l ),
    remove_duplicates( _l, _ll ),
    retractall( state( _ ) ),
    add_state_clauses( _ll ).
```

```
state_peek_next( _ml, _sl ) :-
    add_does_clauses( _ml ),
    bagof( A, next( A ), _l ),
    retractall( does( _, _ ) ),
    remove_duplicates( _l, _sl ).

state_effects_plus( _ml, _ps ) :-
    state_peek_next( _ml, _nl ),
    list_to_ord_set( _nl, _ns ),
    state_retract_info( _cl ),
    list_to_ord_set( _cl, _cs ),
    ord_subtract( _ns, _cs, _ps ).

state_effects_minus( _ml, _ms ) :-
    state_peek_next( _ml, _nl ),
    list_to_ord_set( _nl, _ns ),
    state_retract_info( _cl ),
    list_to_ord_set( _cl , _cs ),
    not(state_get_has_seen( _hl )),
    ord_subtract( _cs, _ns, _ms ).

state_effects_minus( _ml, _ms ) :-
    state_peek_next( _ml, _nl ),
    list_to_ord_set( _nl, _ns ),
    state_retract_info( _cl ),
    list_to_ord_set( _cl , _cs ),
    state_get_has_seen( _hl ),
    list_to_ord_set( _hl, _hs ),
    ord_subtract( _cs, _ns, _ds ),
    ord_subtract( _ds, _hs, _ms ).

state_get_has_seen( _hl) :-
    bagof( has_seen( A ), state( has_seen( A ) ), _hl ).

state_is_terminal :-
    terminal.

state_goal( _p, _r ) :-
    goal( _p, _r ).

state_assert_clause( _x ) :-
    assert( state( _x ) ).

state_retract_clause( _x ) :-
    retract( state( _x ) ).

state_initialize :-
    bagof( A, state( A ), _l ),
    retractall( state( _ ) ),
    remove_duplicates( _l, _ll ),
    add_state_clauses( _ll ).

has_seen_function(state(X)) :-
    state(has_seen(X)).

has_seen_function(X) :-
    not(X).
```

Appendix C

Rules of Games used in Experiments

Following are descriptions of all the GGP games used in the experiments presented of this thesis.

3D Tic-Tac-Toe *(Turn-taking, 2-player)*:
3D Tic-Tac-Toe is a variant of Tic-Tac-Toe played on a $4 \times 4 \times 4$ cube with the objective of lining up four of your pieces in a straight line.

Battle *(Simultaneous move, 2-player)*:
Battle is a simultaneous move game played on an 8×8 board where each player has 20 disks along the edges of two adjacent sides that can move one square or capture an opponent disks next to them. Instead of moving, players can also opt for defending a square occupied by their piece. If a defending piece is attacked the attacker is captured. The goal is to be the first to capture 10 opponent disks.

Bidding Tic-Tac-Toe *(Simultaneous move, 2-player)*:
This is a variation of normal Tic-Tac-Toe where the there is a bidding round between normal play that decides who gets to place a maker on the board. Each player begins with three coins and the x player has an additional tiebreaker token. When a player wins a bidding round, allowing him to place a marker, all coins bid go to the opponent. It is up to the holder of the tiebreaker if he wants to play it or not, but just as with the coins, it changes hands when it is a part of a winning bid. The winning conditions are the same as in normal Tic-Tac-Toe.

Breakthrough *(Turn-taking, 2-player)*:
Breakthrough is played on a chess board where the players, armed with two ranks of pawns, try to be the first to break through to reach the opponent's back rank. The pawns can move forward and diagonally, but can only capture diagonally.

Checkers *(Turn-taking, 2-player)*:
Checkers is played on a 8×8 square board with alternating colored squares (like a chess board) and is played only on one of the colors. Each player has twelve pawns placed on the three ranks closest to him. The pieces move diagonally forward and capture by jumping over diagonally adjacent opponent and multiple jumps/captures are allowed in a single turn. If a pawn reaches the opposite end of the board it is turned into a king. The kings may also move and capture backwards. The player who captures all his opponent's pieces wins.

Chinese Checkers 3P *(Turn-taking, 3-player)*:
Chinese Checkers with three players is played on the traditional hexagram Chinese Checkers board. Each player starts in one corner such that the corner opposite each player is not occupied. The GGP version of Chinese Checkers has corners of size three and therefore each player has three pieces. The pieces can either move to an adjacent empty position or do chains of jumps over an adjacent own or opponent piece. There are no captures. The first player to completely populate the opposite corner of his starting corner, wins.

Chinook *(Simultaneous move, 2-player)*:
Chinook is a variant of Breakthrough played with checkers pawns and the added twist that two independent games are played simultaneously, one on the white squares and another on the black ones. Chinook is a games presented at the 2010 GGP competition and is a breakthrough type game using pieces that move like Checkers pawns. Chinook also has simultaneous moves such that White alternates moving pawns on white and black squares each ply starting with a pawn on a white square while Black moves simultaneously one of its pawns that reside on the opposite colored squares. The sided of the board are also connected making it cylindrical.

Connect 5 *(Turn-taking, 2-player)*:
Connect 5 is played on an 8×8 board where the players take turn in securing a square by placing their marker on it. the first one to line up five of their own markers in a straight line, diagonally included, wins.

Goofspiel *(Simultaneous move, 2-player)*:
In this thesis we use a few variations by changing the number of cards used. Lets use the five card variation as an example. Each player then gets five cards, ace to a five which

they may play in any which order they choose. On the table are also five cards with the top one face up. Each player now plays a card and the one with the higher card gets the point value of the card on the table and it is discarded. If both play the same card neither gets any points. When all cards have been played the one with more points wins.

Knightthrough *(Turn-taking, 2-player)*:
Knightthrough is the same as Breakthrough except it is played with chess knights that may only play knight-type moves that advance them on the board.

Othello *(Turn-taking, 2-player)*:
Othello (also known as Reversi) is played on an 8×8 board. There are 64 disks, having one side dark and the other light, to be played onto the board by both players. One player may only place disks with the dark side up and the other only with the light side up. The starting position has four disks in the middle of the board, two of each color connected diagonally. Each turn a player can only place a disk adjacent to another disk that has the opponents color if somewhere in a straight line (diagonally included) from those two disks respectively there is another disk with the player's color and no empty squares between them. Once a disk has been placed all the opponent's disk in between are turned over and therefore change color. If a player has nowhere to place he looses his turn. The winner is the one with more disks in his color when all 64 disks have been placed, or if all the disks on the boards become the same color.

Pawn Whopping *(Simultaneous move, 2-player)*:
This game is played on a normal chess board where all pieces except the pawns have been removed. The pawns move just like they would in chess. The participants play simultaneously and if opposing pawns try to capture each other or move to the same square, the moves on both sides are cancelled. The first one to reach the back ranks of the opponent wins.

Skirmish *(Turn-taking, 2-player)*:
Skirmish is a chess-like game. We used the variation played in the 2007 GGP finals where each player has two knights, two bishops, two rooks, and four pawns. The objective is to capture as many of the opponent's pieces as possible. In the version used in this thesis the pawns can only move by capturing, in addition the squares at *c3*, *c6*, *f3* and *f6* are off limits.

TCCC4 *(Turn-taking, 2-player)*:
TCCC4 is a hybrid of several games played on a 5×5 size board with three piece start squares on either side where each player has a chess pawn, a chess knight, and a checkers king — these pieces re-spawn on their start square if captured. Instead of moving a piece a player can choose to drop a disk onto the board like in Connect 4 (captured disks do not respawn). The goal is to form a 3-in-a-row formation with your pieces anywhere on the 3×3 center squares of the table as int Tic-Tac-Toe. TCCC4 stands for the games it is a mix of, **T**ic-Tac-Toe, **C**hess, **C**heckers, and **C**onnect**4**.

TCCC4 3P *(Turn-taking, 3-player)*:
Same as TCCC4 with an additional player at the bottom of the board.

Appendix D

GGP Competition Results

There have been different knock-out formats in the competition finals through the years and sometimes preliminaries were held. For the last years, then number of players competing has been between 10 and 20. The following is an overview of the results of the annual GGP competition. Due to availability of the data and competition formatting, we show as many of the top three places as we can.

GGP Competition 2005 - *Finals*

1.	CLUNEPLAYER	University of California, Los Angeles
2.	OGRE	Florida International University
3.	FLUXPLAYER	Technical University of Dresden

GGP Competition 2006 - *Finals*

1.	FLUXPLAYER	Technical University of Dresden
2.	CLUNEPLAYER	University of California, Los Angeles
3.	UT-AUSTIN-LARG	University of Texas, Austin

GGP Competition 2007 - *Preliminaries*

1.	CADIAPLAYER	Reykjavík University
2.	FLUXPLAYER	Technical University of Dresden
3.	ARY	University of Paris 8

GGP Competition 2007 - *Finals*

1.	CADIAPLAYER	Reykjavík University
2.	CLUNEPLAYER	University of California, Los Angeles

GGP Competition 2008 - *Preliminaries*

1.	CADIAPLAYER	Reykjavík University
2.	CLUNEPLAYER	University of California, LA
3.	ARY	University of Paris 8

GGP Competition 2008 - *Finals*

1.	CADIAPLAYER	Reykjavík University
2.	CLUNEPLAYER	University of California, Los Angeles

GGP Competition 2009 - *Preliminaries*

1.	CADIAPLAYER	Reykjavík University
2.	ARY	University of Paris 8
3.	TURBOTURTLE	Sam Schreiber, Independent

GGP Competition 2009 - *Finals*

1.	ARY	University of Paris 8
2.	FLUXPLAYER	Technical University of Dresden
3.	MALIGNE	University of Alberta, Canada

GGP Competition 2010 - *Finals*

1.	ARY	University of Paris 8
2.	MALIGNE	University of Alberta, Canada
3.	CADIAPLAYER	Reykjavík University

GGP Competition 2011 - *Finals*

1.	TURBOTURTLE	Sam Schreiber, Independent
2.	CADIAPLAYER	Reykjavík University
3.	ARY	University of Paris 8

School of Computer Science
Reykjavík University
Menntavegi 1
101 Reykjavík, Iceland
Tel. +354 599 6200
Fax +354 599 6201
www.reykjavikuniversity.is
ISSN 1670-8539