REYKJAVÍK UNIVERSITY
HÁSKÓLINN Í REYKJAVÍK

CADIA-Player:
A General Game Playing Agent

Hilmar Finnsson

Master of Science
December 2007

Reykjavík University - School of Computer Science

M.Sc. Thesis

CADIA-Player:
A General Game Playing Agent

by

Hilmar Finnsson

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

December 2007

Thesis Committee:

Dr. Yngvi Björnsson, supervisor
Associate Professor, Reykjavík University, Iceland

Dr. Martin Müller
Associate Professor, University of Alberta, Canada

Dr. Ari Kristinn Jónsson
Dean, Reykjavík University, Iceland

The undersigned hereby certify that they recommend to the School of Computer Science at Reykjavík University for acceptance this thesis entitled **CADIA-Player: A General Game Playing Agent** submitted by Hilmar Finnsson in partial fulfillment of the requirements for the degree of **Master of Science**.

_____

Date

_____

Dr. Yngvi Björnsson, supervisor
Associate Professor, Reykjavík University, Iceland

_____

Dr. Martin Müller
Associate Professor, University of Alberta, Canada

_____

Dr. Ari Kristinn Jónsson
Dean, Reykjavík University, Iceland

CADIA-Player:
A General Game Playing Agent

by

Hilmar Finnsson

December 2007

Abstract

The aim of *General Game Playing* (GGP) is to create intelligent agents that can automatically learn how to play many different games well without any human intervention, given only a description of the game rules. This forces the agents to be able to learn a strategy without having any domain-specific knowledge provided by their developers. The most successful GGP agents have so far been based on the traditional approach of using game-tree search augmented with an automatically learned evaluation function for encapsulating the domain-specific knowledge. In this thesis we describe CADIA-Player, a GGP agent that instead uses a simulation-based approach to reason about its actions. More specifically, it uses Monte Carlo rollouts with upper confidence bounds for trees (UCT) as its main search procedure. CADIA-Player has already proven the effectiveness of this simulation-based approach in the context of GGP by winning the Third Annual GGP Competition. We describe its implementation as well as several algorithmic improvements for making the simulations more effective. Empirical data is presented showing that CADIA-Player outperforms naïve Monte Carlo by close to 90% winning ratio on average on a wide range of games, including Checkers and Othello. We further investigate the relative importance of UCT's action-selection rule, its memory model, and the various enhancements in achieving this result.

CADIA-Player:
Alhliða Leikjaspilari

eftir

Hilmar Finnsson

Desember 2007

Útdráttur

Markmið Alhliða Leikjaspilunar (e. General Game Playing) er að búa til greind forrit sem ekki eru einskorðuð við einn leik, heldur fá sem inntak leikreglur og þurfa að geta lært að spila leikinn sem þær lýsa. Þetta neyðir forritið til að mynda herkænsku sína á eigin spýtur án þess að styðjast við upplýsingar um leikinn sem hönnuður þess hefur sett inn í það. Hingað til hafa þau forrit sem notið hafa mestrar velgengni í alhliða leikjaspilun notað hina hefðbundnu aðferð að leita í leiktrénu með sjálfvirkri uppgötvun gildisákvörðunarfalls til að hjúpa þekkingu út frá lýsingu leiksins. Í þessari ritgerð lýsum við CADIA-Player, alhliða leikjaspilara sem notar hermanir til að draga ályktanir um leiki. Nánar tiltekið notar hann Monte Carlo útspilun með UCT (Upper Confidence Bound fyrir tré) sem aðal leitaraðferð sína. CADIA-Player hefur þegar, með því að vinna þriðju árlegu keppni slíkra forrita sannað hversu áhrifaríkar aðferðir byggðar á hermun geta verið í alhliða leikjaspilun. Við lýsum útfærslu spilarans auk þess að sýna nokkrar betrumbætur á algríminu sem auka afköst þess. Niðurstöður tilrauna eru gefnar sem sýna að CADIA-Player hefur mikla yfirburði yfir einfaldan Monte Carlo spilara, eða rétt undir 90% vinningshlutfall að meðaltali í hinum ýmsu leikjum, þ.m.t. Checkers og Othello. Við rannsökum enn fremur tölfræðilegt mikilvægi þess hvernig UCT velur aðgerðir, minnislíkans hans og hinna ýmsu viðbóta við að ná þessum árangri.

To my grandmother Þóra Sigurðardóttir
and the countless hands of cards we have played.

Acknowledgements

Dr. Yngvi Björnsson my supervisor and co-creator of CADIA-Player. Without his input and help this project would not be what it is today.
Pálmi Skowronski a fellow students at the CADIA lab for proof reading parts of this thesis.
My family and friends, for their support through the years.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Many intelligent programs exist that are excellent game players and can defeat almost any, if not all, human challengers. But even though these programs display a high level of intelligence at the game they are programmed for, they have no chance of playing anything else. Even the most simple game is out of their league. Today, it can be said for AI gaming programs that they are good not at gaming, but good at a certain game. Humans, on the other hand, do not have these limitations and can learn to play new games when supplied with a set of rules for them and develop new strategies, sometimes even drawn from similarities between games.

Research into *General Game Playing* (GGP) aims at building intelligent software agents that can, given the rules of any game, automatically learn a strategy for playing that game at an expert level without any human intervention. Successful realization of this task poses many interesting research challenges for a wide variety of artificial intelligence sub-disciplines including: knowledge representation, agent-based reasoning, heuristic search, and machine learning. Substantial progress has been made towards this goal in the past few years, and to facilitate further research into this area the Stanford Logic Group started the *General Game Playing Project* (*General Game Playing Project*, n.d.) a few years ago, along with the annual GGP competition (Genesereth, Love, & Pell, 2005). In part because of this initiative GGP has received a renewed interest from the artificial intelligence community, and currently there are several prominent research groups world-wide working on developing techniques for state-of-the-art GGP systems (Clune, 2007; Schiffel & Thielscher, 2007b; Kuhlmann, Dresner, & Stone, 2006).

In this thesis we describe CADIA-Player, a high-performance GGP agent. Unlike existing state-of-the-art GGP agents that use the traditional approach of game tree search combined with (automatically learned) evaluation functions, our agent uses simulation-

based approaches exclusively for reasoning about its actions. More specifically it uses Monte-Carlo rollouts augmented with upper confidence bounds applied to trees, the so-called UCT algorithm (Kocsis & Szepesvári, 2006). This algorithm is a recent and exciting new approach for controlling simulation-based rollouts in game trees and has, for example, proved quite effective in the game of Go (Coulom, 2006, 2007; Gelly, Wang, Munos, & Teytaud, 2006). It also proved effective in our agent. CADIA-Player won the *Third Annual GGP Competition* held at the AAAI conference this year (2007) and is currently the reigning GGP world champion. Furthermore, empirical evaluations presented in this thesis show that UCT outperforms naïve Monte Carlo simulation approaches with a significant margin on a wide range of adversary games.

The main contributions of this thesis are:

1. Showing the effectiveness of simulation-based approaches in the context of GGP.
2. The development of a world-class GGP agent.
3. Empirical evaluation of the UCT algorithm on a wide variety of games.
4. Introduction of new practical domain-independent techniques for further enhancing UCT simulation rollouts in unseen parts of the game tree.

The thesis is structured as follows: In the next chapter we go over background work. Chapter 3 describes the architecture and inner workings of CADIA-Player and Chapter 4 focuses on the search algorithms used in CADIA-Player. In Chapter 5 we present experimental results and end by giving conclusions in Chapter 6.

Chapter 2

Background

In this chapter we give an overview of the most common algorithmic game-playing techniques as well as providing the necessary background in General Game Playing. We mainly focus on the search component of game playing as this is where the focus of the thesis is.

2.1 AI and Game Playing

The mainstream method for applying AI to game playing is to use *heuristic search*. Heuristic search is a method that uses evaluation from game-specific knowledge (e.g. piece values in chess) and assigns a heuristic value to game positions indicating how good they are. The higher the value, the better. Many such methods exist for single-player games (puzzles) and we will describe one such named *Enhanced Iterative Deepening* A^* which is a variation of the well known algorithm A^*. For two-player games the heuristic approach usually is based on an algorithm known as *MiniMax*. Many enhancements and extensions have been added to it, but for this thesis it is enough to understand MiniMax and one of its enhancements named *Alpha-Beta pruning*.

Another approach is to run simulations to estimate the values of states. We explain one such method called *Monte Carlo Simulation*. Simulation approaches have been used e.g. in programs for the game of *Go*, and the best *Go* programs today are all based on such an approach (Gelly et al., 2006; Coulom, 2006).

2.1.1 Enhanced Iterative Deepening A*

Iterative Deepening A* (IDA*) (Korf, 1985) is a depth-first variation of A*. It limits how deep it searches and raises this bound iteratively while it has not found a solution. The depth limit is controlled by a cost function which is evaluated at each node as the sum of actual depth traversed plus the heuristic value of reaching a goal. When no goal is found at a certain depth the next depth limit becomes the lowest cost value seen in the previous iteration that was higher than the cost limit then.

The enhanced part in the name comes from the fact that this variation has a transposition table and is therefore memory enhanced (Reinefeld & Marsland, 1994) .

CADIA-Player uses this algorithm as a first try for all single-player games.

2.1.2 MiniMax

MiniMax is a depth-first heuristic search method for adversary games that is centered around the simple assumption that our opponent will try to minimize our gain as we try to maximize it. It means that the actual gain of a move is limited by the minimal value the opponent can select during his turn. Figure 2.1 shows a game tree with minimax values calculated. The squares represent states where it is our turn to move and the circles when it is our opponents turn. The bottom squares are leaves and their value is what the evaluation function returns. An example evaluation function in checkers would, e.g. consider the difference in piece count between the two players, making the search favor having more pieces than the opponent. We traverse the tree depth-first so we look at the leaves for the left-most branch and see that they have the values 7 and 5, respectively. Since it is the opponent's turn when transitioning into one of these leaves he or she will always try to minimize the values available and select to go to the leaf that gives 5, therefore the circular node connecting these leaves has the value 5. In the square node above that it is now our turn to move and we can select between getting 1 and 5. We want to maximize the value so we select the branch that gives us 5. All the values are calculated with these maximizing and minimizing rules. Because we do not have the time to expand the whole game tree for any game of interest, the search depth is usually bounded and we select the move leading to the state with the highest minimax value as it is the best move we can see on the "horizon". The depth bound is thus raised iteratively until time is up.

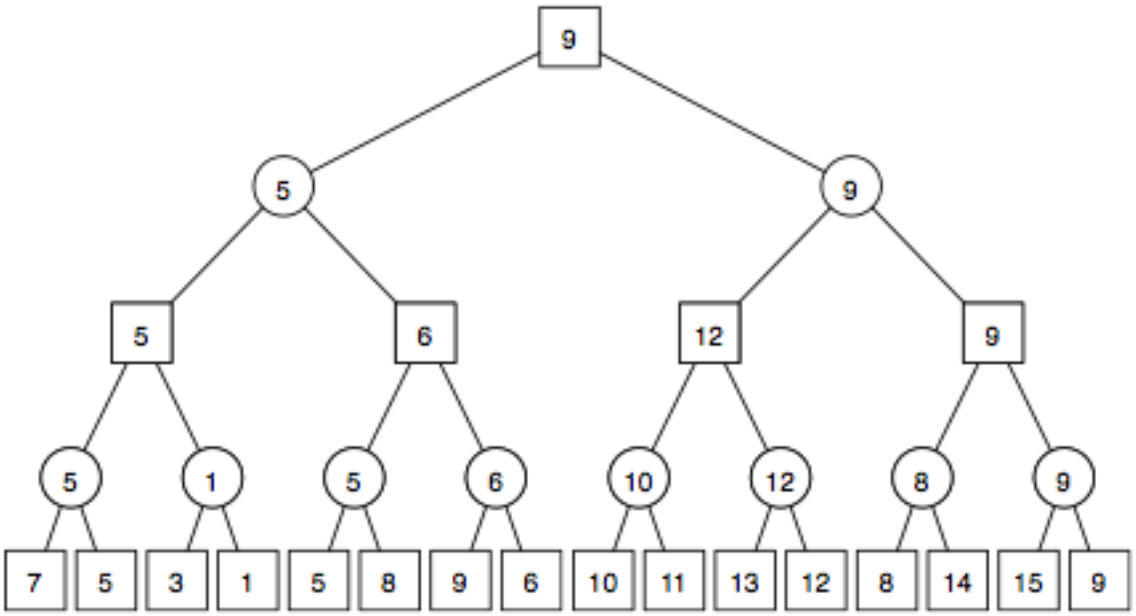

Figure 2.1: MiniMax game tree

2.1.3 Alpha-Beta Pruning

MiniMax is good in theory, but it is much too slow to be useable in practice in competitive programs. One of the most effective enhancements to it is *Alpha-Beta Pruning*. As the name indicates, this method prunes the game tree so we search a smaller tree and can therefore expand our search horizon deeper. Alpha-Beta identifies nodes that lead to subtrees that have no chance of changing the current game tree value. Look at the node where it is the opponent's turn in the second node in the fourth ply row in Figure 2.1. There we evaluate the first child to have the value 3. This means that no matter what the other children evaluate to, the opponent can select it so we will get at most 3. We already know that we can get 5 by selecting the move that leads to the node left of this one so we can conclude that there is no need to expand and evaluate any more children of the node as it would return a lower value than the one we have already secured. This way we can prune many irrelevant subtrees. Similarly we can see that when the opponent is guaranteed to minimize some value, we do not need to explore branches that produce higher values on the same level as the opponent will never select them. This extension therefore forms a window around the nodes that can possibly change the MiniMax value by storing a lower bound (α) and an upper bound (β) on each alternating level of the game tree.

2.1.4 Multi-Player Games

Max^n (Luckhart & Irani, 1986; Sturtevant & Korf, 2000) is an algorithm that extends MiniMax to accommodate multi-player games. At each node a tuple with the scores of all players is recorded and the Minimax values backed up the tree are the tuples resulting from all players trying to maximize their own score when its their turn.

It is also possible to reduce multi-player games to two-player games, but it comes at a cost and some information is lost. This is done by assuming that all the other players have united against you and we therefore combine them all into a single player. This is known as the *Paranoid algorithm* (Sturtevant & Korf, 2000).

As currently all GGP agents we are aware of do this simplification, including our; we will not discuss it further here.

2.1.5 Monte Carlo

Monte Carlo (MC) (Sutton & Barto, 1998) methods are a form of reinforcement learning methods. They learn only from experience and therefore have no need for any prior knowledge of the task they are applied to. To immediately start showing some intelligent behavior, especially when playing games, it is not very effective to learn from real experience as learning good strategies may take tens or hundreds of games, even for a human. If MC has an internal representation of the task at hand available, the experience can be generated by simulating lots of games between moves during actual play. The next move in the actual game will then be chosen from the experience gained from the simulations. A simulated task that has finitely many steps is also known as an *episodic task*. A single simulation through such a task is known as an *episode* and the sum of the rewards gained during an episode is known as a *return*.

If we assume that all games have a state space too big to be exhaustively explored then MC methods actually learn by just sampling the state space of a task by using the rewards encountered to estimate state values. The rewards are discounted so rewards of equal value when encountered will seem different to the evaluating state if the path to them is different in length, making the one with the shorter path more preferable. Every state-action pair in the model keeps track of the estimated total reward, $\mathcal{Q}(s, a)$, the player can expect to get in the remainder of the task if action a in state s is taken. Each time a simulated episode reaches a terminal state the rewards are backed up to all state-action pairs included in the episode and are averaged into the estimated rewards. The collection of all these estimated rewards make up the MC value function.

To run the simulations there must be some kind of policy to select which actions to take, whereof the random policy is the simplest one to ensure that every state will at some point be visited as the number of simulations grows. The MC policy does not need to be fixed and can evolve after each simulation. The simplest of these is the greedy policy, or just "go to the state with the highest value". Such a policy does not explore anything beyond what initially appears to be the best course of action if the state value does not converge to a lower value. In this case we can miss something because we do not explore, we just exploit what we already know, not wasting time rechecking or verifying anything. Intuition therefore tells us that when we do not have time or space to examine everything, the way exploration and exploitation is balanced in the policy will be the deciding factor in how intelligent the program will be. MC policies can be learned on-line and off-line, and it is even possible to learn one policy while following another. Therefore when playing a game it is best to learn the greedy policy by following a more exploratory one so good actions don't get overlooked and moves in the actual game are not exploratory moves.

The averaging of state values can be implemented incrementally by storing only the current value and number of visits for each node at time t in the following way:

$$\mathcal{Q}(s,a)_t = \mathcal{Q}(s,a)_{t-1} + \frac{1}{N(s,a)_t} \times (R_t - \mathcal{Q}(s,a)_{t-1})$$

where $N(s,a)$ is the number of times the state-action pair (s,a) has been selected for execution and R_t is the reward received when moving to state s. Note that the divider $N(s,a)$ can never be 0 because we count the visit to the node before we average its value.

MC can also be used to estimate the average state values in just the same way as state-action values.

UCT

The UCT algorithm (Kocsis & Szepesvári, 2006) is a recent and exiting development in how to handle the exploration-exploitation tradeoff in MC simulations of games. We discuss it in detail in Chapter 4, where we describe the search component of CADIA-Player.

2.2 General Game Playing

GGP is the brainchild of the Stanford Logic Group and provides a standard way of both describing game rules and how players can communicate with each other to play. The aim of an intelligent player in GGP is to be able to form a good strategy for any game described to it. This ability to play any game sets GGP programs apart from any other game playing programs that are programmed to play only a single game well. Such programs might be able to play a very complex game at a master level but have absolutely no idea how to play a simple game like Tic-Tac-Toe. An overview of the GGP concept can be found in (Genesereth et al., 2005). An annual GGP Competition is held by The Stanford Logic Group.

2.2.1 Game Description Language

The game rules in GGP are described by a language called *Game Description Language* (GDL). GDL is a variant of *Datalog* which is a query and rule language similar to *Prolog* and uses first order logic. GDL is expressive enough to describe a wide range of games, including various single-agent puzzle games, different two-player adversary board games (like Chess, Checkers, etc), and even multi-player games. The games can be adversary and co-operative, and need not be zero-sum games. The two main current restrictions on the game type are that they must be deterministic and complete information games.

In addition to general logic functions, GDL has some reserved keywords to describe things like game states, transitions, and terminal conditions known as *relations*. To ensure that the states of a GDL description are finite and its transition function is decidable, recursion is restricted. GDL is presented in Knowledge Interchange Format (KIF) (Genesereth & Fikes, 1992). Following is a brief overview of the GDL relations; the explanations of the relations use the selected lines from Tic-Tac-Toe in Figure 2.2 as a reference (the complete description of Tic-Tac-Toe in GDL/KIF can be found in Appendix A).

The *role* relation takes an atom as a parameter and declares that atom to be a name (role) of a player participating in the game being described. We can see in Figure 2.2 that two roles are declared, *xplayer* and *oplayer*. No roles may be added to or removed from the knowledge base after the initial role declarations. The *true* and *init* relations take a rule or an atom as a parameter and hold when that rule is present in the knowledge base as a part of the current state. The difference between them is that *init* is used to create the initial game state and is not used after that where as *true* is used throughout the game to evaluate if something is true in the current game state. When the *legal* relations are evaluated with

```
(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)))
...
(<= (next (cell ?m ?n x))
    (does xplayer (mark ?m ?n))
    (true (cell ?m ?n b)))
...
(<= (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 2.2: Selected lines from Tic-Tac-Toe in GDL

a given role name its solutions are the legal actions for that role in the current state. If we take a look at the *legal* relation in Figure 2.2 we see that it contains some atoms beginning with "?". These are variables in KIF. We replace $?w$ with a role name and ask for solutions to $mark\ ?x\ ?y$. The solutions are all those where a corresponding cell contains the atom b and this role holds for the *control* predicate. If we apply this rule to our role in the game, this rule tells us that if a cell on the board is blank and it is our turn, we may place a mark in it. To actually place a mark in the cell we have to prepare the knowledge base for state transition. One legal action per role is added to the knowledge base wrapped in the *does* relation including the role name. This allows the next game state to be proven against the current state with the actions the players choose to take. It is important to remember the fact that a move is needed for all roles before a state transition can take place. This allows for simultaneous moves, and when turn taking is simulated, players can only choose one ineffectual action when it is not their turn. This action is often referred to as the *noop* action. After the *does* relations have been added to the knowledge base, the *next* relation can be proven. Its solutions are the rules that are true in the next game state. What the *next* relation in Figure 2.2 states is really just, "if *xplayer* marks a cell that is currently blank, it will contain x in the next state". Note that the rules and atoms describing the prior state should be discarded completely, as should the *does* relations when the new state has been asserted into the knowledge base. The *terminal* relation takes no parameters and if it is proven the game has reached a terminal state and is over. In Figure 2.2 we see that the game is terminal if $line\ x$ is true. The $line$ predicate is declared a few lines above as being a synonym for a row, $column$ or a $diagonal$ of a given symbol (x in this case). Then above that we see how row (declarations for $column$ and $diagonal$ are omitted) is declared from what is true about the cells in the current state. Finally we look at the *goal* relation. Once a game has reached a terminal state the scoring for the game is performed by proving the *goal* relation given a role name. Scores can vary from 0 to 100 and these scoring atoms are the only ones that may be treated as having a numerical value in the game description. We can see that in Figure 2.2 there are two goals declared for $xplayer$ who marks his cells with an x, one of a score 100 if there is a $line$ of xs on the board when the game is over, and the other of a score 0 if there is a $line$ of os. In the full descriptions the reverse of these goals is defined for $oplayer$. The complete GDL Specification is in (Love, Hinrichs, & Genesereth, 2006).

2.2.2 GGP Communication Protocol

In order for a GGP agent to play a game it must be able to communicate with a *Game Master* (GM). The GM is a server that stores game descriptions and has the ability to

contact multiple GGP agents and broker a match between them, making sure it does not get corrupted by illegal moves. To communicate with the GGP agents, GMs implement HTTP clients that send requests to the GGP agents which in turn run HTTP servers. Before a game can be played all GGP agents participating in it must be registered at the GM. The GM starts by sending all the players a match identifier, the game description, their role in the game and the time limits they have for both preparing and playing. After all players have responded, play commences by the GM requesting a move from all players and once all have replied the GM sends another move request including all moves made in the last round. This way each player can update its game state in accordance with what moves the other players chose. 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 value sent for them is presented in seconds and the *startclock* indicates the time left until the game begins from receiving the rules and the *playclock* indicates the think time the player has between moves.

More details, including the format of the HTTP message contents can be found in (Love et al., 2006). A full description of the GM capabilities is given in (Genesereth et al., 2005).

2.3 Conclusions

We gave a brief background of the workings of game-playing programs, and highlighted the challenges of General Game Playing. In the next chapter we describe the architecture of our GGP agent.

Chapter 3

CADIA-Player

> Human beings, who are almost unique in having the ability to learn from the experience of others, are also remarkable for their apparent disinclination to do so.
>
> *Douglas Adams (1952-2001), Last Chance to See.*

CADIA-Player 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 take part in the annual GGP competition held by the Stanford Logic Group.

An agent competing in the GGP competition requires at least three components: a HTTP server to interact with other players through the GM, the ability to reason using GDL and, of course, AI to play the games presented to it. In this chapter we describe the architecture of CADIA-Player and the extendable framework it is built on.

The player is written in C++ on Linux/Unix and can be compiled and run on both systems unchanged.

3.1 Architecture

The architecture of CADIA-Player is shown in Figure 3.1. The topmost layer of the figure is a 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 process is spawned and the old one suspended. This HTTP server can be thought of as an extension to CADIA-Player, allowing it to communicate over HTTP to the GM, but it has nothing to do per se with playing GGP games. CADIA-Player itself can be split up into three conceptual layers, the *Game Agent Interface*, the *Game Play Interface* and the *Game Logic Interface*. On top

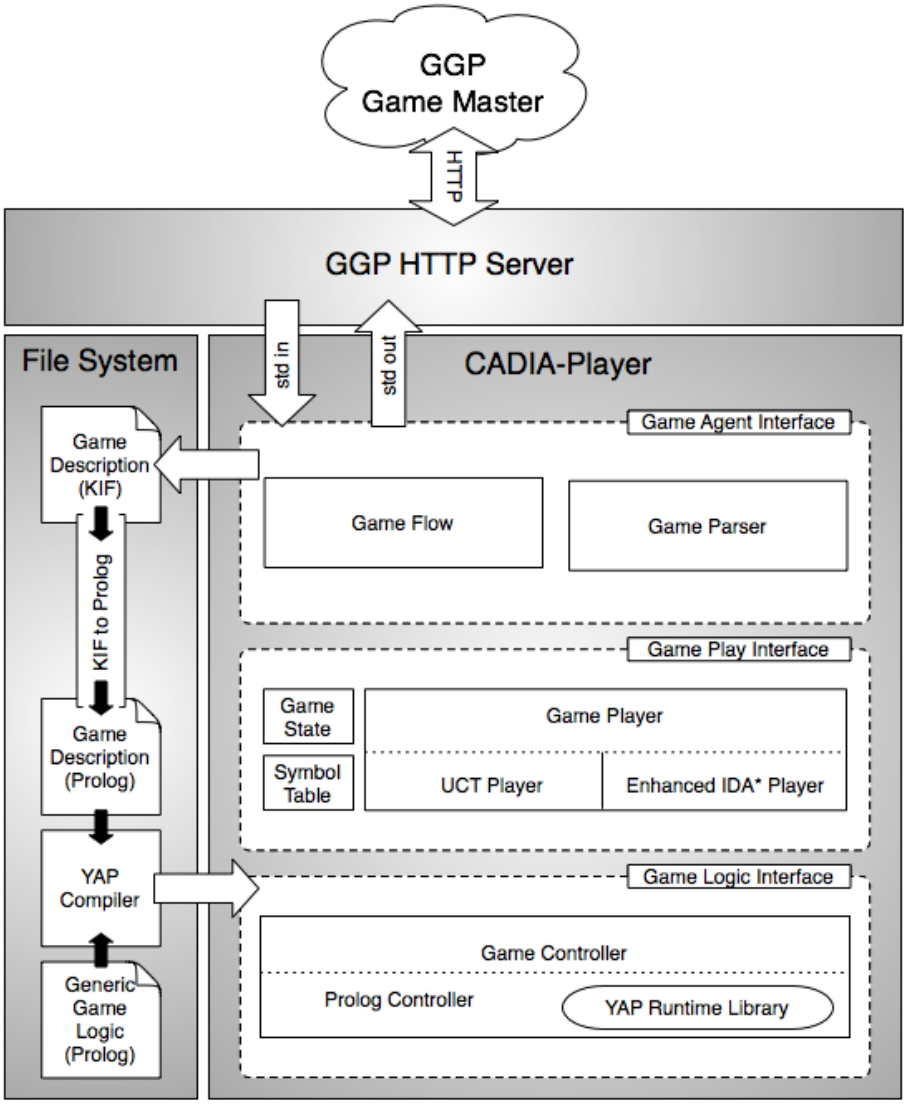

Figure 3.1: Overview of the architecture of CADIA-Player

we have the Game Agent Interface which handles external communications in addition to managing the flow of the game being played. The Game Agent Interface queries the Game Play Interface for all intelligent behavior regarding the game.The bottom layer is called Game Logic Interface and is where the state space of the game being played is calculated and manipulated. CADIA-Player is a collaborative project, and the HTTP server, KIF to Prolog conversion and the Prolog wrapper code were written by Yngvi Björnsson.

3.1.1 Overview

The following subsections give a detailed description of each of the components of Figure 3.1. However, we start with a quick run-through example of how the components interact when playing a game. All names in italics represent a component in Figure 3.1.

Starting a New Game

A game begins when the *GGP HTTP Server* gets a message from a GM announcing a new game. The server starts a new *CADIA-Player* 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 Flow* 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 Flow*. The *Game Flow* proceeds to selecting the type of *Game Controller* to use. CADIA-Player uses the *Prolog Controller* which utilizes the Prolog engine YAP (*YAP Prolog*, 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. The control is now returned back to the *Game Flow* which selects the *Game Player* that the *Game Play Interface* should use as its AI. 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 startclock is up a message indicating that CADIA-Player is ready to play is returned back through the pipe to the *GGP HTTP Server* so it can notify the GM.

Making the First Move

When CADIA-Player has announced that it is ready a new HTTP request can be expected. This time the GM is requesting a move decision. Like before the message is sent through the pipe and ends up in the *Game Flow*. 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 Flow* which relays it back to the *GGP HTTP Server* to be sent to the GM.

Playing the Game

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. Remember that all GGP games use simultaneous moves (see Section 2.2.1). CADIA-Player 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 to send to the GM in the same way as for the first move. This exchange of move lists and move decisions is now repeated until the game ends.

Game Ends

A special stop message is sent from the GM when the game is over. This message contains the last moves and after parsing them the state in the *Game Logic Interface* should be terminal. The *Game Logic Interface* can now be queried for the final scores, i.e. the goals the players have reached.

3.1.2 Game Agent Interface

This layer is responsible for connecting the communication from the GM to the actual play logic. It has access to functionality to communicate with the GGP HTTP server using structures rather than just the text content of the messages from the GM. It should control the flow of the game with regards to the GM messages so that the player is in sync with it.

As mentioned earlier the player starts by saving the game descriptions it gets to a file. The file is named after the match identifier received from the GM and given the extension ".kif" (as it is in KIF format). The match identifier is propagated throughout all major parts of the player.

It is the responsibility of this layer to initialize the Game Play Interface (see Section 3.1.3) and, depending on the game description, to select what implementation of play logic to use. Because the design for the play logic (called Game Players) uses an interface that only requests an action given a game state (see Section 3.1.3), there is nothing that says one has to stick with the same play logic for every game or even throughout a single game. It just depends on how the Game Agent Interface is implemented. CADIA-Player has two different Game Players and sometimes uses them both in the same game instance. This layer is also responsible for which method to use for game advancement in the Game Logic Interface. In the case of CADIA-Player it selects Prolog for this.

The two internal parts of the Game Agent Interface, Game Flow and Game Parser, have the following purpose:

Game Flow

This represents the actual main function of CADIA-Player. It starts by initializing the Game Play Interface and the play logic and then the execution enters a structure of conditional statements that react to the incoming messages from the GM. This is basically starting new games, updating the state space, replying to move requests, and trying to restore communication synchronization if something unexpected happens like lag in the HTTP.

Game Parser

This is a parser used to build an internal representation for the Game Play Interface from the KIF game descriptions so it can reference any atom and also produce KIF strings from it. While parsing, a symbol table is created that maps all strings in the KIF game descriptions into unsigned integers. This lays the groundwork for the ability to model a game tree in an efficient manner. The parser also converts moves sent from the GM into the internal form.

The lexical analyzer of this parser was built using *Flex* (*flex: The Fast Lexical Analyzer*, n.d.). This parser was originally created for a theorem prover that we built for move generation but was discontinued later as YAP Prolog had better performance. The structures our theorem prover used were robust when it came to storing and looking up states and actions and we therefore kept them and further developed them into a standardized representation of the game logic in the framework.

3.1.3 Game Play Interface

This 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 to GDL/KIF.

The following subsections correspond to the internal parts of the Game Play Interface in Figure 3.1.

Symbol Table

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 32-bit unsigned integer value. As GDL demands that the names of the atoms have no lexical meaning, no information is lost by this transformation. 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.

The symbol table uses one little trick to make it easier to do a lookup on role names. When all roles have been parsed, the roles are moved to the front of the symbol table in the order they were declared. The role names thus get values $0, 1, 2, \ldots$, which is convenient because they are used as indices into play structures.

Game State

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. Figure 3.2 shows an example of a game history in Tic-Tac-Toe.

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. 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 is not guaranteed to be in the same state as the GM and can unknowingly start to send illegal moves. The inner structure of the game state is explained more thoroughly in Section 3.2.

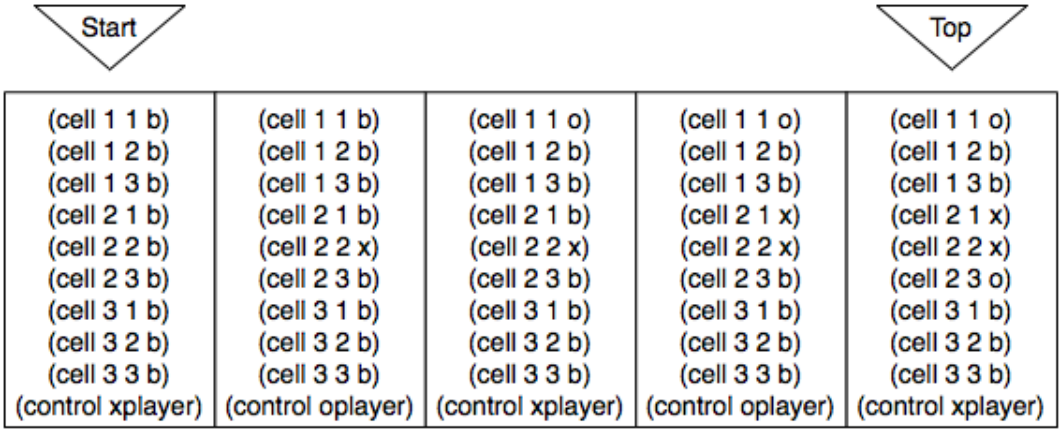

Figure 3.2: Example game state stack in Tic-Tac-Toe

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. A detailed description of the Game Player interface can be seen in Table 3.1.

CADIA-Player uses two implementations of Game Player: one uses UCT search and the other Enhanced Iterative-Deepening A*.

UCT Player

An implementation of a game player that uses UCT search to decide on the next action. This is the main player CADIA-Player uses. A detailed description of the inner workings of the UCT player is given in Chapter 4

Enhanced IDA* Player

A player that implements the Enhanced IDA* search algorithm tries to solve single-player games during the startclock. If it is successful in finding at least a partial solution (i.e. a goal with more than 0 points reward) it is also used on the playclock. However, if unsuccessful the Game Agent switches to the UCT player for the playclock searches.

The main focus in CADIA-Player is adversary games, so this single-player solver is still rather rudimentary. In particular, because we use no domain-specific information the heuristic function $h(s)$ always returns 0. The cost function for the search, $f(s)$, only uses

Table 3.1: 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.
play	Queries the player for what action it would take given the current state.
getPlayerName	For logging purposes only to allow the programmer 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 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 multi-player games or to get at least some 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. This function is implemented in the base class.
hasTime	Checks if the internal timer has exceeded the think time allowed for the player. This function is implemented in the base class.

the $g(s)$ part, evenly expanding all paths in an iterative-deepening depth-first fashion. This player could improve dramatically with any form of heuristics that can be drawn out of the game description.

Using this player ensures that for easy puzzles CADIA-Player is guaranteed to find the optimal solution.

3.1.4 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 a well defined interface called Game Controller.

Table 3.2: Game Controller

Function	Description
init	To initialize the Game Controller. Should be called before any other method.
getMoves	Returns the list of available moves for a specific role.
make	If given moves for all roles it should advance the game with those moves, but if only given a move by a single player the controller should add random moves for all other players before advancing the game (remember from Section 2.2.1 that all games use simultaneous moves and turn taking is actually simulated).
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.
ask	Does not need to be implemented for the standard Game Play Interface and is for future development. Because the game description is first order logic based it should be possible to ask any logical question of a knowledge base created from the GDL.
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.

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 beneath. As briefly mentioned before, we have tried two different theorem provers as Game Controllers, YAP Prolog and a custom implementation.

A detailed description of this interface is provided in Table 3.2.

Yap Runtime Library

This is a C interface to YAP Prolog (Yet Another Prolog) (*YAP Prolog*, n.d.). YAP is free to use in academic environments and is a high-performance Prolog compiler. An advantage of this Prolog for our purposes is that it provides a runtime component making it relatively straightforward to call it from other programming languages.

Prolog Controller

This is the game controller implementation that CADIA-Player uses. As the name indicates it uses Prolog as a theorem prover for the GDL logic of the game. It 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 very similar to Prolog code so it is easy to write it back out as such by making some minor syntactical changes like format of parameters and operator symbols. Then it calls YAP through a system command making it load the converted file plus a file containing some predefined functions (see Section 3.3) 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.

3.2 Game Tree

The player builds a game tree in memory by incrementally storing simulation results. The game tree is modeled in the Game Play Interface of the architecture. An overview of the implementation is given in Figure 3.3. As depicted in the figure, we have sets of states stored in hash tables on different levels that form the game tree where each level stands for a certain depth in the game tree. Every state has a collection of possible actions and each action references a state-action pair node containing the data needed to maintain the Monte Carlo value function.

The game tree uses 64-bit Zobrist keys (Zobrist, 1970) to identify its states and actions and allows fast retrieval. When the game tree is initialized against a game description, it builds a map where each symbol in the symbol table is assigned a 64-bit pseudo-random number. 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. There is the chance of getting non-unique identifiers, but with this scheme the chances are so remote that we do not care in the cases it happens. An identifier for each of the compound sentences is retrieved by applying XOR to the Zobrist keys for the atoms. These identifiers are then

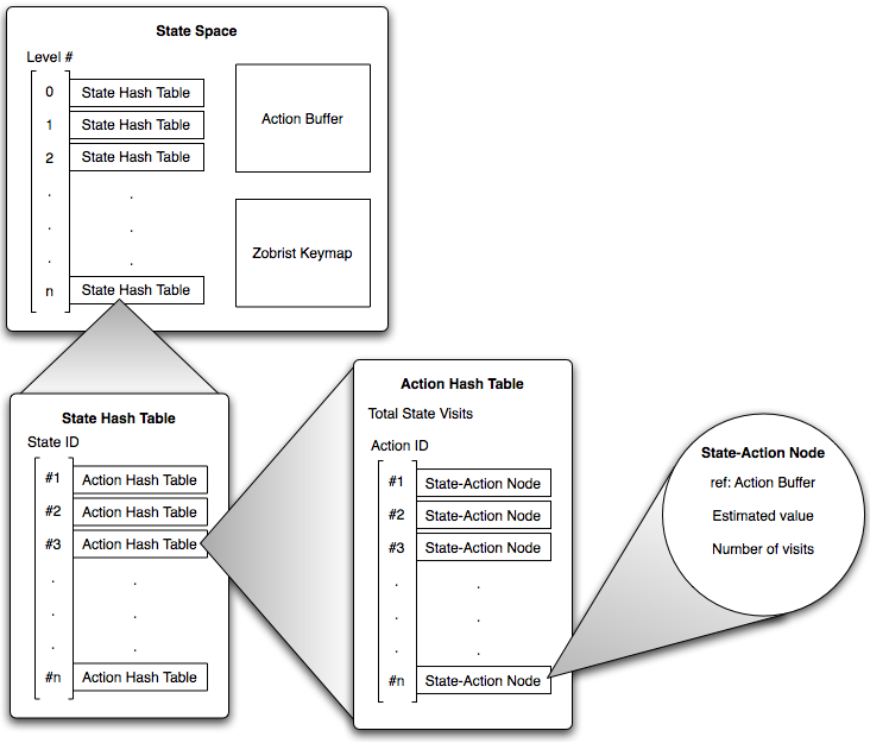

Figure 3.3: Overview of the game tree implementation

combined into a single identifier for the state using XOR. Algorithm 1 shows how we can get an identifier for a compound sentence which can recursively call itself for any nested compounds.

In Algorithm 1 the *symbol* function is used to get the identifier symbolizing the predicate atom of the compound sentence denoted by C. The *keymap* function is used to look up the Zobrist key for the identifier from the keymap in the model (see Figure 3.3). Shifting the bits after every symbol is needed to maintain uniqueness. Consider these two compound sentences that can be found in many states of the game *Tic Tac Toe*:

```
(cell 1 2 b)
(cell 2 1 b)
```

They have different meaning but contain exactly the same symbols. If these were to be individually given an identifier without any bit-shifting they would get exactly the same identifier because using XOR to combine each symbol key is indifferent when it comes to operand ordering. Also the circulation of bits must be done in case of compound predicates with many parameters. The shift makes the later parameters affect fewer and fewer bits until they have been pushed completely out of the 64-bit key. This means that

Algorithm 1 getZKey(Compound C)

1: $key \leftarrow keymap(C.symbol())$
2: $shift \leftarrow 7$
3: **for all** $c_i \in args(C)$ **do**
4: $\quad key \leftarrow key \oplus rotate(getZKey(c_i), shift)$
5: $\quad shift \leftarrow (shift + 7) \bmod 64$
6: **end for**
7: **return** key

State A	State B
`(cell 1 b)`	`(cell 1 x)`
`(cell 2 x)`	`(cell 2 b)`

Figure 3.4: Example game states

predicates with many parameters that only have the last one different from each other would get exactly the same identifier if nothing is done.

Creating an identifier for the state itself is done using Algorithm 2.

The reason why this also needs to be bit-shifted is simple. Consider an example where we have a game which can be at the two states shown in Figure 3.4 amongst others. Their predicates and first parameter will map to the same identifier. Clearly what has any chance of distinguishing them are the identifiers of the second parameters. But if all compound sentences are offset the same, then the states will change their identifier by:

State A: *rotate(getZKey(b), 14)* $\oplus$ *rotate(getZKey(x), 14)*

State B: *rotate(getZKey(x), 14)* $\oplus$ *rotate(getZKey(b), 14)*

where *rotate(id, offset)* is defined as offsetting *id* by *offset* bits in a circular fashion. Both these calculations will end up with the same identifier. To avoid this we need to have different offsets of the entire compound sentence for different states.

Algorithm 2 getZKey(State S)

1: $key \leftarrow 0$
2: $shift \leftarrow 0$
3: $C \leftarrow getCompounds(S)$
4: **for all** $C_i \in C$ **do**
5: $\quad key \leftarrow key \oplus rotate(getZKey(C_i), shift)$
6: $\quad shift \leftarrow (shift + 1) \bmod 64$
7: **end for**
8: **return** key

Having to offset the compounds of the state adds one constraint. To be able to recognize a state again, its compound sentence must appear in the same order so the offset of each one will be identical to when it was first encountered. This forces us to introduce some sort of consistent ordering to the sentences in the state. This is done by ordering on the numeric identifiers of the atoms as they appear in the compound sentences while the sorting order is unresolved.

3.2.1 Action Buffer

The box marked as *Action Buffer* in Figure 3.3 is a buffer for the actions encountered. As an action can reoccur in multiple states (like being able to play the same pawn in many different states at various game tree depths in chess), we save space by storing the action on the internal form when first encountered and then just referencing it in the nodes. Because the model now knows the available actions for any state that has been added to it, there is no need to query the Game Logic Interface for available actions, as all data needed to make a transition is stored in the *Action Buffer*.

3.3 State Space

Initially we built a simple theorem prover for finding available moves and making moves based on GDL game descriptions. Later we abandoned it, and instead translated the GDL descriptions into Prolog code as it improved performance significantly.

CADIA-Player uses Prolog to set up the functions for traversing the state space for every game from the GDL description. Even though all logic concerning the state space is included in the GDL, there are some things that need to be done extra in Prolog to get a practical working state space.

3.3.1 Adapting the Game Description to Prolog

Before the first move is made the contents of the *init* relation are dropped from the description, moved into *true* relations and asserted into the knowledge base as the initial state. Now we can start querying what moves are legal through the *legal* relation. Making a move just using the logic in the GDL takes quite a few calls to the Prolog engine and we would like to encapsulate this into a single function. The new function should take the

Algorithm 3 makeTransition(Role[] r, Move[] m)

1: **for** $i \leftarrow 1$ to $length(r)$ **do**
2: $\quad addDoesRelation(r[i], m[i])$
3: **end for**
4: $S' \leftarrow proveNextRelations()$
5: $retractRelations(DOES)$
6: $retractRelations(TRUE)$
7: $C \leftarrow getCompounds(S')$
8: **for all** $C_i \in C$ **do**
9: $\quad addTrueRelation(C_i)$
10: **end for**
11: **return**

actions of all the players and when it returns the knowledge base is in the new state. Algorithm 3 outlines how we can do this. We begin by asserting all actions into the knowledge base and then get all proofs of the *next* relations (which will be the new state). Then we remove the actions and the current state from the knowledge base and end by asserting the new state into it.

This poses one problem. There is nothing in the GDL that describes how to undo a transition. Therefore we need to extract the current state before making the transition to the next state. This can be done by pulling out a list of all proofs of the *true* relation and store it for when we want to retract to this state. To retract a transition we finally add a new function that takes in a list of stored rules as a parameter, removes the current state from the knowledge base and asserts all elements of the list embedded in *true* relations.

Nothing needs to be done to simplify proving the *terminal* and *goal* relations.

GDL contains a predicate named *distinct* and takes two clauses as parameters and represents the *not equal* relationship between the parameters. This can be resolved in YAP with:

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

Similar predicate named `or` used to exist for the "logical or" relationship and could take arbitrarily many parameters, but it has been deprecated. Still many KIF files exist that contain it so it is good to implement it also which can be done in just the same way as with *distinct* using multiple definitions for each parameter count. An example of the three parameter version is the following:

```
or( _x, _y, _z ) :- _x ; _y ; _z.
```

In our experience no KIF file has included an *or* with more than six parameters.

3.4 Search

As has been mentioned before, CADIA-Player is not restricted to use only one search algorithm. It can select between two different algorithms, one for single-player games and one for games with more players.

3.4.1 Single-Player Games

CADIA-Player runs an Enhanced IDA* search algorithm on the startclock for single-player games. The algorithm runs on the startclock while no solution with 100 points is found. The first solution found with any points is stored as the current best known solution and gets replaced if another solution with a higher score is found. If any solution has been found when the startclock runs out, CADIA player will stick to using the Enhanced IDA* during each playclock until the game terminates or a solution with the score of 100 is found. When the player decides on a move it simply pops the top action of the best known solution. If the best known solution has a score of 100 there is no need to keep on searching because 100 is the maximum possible score in GDL. At this point the player replies instantly with the next action of the best known solution.

If no solution is found on the startclock it is assumed that no solutions will be found on the playclock using the same algorithm and it is very probable that a good solution will not be found because with every step we have no solution we are forced to make a random move. To increase its chances, CADIA-Player switches to the UCT search because its depth-first search has the chance of hitting some return that might guide the search better. There might be some scheme that could be used to switch back to Enhanced IDA* when UCT detects that it is getting close enough to the terminal states that the exhaustive search of the Enhanced IDA* would become useful again, but we did not look into it.

3.4.2 Two- and Multi-Player Games

Two- and multi-player games are handled with UCT search from the start. The only difference in implementation is how the returns of the simulated episodes are calculated. For two-player games the return is calculated as the difference between our and the opponent's return. In multi-player games the player takes the paranoid approach and just uses its return unchanged, believing everyone else is against him. The logic behind this is that if there is more than one additional participant then it is not possible to know if you should maximize or minimize their returns from the game description so it is best to

just stick to what we know is always true. Maximizing our own score is never bad, even though it is debatable how good it is.

3.5 Related Work

Before GGP, Pell (Pell, 1996) suggested that AI programs should be able to play a whole range of games where it was part of the program itself to analyze the game rules and figure out how best to play the game they describe. From his theories he created the program *Metagamer*. Pell restricted *Metagamer* to chess-like games.

There are two other successful players besides CADIA-Player that have participated in the Annual GGP Competition and won. In the following section we will give an overview of these players and of a technique to transfer knowledge between games.

3.5.1 Cluneplayer

Cluneplayer (Clune, 2007) is a GGP agent written by James Clune and was the winner of the First Annual General Game Playing Competition in 2005. It uses an approach that is quite different from CADIA-Player. This player extracts features from the GDL description automatically to use in a heuristic evaluation function for the game states. It then uses this heuristic evaluation function in a MiniMax tree search with alpha-beta pruning, transposition tables and aspiration windows to play.

To discover usable features Cluneplayer begins by collecting candidate features that are calculated from the GDL. They include how often atoms appear (e.g. number of pieces), distances between atoms when set up in a graph whose edges are generated by GDL rules that advance the game (e.g. distance to king promotion in checkers) and how close a compound sentence that fulfills some goal is to being true. It also detects some additional features to exploit symmetry in the game description. This method collects a large number of candidates, many of which have no relevance when evaluating a state. The number of pieces for example may be in there, but so can much other cardinality information that has no value. There is nothing in the GDL that can help us directly pinpoint which of the cardinality features is the actual piece count. This problem is solved by introducing the notion of *stability*. A factor to estimate the stability of each candidate feature is calculated and only those deemed stable are used in the evaluation function. The stability is a number that is calculated by measuring how a feature changes through random plays of the game at hand. The higher the number the more stable the feature is. For example, features that

change incrementally get marked as very stable where as features which change up and down unpredictably get marked as unstable. When calculating feature stability, the more games that are measured the more accurate the results are so to guarantee at least some answer and still give the best answer reachable, the player starts by evaluating 25 games and then doubling that number and so on. The latest calculations that it was able to finish before the startclock ran out are then used to evaluate the game states during play.

The discovered features are categorized by if they contribute to payoff or control. When they all have been established they are weighted according to their stability and used in the evaluation formula. The evaluation looks at how close the game is to terminating and favors control features in the beginning of the game but shifts more towards the payoff features as the game gets closer to termination.

3.5.2 Fluxplayer

Created by Stephan Shiffel and Michael Thielscher, *Fluxplayer* (Schiffel & Thielscher, 2007b) was the winner of the Second Annual General Game Playing Competition in 2006. They use Thielscher's Prolog-based implementation of the *Fluent Calculus* called FLUX (hence the name of the player).

Fluxplayer uses Iterative-deepening depth-first search with transposition tables and the history heuristic (Schaeffer, 1989) for move decision. To create the evaluation function it recognizes structures in the semantical properties of the game description to retrieve successor relations, order relations, quantities and game boards. From this information Fluxplayer is able to estimate with *Fuzzy Logic* how close a positions is to being a winning position.

3.5.3 Knowledge Transfer

Knowledge transfer in GGP refers to taking knowledge from one game and transferring it to another. We will describe one of the approaches that researchers have worked on and give references to more resources on the matter.

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, have participated in all three GGP competitions with their player *UTexas LARG* (Kuhlmann et al., 2006), which we expect will include knowledge transfer techniques in the GGP competition next year.

To read more on the subject of knowledge transfer we refer to (Sherstov & Stone, 2005), (Taylor, Whiteson, & Stone, 2006) and (Banerjee & Stone, 2007).

3.6 Conclusions

We have given insight into the architecture of CADIA-Player and how it can be divided into three distinct interfaces. We have discussed in detail how the player builds a model of the game tree in memory and also how we build the state space in Prolog. The way CADIA-Player selects different algorithmic approaches to single and multi player games has been shown and an overview of other successful GGP agents given. Chapter 4 will discuss in detail the UCT search implemented in CADIA-Player.

Chapter 4

Search

CADIA-Player uses the UCT algorithm as its main search method. In this chapter we explain this algorithm as well as its implementation in CADIA-Player. We also discuss how CADIA-Player utilizes multiple CPUs and present an extension to its UCT search, which was developed after the GGP Competition.

4.1 UCT

The UCT algorithm (Kocsis & Szepesvári, 2006) is a variation of the Upper Confidence Bounds algorithm (UCB1) and simply stands for UCB applied to trees. UCB1 (Auer, Cesa-Bianchi, & Fischer, 2002) is a simple yet effective way to balance exploration and exploitation. It solves the exploration-exploitation tradeoff, which means that its regret (loss from not always making the best action) growth rate is bounded by a constant times the best possible regret rate. It keeps track of the average returns of all available actions $a \in A$ at time t and samples the one with the highest upper confidence bound given as:

$$a_t = argmax_{a \in A_t} \left\{ \overline{X}_a + \sqrt{\frac{2 \ln (t-1)}{s}} \right\}$$

where s is the number of times action a has been selected up to time $t-1$. If there exists an action that has never been selected and has therefore no estimated value, the algorithm defaults to selecting it before any, previously sampled action. There are exceptions to this rule, especially when we have some prior knowledge, allowing the action value to be initialized differently. The confidence bound can be described as the average estimated

value of taking an action plus the UCB bonus. Because of how the bonus is calculated with respect to visits, actions gradually build up their bonus when they are not visited and each time they are, the bonus drops. When visit counts are low the action with the current best estimated return value is selected for sampling, but in time the bonuses kick in and ensure the exploration of other actions that were initially estimated as suboptimal. If the actions continue to look suboptimal they will have to rebuild their bonus value to be considered again, which takes longer and longer each time, but if they pay off more than initially believed, their average value rises and they get chosen more frequently for sampling. The selection method chooses the actions from a confidence distribution. We exploit the best action until the number of samples including it has generated a certain level of confidence in its estimated return value. When a suboptimal action is explored, it means that the confidence level of the best known action is high enough that it is better to lower the uncertainty of the estimated value of the suboptimal action. The closer the value of a suboptimal action is to the best action the sooner the need to select it arises.

To apply this algorithm to trees UCT adds a depth parameter to it, so instead of keeping track of only the values for actions available in the current state, $\mathcal{Q}(s,a)$, it keeps track of $\mathcal{Q}(s,a,d)$ where d is the depth in the game tree. To simplify, we will for the remainder of the chapter assume that the depth d is a part of the state s, making states at each depth distinct from states at other depths, and just refer to the value function as $\mathcal{Q}(s,a)$.

In order to be able to tune the UCB bonus, UCT uses it in the form:

$$2C_p\sqrt{\frac{\ln{(t-1)}}{s}}$$

where $C_p = \frac{1}{\sqrt{2}}$ matches the UCB formula, but can now be tuned by altering C_p. To simplify, we merge the constant into the C_p parameter and adjust the formula to use the notation of the MC value update formula in Section 2.1.5. We do this to clarify how it is used in the action selection of the MC rollout of CADIA-Player:

$$a_t = argmax_{a \in A_t}\left\{\mathcal{Q}(s,a) + C_p\sqrt{\frac{\ln N(s)}{N(s,a)}}\right\}$$

Recall that the N function returns the number of visits. CADIA-Player has the C_p parameter set to 40. This value was empirically selected by running it with various C_p settings, ranging from 1 to 100, against a CADIA-Player instance with the original C_p setting. The game used was Connect Four with both start and play clocks set to 30 seconds. The excact

Algorithm 4 search()

```
1: if isTerminal() then
2:     return goal(role)
3: end if
4: moves ← getMoves(role)
5: move ← selectAction(moves)
6: reward ← make(moves[move])
7: q ← reward + γ * search()
8: retract()
9: updateValue(moves[move], q)
10: return q
```

value is not critical and setting it a little larger or smaller has no significant effect. Future work may include fine tuning this value and seeing if the most appropriate setting differs between games or even between phases of a game.

4.2 Implementation

The implementation of the basic UCT algorithm is rather straightforward. It is used in the MC rollout phase and changes how the next action in the simulation is selected. It also constructs a tree in memory to keep track of future action values. The basic recursive MC rollout encapsulating the UCT action selection can be seen as Algorithm 4 and the UCT action selection itself is shown as Algorithm 5. Note that γ is a discount factor. Discounting makes the algorithm prefer earlier rather than later payoffs and CADIA-Player uses 0.99 as its discounting factor.

Algorithm 4 is a simplified version of the one used in CADIA-Player and is presented first for clarity. It works by first checking if the current game has reached a terminal state. If the game has terminated, the goal value for the role we are playing is returned. However, if the game is not over, we start by getting all available moves for the role we are playing and then select which one of them to play (line 5). Remember from Table 3.2 that $make$ (line 6) will select a random move for all opponents when given only one move (because all roles move simultaneously). Next we advance the game to the next state and recursively search it (line 7). When the recursion starts to unwind, the returned value is discounted. We then retract the action taken to restore the game state as it was when this execution instance was entered. Now that we are in the correct state we can update the action we sampled with the q value. Finally, we return the q value up the execution stack so the previous state can discount and update its sampled action value.

Algorithm 5 selectAction(Moves) for UCT

```
1:  val ← −∞
2:  curVal ← 0
3:  foundmax ← 0
4:  randmax ← ∅
5:  for all m_i ∈ Moves do
6:     node ← statespace.getNode(Moves[i])
7:     if node = NULL then
8:        curVal = +∞
9:     else
10:       curVal ← node.q + C_p * sqrt(ln(node.state.n)/node.n)
11:    end if
12:    if curVal = val then
13:       randmax[foundmax++] ← i
14:    end if
15:    if curVal > val then
16:       val ← curVal
17:       foundmax ← 1
18:       randmax[0] ← i
19:    end if
20: end for
21: M ← randmax[rand() mod foundmax]
22: return M
```

In Algorithm 5 we show how to choose among available actions (called in line 5 in Algorithm 4). In it we loop over all the moves sent as a parameter into the algorithm and store the ones having the highest UCT value in a list. Each loop starts by retrieving the node for the current action from our game-tree model. If no such node exists[1] ($NULL$ is returned) it gets a value of $+\infty$ as it should be selected before all other sampled nodes. We end by selecting an action for sampling, which is done by selecting randomly from the list.

With many simulations, if storing the entire simulation tree, there is a possibility that the UCT tree grows too large to store in memory. We have two ways of counteracting this. First, after making a move, we can delete the parts of the tree that are no longer relevant for the game (e.g. moves explored that were not played). Secondly, for every simulated episode, only the first node beyond the UCT border (see Figure 4.1) is stored (Coulom, 2006). In order to do that we change the implementation of the $updateValue$ function so that it does not update the state space model if the state-action pair being updated is further than one ply away from the UCT border. An overview of a single UCT simulation

[1] Or they have been buffered when the state was previously encountered, but never selected for simulation.

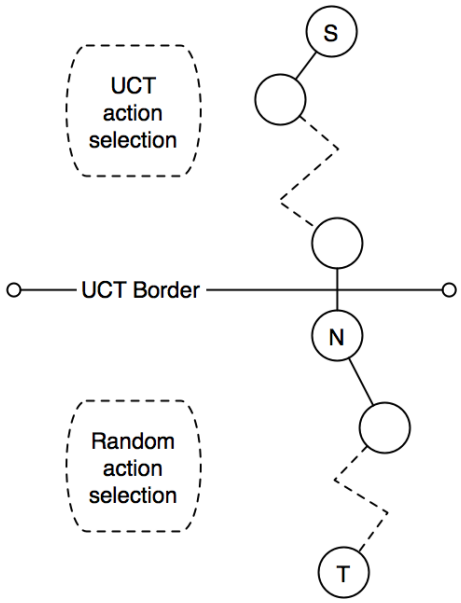

Figure 4.1: Conceptual overview of a single UCT simulation

is given in Figure 4.1. The start state is denoted by S, the terminal state with T and N is the new state added to the model after the simulation finishes. Because GDL rules require a move from all players for each state transition, the edges in the figure represent a set of moves. When the GDL simulates a turn taking game, the players not having their turn return a *noop* action that has no effect.

When the UCT border has been passed the model returns *NULL* nodes which have the value of $+\infty$ and are therefore explored first and if all nodes are *NULL* the tie breaking scheme results in playing a random game to the end of the episode. Because better actions are selected more often than suboptimal ones, the UCT tree grows asymmetrically. If an action on top of a particular branch is consistently good, the UCT tree grows this branch and in time completely takes over the random rollout below it. The opposite happens when the algorithm shows little interest in some branch and explores it rarely. That UCT branch does not grow much and we save memory by not wasting space on storing this branch in its entirety. This memory scheme can be said to save space where the algorithm saves time.

4.3 Opponent Modeling

Although UCT helps CADIA-Player to focus on its more promising paths, there still is a risk of much time being spent on irrelevant paths, namely paths that the opponent would never lead us down. Since CADIA-Player assumes no knowledge when the simulations

Algorithm 6 search(ref qValues[])

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

begin it will have to learn from experience what is good for the opponent, but the same logic applies here regarding action selection, it is better to explore the state space where the opponent believes he will do better. So for each opponent in the game a separate game-tree model is set up where estimates are built from the rewards received by that opponent. Because GGP is not limited to zero-sum games, the opponents cannot be modeled by using the negation of our rewards. Any participant can have its own agenda and therefore needs its own value function. All these game-tree models still share the simulations of the player and control action selection for the opponent they are modeling using the same UCT algorithm as the actual player. In fact inside the game player, every role in the game gets the same treatment during simulation and only the interfacing parts of the player see the UCT player as having a specific role.

A modified version of Algorithm 4 taking this into account can be seen as Algorithm 6 and is the one used by CADIA-Player. The *StateSpaces* array stores the different models. The functions *selectAction* and *updateValue* then use the model selected into *stateSpace*

to make their selections and updates. When the time comes to select the best action CADIA-Player's model is queried for the action with the highest $\mathcal{Q}(s, a)$ value.

4.4 Parallelization

Simulations are an appealing choice when it comes to parallelization because it is easy to run them concurrently and therefore distribute them between CPUs. Each CPU then runs its own set of simulations. The main concerns are how to combine the results of all the concurrent runs on another processor since results from one CPU may influence which simulations to run next.

Because the UCT tree is grown one node per simulation at the most, the first node to be selected randomly (if any) will be the only one of the randomly selected ones to need the return value of the episode. To calculate the value of the new node we need to know the number of actions from it until the episode terminated and the return of the episode. The number of actions is needed to be able to discount the return correctly. The estimated value of the new node will be $\gamma^l * return$ where l is the number of actions to the terminal state of the episode. We then back this estimated value normally up the UCT tree.

To set up a simple parallel search: When we cross the UCT border the random run through the state space can be unloaded to another CPU on the same or a different computer. When CADIA-Player starts up, it reads a file which contains a list of identities, IP addresses and port numbers, that indicate slaves that the processing can be unloaded to. Using TCP/IP communication the slaves are initialized with the game descriptions so they can simulate the game and, when needed, they are sent a representation of the current state and their result can be collected later containing the length and return value they got from running a random simulation from the state they were given. This parallelization scheme is based on ideas presented in (Cazenave & Jouandeau, 2007) and the low level parallel communication was written by Yngvi Björnsson.

4.4.1 Slave Programs for Parallelization

The slave programs used to run the random part of a simulation are a thin C++ layer on top of the same kind of Prolog initialization that the master program uses. The C++ layer contains the process communication and logic to randomly select moves to advance the game state in YAP until a terminal state is reached and a goal value can be obtained to return to the master player. In order to lower the communication overhead, CADIA-Player

also sends the slaves the discounting factor and a parameter indicating how many simulations they should run. The discounting factor is needed so the result of all simulations run can be averaged correctly into a single result, because taking averages of the path lengths and the goals will not give the correct result when discounted, it must be done separately for each simulation.

4.4.2 Other Parallelization Methods

Two other parallelization methods for UCT are presented in (Cazenave & Jouandeau, 2007). The first one is called *Single-Run Parallelization* and works by running many UCT search slaves under a single master that do not share any information. When the slaves run out of time they all report the estimated action values they calculated for their root. The master then combines all this information into its own root and selects what action to perform greedily from all available actions. The second method is called *Multiple-Runs Parallelization*. The setup is similar, but now the slaves report back to the master at predetermined intervals and at those intervals the information that the master calculates as its root is communicated to the slaves and they start a new UCT search.

4.5 Selecting among Unexplored Actions

The following section describes improvements we made to CADIA-Player after the GGP competition.

When encountering a state with unexplored actions CADIA-Player selects randomly from them until in the end they have all been selected once because it has no criteria for guessing which one is most likely to be best. One way to add such criteria in a domain independent way is to exploit the fact that the same actions can occur in the game tree, although at different states (Gelly & Silver, 2007). We can gather a form of history heuristics about all actions encountered by keeping track of a special $\mathcal{Q}_h(a)$ value for each action, independent of the state it was executed in. Like with the $\mathcal{Q}(s,a)$ values, this value is incrementally updated each time that action a is used in a simulation.

Now that we have our history heuristics, how do we use it to our advantage? We handle action selection differently in states that have unexplored actions. Instead of selecting randomly from the unexplored actions by assigning them all the initial value $+\infty$, each of them is checked to see if it has a history heuristic value and is assigned the historical $\mathcal{Q}_h(a)$ value if found. If no historical value is found, the action is assigned the maximum

GGP goal value (100) to bias towards similar exploration as is the default in the UCT algorithm. Future work may include examining different initialization values, like the average action-value of the state, for actions with no historical data. Now we have a set of actions that have been assigned a history value, and to select one of them we use a Gibbs distribution. Gibbs distribution is a modified Boltzmann distibution, and we use it in the form

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

where a and b are actions from the set of n available actions in the current state. Each action a has its probabilities of being selected calculated by this formula and then the next action is selected with these probabilities. The τ parameter is called the *temperature*. By manipulating it one can bias the probability distribution. As $\tau \rightarrow 0$ the selection becomes greedy and the action with the highest value is selected with probability 1. Higher values for τ flatten the distribution, making the difference in selection probabilities between the best and worst action become smaller.

Implementing this extension into CADIA-Player was done by storing the $\mathcal{Q}_h(a)$ value and the instance counter for the incremental update to the actions stored in the *Action Buffer* (see Section 3.2.1) maintained by the model. The new version adds to its $updateValue$ function the updates to the *Action Buffer*, changes the $selectAction$ function so it could detect when a totally unexplored action was amongst the ones to choose from, and if so switch to the Gibbs distribution action selection. An empirical comparison of the effectiveness of this improved version of CADIA-Player is found in Chapter 5.

4.6 Related Work

The UCT algorithm has been a great success in the game of Go and is used by the best computer Go programs today, e.g. *MoGo* (Gelly et al., 2006) and *Crazy Stone* (Coulom, 2006). Experiments showing how UCT can benefit from using an informed policy, instead of random, when the rollout exceeds the UCT border in Go are presented in (Gelly & Silver, 2007). The method, however, requires game-specific knowledge which makes it difficult to apply to GGP. Also in (Gelly & Silver, 2007) they show how to speed up the initial stages of the learning process in UCT by using *Rapid Action Value Estimation* which when updating an state-action pair (s, a) also updates all of (s', a) where s' occurred before s and action a was available. We did not experiment with this type of rapid learning in GGP, but it can be expected that the game being played will highly influence the benefits and perhaps there may be cases of negative effects.

4.7 Conclusions

We explained the UCT algorithm and showed its implementation in CADIA-Player where it is extended to play for all opponents to bias the simulations towards a more intelligent rollout for all participants. Ways to extend the UCT search to use more than one CPU were described. An improved technique to select from previously unexplored actions was introduced, which utilizes knowledge gathered when these actions are encountered at other states. Our implementation of the UCT search, along with the aforementioned improvements, enhances the playing skills of CADIA-Player.

In the next chapter we empirically evaluate how efficient our UCT search implementation, and the proposed algorithm enhancements, are in CADIA-Player.

Chapter 5

Results

In this chapter we give an empirical evaluation of CADIA-Player's UCT search in the context of GGP. The objective of the experiments is threefold: to demonstrate the effectiveness of UCT over standard MC-based approaches, to see how more simulations affect quality of play, and to evaluate the effectiveness of our improved action selection scheme for choosing among unexplored actions. The first has two parts to it as UCT augments MC in two ways, both by offering a different action-selection rule and by caching results between moves. We evaluate the effects of these separately.

To perform these experiments we have two baseline players. The first baseline player is a standard MC player that uses a uniform random distribution for action selection and has no memory to retain information about the game tree between moves. This player represents how of CADIA-Player would do if stripped of both the UCT action selection rule and its memory scheme. The second player is an identical MC player, but with the same memory scheme as the UCT algorithm. When compared to the first baseline player, this player shows the benefits of adding game-tree memory. Furthermore, comparing it to CADIA-Player shows the effectiveness of the UCT action selection scheme, the only difference between the two players. To minimize the effect of implementation details all players are built on the same software framework. Furthermore, no parallel roll-out scheme was used in these experiments, each player only ran on a single CPU. In the following sections the MC player without memory will be referred to as MC_{org} and the one with memory as MC_{mem}. CADIA-Player is referred to as CP_{uct} and its version using the improved action selection scheme for unexplored actions (see Section 4.5) is referred to as CP_{imp}. All the experiments use adversary games, which were our main focus when developing CADIA-Player.

5.1 UCT Competition Performance

To measure the level of improvement CP_{uct} achieves using UCT we selected three well known games as a testbed: *Connect Four, Checkers* and *Othello*. CP_{uct} competed against the two aforementioned baseline players and they also competed against each other.

5.1.1 Experiment Setup

The improvements from using UCT were measured by playing tournaments between CP_{uct} and the two baseline players, MC_{org} and MC_{mem}. Each pair of players played 250-game matches, 125 games as each color. In total, nine such matches were played, three for each type of game with three different time controls. For each game, tournaments with 10, 20, and 30 second start and play clocks were played. The GDL for the games used were downloaded from `http://games.stanford.edu` (conn4.kif, checkers.kif, and othello.kif).

The rows in the results tables in the following sections show the winning percentage of each player against each of the other two. The *Win ratio* column shows the average win percentage and the last column shows the 95% confidence bound.

All experiments were run on Linux 2.6.19-gentoo-r1 running on computers with two x86 Intel(R) Xeon(TM) 3.20GHz CPUs. Each game uses a single CPU.

5.1.2 Connect Four

The result of the three Connect Four tournaments can be seen in Table 5.1. Clearly CADIA-Player is the superior player with an average winning percent ranging from 88.7% in the 10 second game, to 91.4% in the 30 seconds game. If we look at the outcome of the 10 seconds game, we can see that having the memory gives MC_{mem} a big advantage over MC_{org}, but this advantage lessens quickly as the thinking time is increased. The action selection of CP_{uct} holds up well and when having the memory seems to have lost the edge MC_{mem} drops considerably in relation with MC_{org}, but CP_{uct} does not. It constantly improves more than MC_{mem} but the numbers against MC_{org} are more stable and could indicate that the model starts to make little difference when given enough time to run many simulations as Connect Four is a relatively simply game, although its state space is estimated to be about 10^{14} (Allis, 1994). The simulation count without the model is probably achieving a similar ranking of the actions. In the 30 second game CP_{uct} even

Table 5.1: Connect four results

Player	MC_{org}	MC_{mem}	CP_{uct}	Win ratio	95% conf.
Start and play clock set to 10 seconds					
MC_{org}	N/A	29.0 %	6.8 %	17.9 %	± 3.33 %
MC_{mem}	71.0 %	N/A	15.8 %	43.4 %	± 4.31 %
CP_{uct}	93.2 %	84.2 %	N/A	88.7 %	± 2.73 %
Start and play clock set to 20 seconds					
MC_{org}	N/A	43.4 %	10.2 %	26.8 %	± 3.83 %
MC_{mem}	56.6 %	N/A	12.4 %	34.5 %	± 4.11 %
CP_{uct}	89.8 %	87.6 %	N/A	88.7 %	± 2.73 %
Start and play clock set to 30 seconds					
MC_{org}	N/A	41.8 %	8.8 %	25.3 %	± 3.71 %
MC_{mem}	58.2 %	N/A	8.4 %	33.3 %	± 4.03 %
CP_{uct}	91.2 %	91.6 %	N/A	91.4 %	± 2.35 %

gets a higher winning percentage against MC_{mem}. This can be explained by the variance because of the limited sample size.

5.1.3 Checkers

When we examine the tournament results from Checkers, which are found in Table 5.2, we see more uniform results than in Connect Four. This is expected as Checkers is bigger with a state space estimated to be 10^{18} (Allis, 1994), and needs much more complex strategy. The game tree can even contain loops when both players have promoted a piece to a king. The memory structure accounts for most of the improvements, but the UCT action selection of CP_{uct} still shows an definite advantage over MC_{mem}, winning about 70% of the games against it on average. Also, notice that MC_{org} is constantly by far the worst player, showing the advantage of having a memory model. The reason why the memory model helps so much in this game might be partly explained by the small branching factor because of forced captures. This helps the model to keep a bigger share of relevant information in memory between moves, even with random action selection.

5.1.4 Othello

Othello has a huge state space, about 10^{28} (Allis, 1994), but loops cannot occur, i.e. every move brings the game closer towards the end. As seen in Table 5.3, the size of the state space affects the accuracy of the model more than in the other two games and MC_{org} is relatively not as bad as in Checkers. But what is interesting is that CP_{uct} seems not to be

Table 5.2: Checkers results

Player	**MC**$_{org}$	**MC**$_{mem}$	**CP**$_{uct}$	**Win ratio**	**95% conf.**
Start and play clock set to 10 seconds					
MC$_{org}$	*N/A*	14.6 %	6.8 %	10.7 %	± 2.51 %
MC$_{mem}$	85.4 %	*N/A*	31.2 %	58.3 %	± 4.12 %
CP$_{uct}$	93.2 %	68.8 %	*N/A*	81.0 %	± 3.22 %
Start and play clock set to 20 seconds					
MC$_{org}$	*N/A*	17.4 %	4.6 %	11.0 %	± 2.53 %
MC$_{mem}$	82.6 %	*N/A*	28.0 %	55.3 %	± 4.13 %
CP$_{uct}$	95.4 %	72.0 %	*N/A*	83.7 %	± 2.97 %
Start and play clock set to 30 seconds					
MC$_{org}$	*N/A*	15.4 %	8.2 %	11.8 %	± 2.57 %
MC$_{mem}$	84.6 %	*N/A*	28.6 %	56.6 %	± 4.10 %
CP$_{uct}$	91.8 %	71.4 %	*N/A*	81.6 %	± 3.07 %

Table 5.3: Othello results

Player	**MC**$_{org}$	**MC**$_{mem}$	**CP**$_{uct}$	**Win ratio**	**95% conf.**
Start and play clock set to 10 seconds					
MC$_{org}$	*N/A*	39.8 %	23.2 %	31.5 %	± 4.00 %
MC$_{mem}$	60.2 %	*N/A*	26.6 %	43.4 %	± 4.28 %
CP$_{uct}$	76.8 %	73.4 %	*N/A*	75.1 %	± 3.71 %
Start and play clock set to 20 seconds					
MC$_{org}$	*N/A*	33.2 %	16.0 %	24.6 %	± 3.75 %
MC$_{mem}$	66.8 %	*N/A*	29.4 %	48.1 %	± 4.36 %
CP$_{uct}$	84.0 %	70.6 %	*N/A*	77.3 %	± 3.64 %
Start and play clock set to 30 seconds					
MC$_{org}$	*N/A*	35.0 %	16.0 %	25.5 %	± 3.78 %
MC$_{mem}$	65.0 %	*N/A*	26.2 %	45.6 %	± 4.33 %
CP$_{uct}$	84.0 %	73.8 %	*N/A*	78.9 %	± 3.53 %

as affected as MC$_{mem}$ by the larger state space when playing MC$_{org}$ and, even though not by much, does better against MC$_{mem}$ than in Checkers. Here the UCT algorithm earns its keep and obviously has an advantage from how it balances exploration and exploitation.

5.2 Node Expansion

The result of the Checkers match shows the players with the memory model dominating MC$_{org}$. Therefore we wanted to get a better idea of how much effect the model has on node expansion. The model does in many cases bypass expensive action lookup in YAP

Table 5.4: Node expansion count for CP_{uct} and baseline players

Player	**Nodes expanded per second**		
	Connect Four	**Checkers**	**Othello**
MC_{org}	536.96	211.04	102.56
MC_{mem}	1124.89	484.58	134.38
CP_{uct}	1060.30	478.58	135.31

by using cashed information. Doing so should increase the number of nodes expanded per second. So to see how much impact adding the memory has on node expansion, and what the overhead is by using the UCT algorithm in CP_{uct} we ran experiments to measure how many nodes each player expanded per second.

5.2.1 Experiment Setup

Each of the players was run against MC_{org} in three separate matches with 10, 20 and 30 second start and play clocks for each of the three test games. Then number of nodes expanded for all move decisions during these games was averaged to get the average over any position in the game tree.

All experiments were run on Linux 2.6.19-gentoo-r1 running on computers with two x86 Intel(R) Xeon(TM) 3.20GHz CPUs. Each match used one CPU.

5.2.2 Node Expansion Results

The results of the node expansion experiments are listed in Table 5.4. There we see clearly that the game-tree model helps achieve faster node expansions and that is the result of caching what moves are available in states that are added to the model (see Section 3.2). This way they can be looked up when the state is encountered again instead of having to let YAP prove them which is costly. Expanding more nodes means more simulations which leads to a more reliable value function, so not only does the model give an advantage in being able to remember simulation information between moves, but also by speeding up the player significantly. In the case of these games, the complexity of proving moves and transitions rises as the state space gets larger, resulting in lower node expansion performance.

The overhead of using the UCT algorithm is notable in the figures for Connect Four and Checkers but nothing to worry about, given its clear dominance when competing with the other two. But the interesting thing is that in the huge state space of Othello, it actually

achieves more node expansions than MC_{mem}. This is an artifact of how it searches the state space (see Chapter 4), because by concentrating on what is likely to pay off, number of revisits increases.

5.3 Time Control Comparison

To see how more simulations will affect the performance of our UCT player, we ran experiments where one player had double the time controls of the other. This is also a good test for forecasting the benefits of using faster future hardware.

5.3.1 Experiment Setup

Two CP_{uct} players competed in the three aforementioned test games using three different time controls, with the second player always having half the amount of time the first one had. The time controls of both start and play clock for the first player were set to 10, 20, and 40 seconds and for each of this setting 250 games were played (125 each side).

The *Win ratio* column in Table 5.5 shows the winning percentage of the player with the higher time controls. The last column shows the 95% confidence bound.

All experiments were run on Linux 2.6.19-gentoo-r1 running on computers with two x86 Intel(R) Xeon(TM) 3.20GHz CPUs. Each game uses a single CPU.

5.3.2 Time Control Comparison Results

The performance within each game show constant gain and any difference can be explained by the variance. Not showing any signs of diminishing returns, as time is increased is a positive result for the simulation approach and indicates that this approach will continue to gain momentum with new technology with even faster multi-core CPUs.

5.4 Improved CADIA-Player

The following experiment evaluates the enhancement of selecting unexplored actions from a Gibbs distribution applied to history heuristic values (see Section 4.5). Is shows us if it has a statistically sigificant impact on the original CADIA-Player.

Table 5.5: Time control comparison for CP_{uct}

Game	Time controls		Win ratio	95% conf.
Connect Four	10 sec	05 sec	64.2 %	± 5.81 %
Connect Four	20 sec	10 sec	63.2 %	± 5.83 %
Connect Four	40 sec	20 sec	65.4 %	± 5.79 %
Checkers	10 sec	05 sec	76.2 %	± 4.85 %
Checkers	20 sec	10 sec	72.2 %	± 4.96 %
Checkers	40 sec	20 sec	77.8 %	± 4.33 %
Othello	10 sec	05 sec	67.0 %	± 5.75 %
Othello	20 sec	10 sec	64.0 %	± 5.86 %
Othello	40 sec	20 sec	69.0 %	± 5.68 %

5.4.1 Experiment Setup

We set up a tournament between the original CP_{uct} and CP_{imp}. They competed in four different games, the three test games mentioned earlier plus the game *Amazons* with both clocks at 10, 20, and 30 seconds and each player as one side half of the time. As before, for every game and clock setting, 250 games were played. *Amazons* was added to the testbed because we where curious of what would happen given its large branching factor (typically many hundreds of moves).

The temperature (see Section 4.5) of the Gibbs distribution was set to 10. This value was empirically obtained by trying a range of temperature values on a small number of games to be able to map out an appropriate setting for proof of concept. This value is not exact and we do not claim that it should be the same for all games.

All experiments were run on Linux 2.6.19-gentoo-r1 running on computers with two x86 Intel(R) Xeon(TM) 3.20GHz CPUs or two x86 Intel(R) Xeon(TM) 3GHz CPUs. Each experiment used one CPU that the players took turn using so in all games both participants used exactly the same hardware.

5.4.2 Improved CADIA-Player Results

A summary of the results is presented in Table 5.6. The third column of the table contains the winning percentage of CP_{imp} and the fourth column the 95% confidence bound. The last column shows the rounded down confidence for improvement using one tailed test. The table shows us that CP_{imp} wins each and every 250-game match, all games and all time controls.

Table 5.6: Tournament winning percentage for CP_{uct} and CP_{imp}

Game	Time	CP_{imp}	95% conf.	Improvement conf.
Connect Four	10 sec	53.6 %	± 6.09 %	>87 %
Connect Four	20 sec	52.8 %	± 6.00 %	>81 %
Connect Four	30 sec	54.2 %	± 6.08 %	>91 %
Checkers	10 sec	54.0 %	± 5.80 %	>91 %
Checkers	20 sec	54.4 %	± 5.80 %	>93 %
Checkers	30 sec	54.4 %	± 5.77 %	>93 %
Othello	10 sec	56.4 %	± 6.08 %	>97 %
Othello	20 sec	65.2 %	± 5.81 %	>99 %
Othello	30 sec	65.0 %	± 5.83 %	>99 %
Amazons	10 sec	58.8 %	± 6.11 %	>99 %
Amazons	20 sec	56.8 %	± 6.15 %	>98 %
Amazons	30 sec	52.8 %	± 6.20 %	>81 %
Average Winnings		**56.53 %**		

In Figure 5.1 we see the numbers from Table 5.6 as a line chart sharing the win ratio of CP_{imp} in the games over the different time controls. We see that for both Connect Four and Checkers there seems to be an constant level of improvement regardless of the clock setting. Most noticeable is how well this improvement works for Othello. This is most likely because of how stationary good moves are in that game. Putting a tile in one of the corners or on the sides is usually when possible a good move when played in many different game positions. In Amazons a move can become just as bad as it was good if one of your Amazons is close to a certain threatening square instead of one of your opponent's. Also if the opponent manages to move an Amazon from an imminent threat, the threatening move becomes a waste of time. The most likely reason for why the improvement works so well for the 10 seconds game of Amazons is that the complexity of the game is greater than the original player can handle with such a short time to think. In other words, it does not have the sense to make the moves that disarm good moves for CP_{imp}. Given more time on the clocks, CP_{uct} quickly catches up though.

There is no noticeable correlation between the branching factor and the level of improvement, it has more to do with the characteristics of the game.

For the individual 250-game matches we can state for almost half of them with over 95% statistical significance that CP_{imp} is the better player (and the remaining with over 90% significance for all except three). If we accumulate results based on all matches for a game on the one hand, and on the other hand all matches of a given time control, we can state with close to 99% significance that CP_{imp} is better at all games and all time controls.

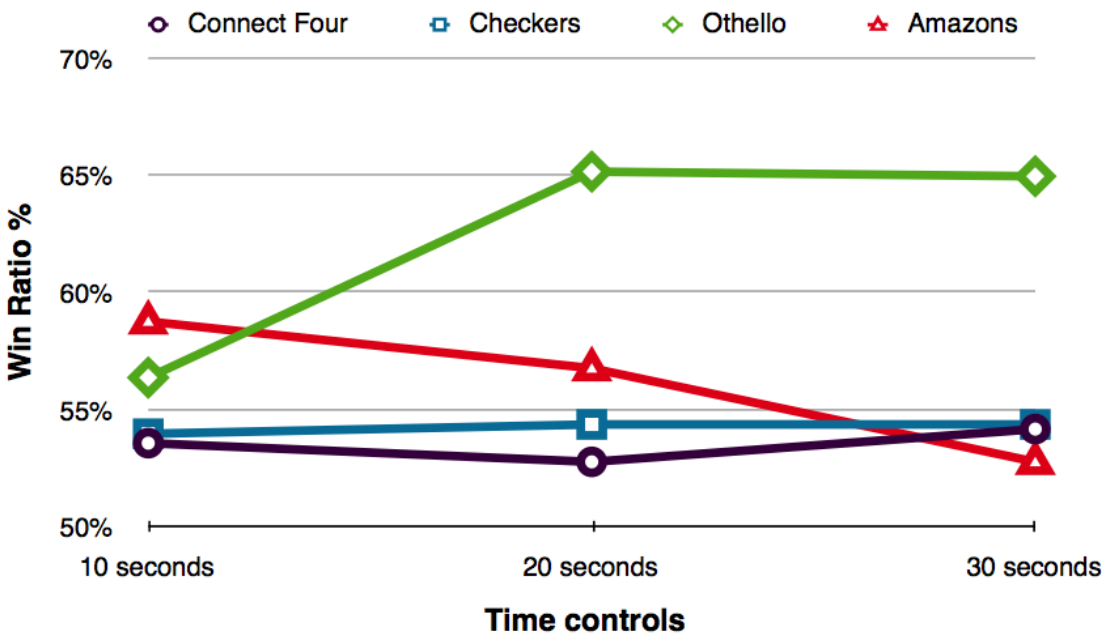

Figure 5.1: Improvements of CP_{imp}

5.5 Conclusions

In Connect Four the CP_{uct} variant dominated the others. It showed how simple games can overcome the benefits of having the model, but still the UCT action selection gives an unmistakable advantage. In Checkers we saw how useful the model can be when a large portion of the game tree it models is still relevant after a move (because of a very small branching factor). Othello showed how the UCT action selection handles huge state spaces much better than random selection. UCT selective nature allows it to verify promising paths while the random selection is likely to get similar values for all moves because of the small size of samples it has for each of them. Overall, the CP_{uct} gave impressive results, demonstrating the strength of UCT.

Besides keeping information, a part of the benefits of adding the game-tree model is that it can more than double average speed. The model also gets rid of the overhead of UCT for very large state spaces as it uses cached information frequently to assert promising paths.

Each time the time controls of the CP_{uct} player are doubled we maintain a constant level of improvement, indicating constant benefits from faster hardware or more robust processing of the state space.

By enhancing UCT with the history heuristic we get a player that won every 250-game test match against CP_{uct}. By accumulating all matches of any game type or time controls

we can state that the new player, CP_{imp}, is better with over 95% statistical significance and if we look at all matches for all games and time controls we get over 99% significance. CP_{imp} level of improvement in each game seems to be connected to the characteristics of the game: mainly how likely an action that is good in one game position is likely to be good in different positions. As far as we have observed, the branching factor does not affect this enhancement.

Chapter 6

Conclusions

In this thesis we presented a complete GGP agent named CADIA-Player. It has already proven its effectiveness by taking first place in the Third Annual General Game Playing Competition. By this it demonstrated the effectiveness of simulation-based approaches in the context of GGP.

We described the architecture of CADIA-Player and explained how it works and by doing so we gave insight into one promising method of how a General Game Player can be built. The main focus of CADIA-Player is adversary games, and it uses the UCT algorithm for most of its game playing. The only exception is for single-player games where it uses Enhanced IDA* by default. CADIA-Player models the game it is playing and conserves memory by adding only one state per simulation to the game-tree model, which in conjunction with the UCT algorithm causes memory usage to focus mostly on the "interesting" branches of the game tree. We also discussed how CADIA-Player is parallelized to use many CPUs. One appeal of our approach to GGP is how easily it can be parallelized. This is an important feature as massively parallel multi-core processors are now becoming increasingly more mainstream.

In Chapter 5 we gave empirical data showing how CADIA-Player improves in adversary games when given its game-tree model and the UCT algorithm for action selection during simulation. We compared it to two Monte Carlo players, one without the game-tree model and one with it. By separating these two aspects we can more clearly see the impact of both adding the game tree and the UCT action selection rule. As the complexity level of the game increases, the overhead of UCT disappears. This is because the search now spends more time in interesting parts of the game tree that have already been explored and therefore cached in the memory model. We examined how better hardware would affect CADIA-Player and found no signs of diminishing returns. For the improved version of

CADIA-Player we give data that confirms that its performance over the original one is statistically significant. The improvements appear to be linked to the characteristics of the game being played, mainly of how persistent good moves are.

For future work there are some open issues that we have observed while developing CADIA-Player. For example, the C_p parameter of the UCT algorithm needs to be investigated further. First of all, it is unlikely that it should be constant for all GGP games and only connected to the goal value range. Instead one should dynamically determine its most appropriate value based on the game description. It may even be that the C_p parameter should change during a game. For the improved CADIA-Player there are similar questions that need to be answered for the temperature parameter. Also, the value that unexplored actions with no historical data are initialized with needs to be investigated further. These are all smaller questions regarding our player. There are grander challenges ahead.

General game-playing systems are still in their infancy. There are still many interesting research topics that need to be addressed to further advance the field. CADIA-Player, with its simulation approach, adds a new promising look at one such avenue. We expect to see many more similar approaches in future GGP competitions. Search is knowledge, and is capable of discovering useful patterns dynamically that would be hard to do with a static evaluation approach. One of the main future challenges in our view is how to effectively combine traditional knowledge-based approaches with the new simulation-based approaches.

Bibliography

Allis, L. V. (1994). *Searching for Solutions in Games and Artificial Intelligence*. PhD dissertation, University of Limburg, Computer Science Department.

Auer, P., Cesa-Bianchi, N., & Fischer, P. (2002). Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, *47*(2/3), 235–256.

Banerjee, B., Kuhlmann, G., & Stone, P. (2006, June). Value function transfer for General Game Playing. In *ICML workshop on structural knowledge transfer for machine learning.*

Banerjee, B., & Stone, P. (2007, January). General game learning using knowledge transfer. In *The 20th international joint conference on artificial intelligence* (pp. 672–677).

Cazenave, T., & Jouandeau, N. (2007, June). On the parallelization of UCT. In *Proceedings of the Computer Games Workshop (CGW2007)* (p. 93-101).

Clune, J. (2007). Heuristic evaluation functions for General Game Playing. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence* (p. 1134-1139). AAAI Press.

Coulom, R. (2006). Efficient selectivity and backup operators in Monte-Carlo tree search. In *The 5th International Conference on Computers and Games (CG2006)* (p. 72-83).

Coulom, R. (2007). Computing Elo ratings of move patterns in the game of Go. In *Computer Games Workshop.* Amsterdam, The Netherlands.

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

Gelly, S., & Silver, D. (2007). Combining online and offline knowledge in UCT. In Z. Ghahramani (Ed.), *ICML* (Vol. 227, p. 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.

General Game Playing Project. (n.d.). The General Game Playing Project Web site: http://games.stanford.edu.

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.

Kocsis, L., & Szepesvári, C. (2006). Bandit based Monte-Carlo planning. In *European Conference on Machine Learning (ECML)* (p. 282-293).

Korf, R. E. (1985). Iterative-Deepening-A*: An optimal admissible tree search. In *IJCAI* (p. 1034-1036).

Kuhlmann, G., Dresner, K., & Stone, P. (2006, July). Automatic heuristic construction in a complete general game player. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence* (pp. 1457–62).

Love, N., Hinrichs, T., & Genesereth, M. (2006). *General Game Playing: Game description language specification* (Technical Report No. April 4 2006). Stanford University.

Luckhart, C., & Irani, K. B. (1986). An algorithmic solution of n-person games. In *AAAI* (p. 158-162).

Pell, B. (1996). A strategic metagame player for general chess-like games. *Computational Intelligence*, *12*, 177-198.

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

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.

Schiffel, S., & Thielscher, M. (2007a). Automatic construction of a heuristic search function for General Game Playing. In *Seventh IJCAI International Workshop on Nonmontonic Reasoning, Action and Change (NRAC07).*

Schiffel, S., & Thielscher, M. (2007b). Fluxplayer: A successful general game player. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence* (p. 1191-1196). AAAI Press.

Sherstov, A. A., & Stone, P. (2005, July). Improving action selection in MDP's via knowledge transfer. In *Proceedings of the Twentieth National Conference on Artificial Intelligence* (p. 1024-1029).

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 innovative applications of artificial intelligence* (pp. 201–207). AAAI Press / The MIT Press.

Sutton, R., & Barto, A. (1998). *Reinforcement Learning: An introduction*. MIT press.

Taylor, M. E., Whiteson, S., & Stone, P. (2006, June). Transfer learning for policy search methods. In *ICML workshop on Structural Knowledge Transfer for Machine Learning*.

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 Example

The KIF description of Tic-Tac-Toe is provided as an example of a game description in GDL. The atoms that start with the character "?" are variables.

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

GGP Competition 2007

The preliminaries of the GGP Competition 2007 took place in the month of June and spanned four weeks. Following are the final results of the preliminaires.

Table B.1: Results of the Third Annual GGP Competition preliminaries

Rank	Player	Total Points	Institution
1	CADIA-Player	2723.50	Reykjavik University
2	Fluxplayer	2355.50	Technical University of Dresden
3	Ary	2252.75	University of Paris 8
4	ClunePlayer	2122.25	University of California, LA
5	UTexas LARG	1798.00	University of Texas, Austin
6	Jigsawbot	1524.00	India Institute of Technology
7	LuckyLemming	1250.50	Technical University of Dresden
8	WWolfe	821.25	Independent (Stanford student)

The finals took place at the AAAI conference in Vancouver in July 2007. They were in knock-out elimination format. CADIA-Player also won that competition, defeating ClunePlayer in the finals.

Following are short descriptions for all games played in the preliminaries as given by the Stanford Logic Group. The game played in the semi finals were Breakthrough (week 3) and three variations of Skirmish (week 4) were played in the final matches against Cluneplayer.

Week 1

Blocks-World : Single-player, easy puzzle.

Maze : Single-player.

8-puzzle : Single-player.

Peg : Single-player, performance-correlated reward (goal is proportional to number of pegs removed from board).

Blocker : Two-player, asymmetric roles, turn taking, zero-sum reward.

Tic Tac Toe : Two-player, asymmetric roles, turn taking, zero-sum reward.

Tic Tic Toe : Two-player, symmetric roles, simultaneous move, zero-sum reward.

Two Player Chinese Checkers : two-player, nearly symmetric roles, turn taking, performance-correlated reward (goal is proportional to number of pegs that a player moves to the other end of the board).

Four Player Chinese Checkers : four-player, nearly symmetric roles, turn taking, performance-correlated reward (goal is proportional to number of pegs that a player moves to the other end of the board).

Asteroids : Single-player, performance-correlated reward (goal is 50 if a player is able to stop the ship from moving, 100 if he can do so in a particular location).

Beat-Mania : Single-player, performance-correlated reward (goal is proportional to the number of blocks caught).

Mummy Maze : Two-player, asymmetric roles, turn taking, zero-sum reward.

Ghost Maze : Two-player, asymmetric roles, turn taking, zero-sum reward.

Pac Man : Three-player, asymmetric roles, turn taking, asymmetric reward profile (reward for pacman are performance correlated - goal is proportional to number of pellets eaten, reward for ghosts is zero sum, 100 if pacman is caught, 0 otherwise).

Week 2

Blocks-World (Parallel) : Two instances of Blocks-World from Week 1, played at the same time. Neither interacts with the other. Single Player. Averaged zero-sum rewards.

Blocks-World (Serial) : Same as above except the two instances are played one after the other.

Asteroids (Parallel) : Two instances of Asteroids from Week 1, played at the same time. Neither interacts with the other. Single Player. Averaged zero-sum rewards.

Asteroids (Serial) : Same as above except the two instances are played one after the other.

Tic-Tac-Toe (Parallel) : Two instances of Tic-Tac-Toe from Week 1, played at the same time. Neither interacts with the other. Two Player, turn-taking. Averaged zero-sum rewards.

Tic-Tac-Toe (Serial) : Same as above except the two instances are played one after the other.

Blocker (Parallel) : Two instances of Blocker from Week 1, played at the same time. Neither interacts with the other. Two Player, asymmetric roles, simultaneous move. Averaged zero-sum rewards.

Blocker (Serial) : Same as above except the two instances are played one after the other.

Rule-Depth (Linear) : A stress test where it is always legal to either give up or continue. The amount of effort to prove it is legal to continue grows linearly with game length. Single Player, performance-correlated reward (proportional to how long a player goes before giving up).

Rule-Depth (Quadratic) : A stress test where it is always legal to either give up or continue. The amount of effort to prove it is legal to continue grows quadratically with game length. Single Player, performance-correlated reward (proportional to how long a player goes before giving up).

Rule-Depth (Exponential) : A stress test where it is always legal to either give up or continue. The amount of effort to prove it is legal to continue grows exponentially with game length. Single Player, performance-correlated reward (proportional to how long a player goes before giving up).

Duplicate-State (Small) : A stress test based on tree search. Of 1000 nodes, only 5 are unique. Single Player, performance-correlated reward (based on path taken through tree.)

Duplicate-State (Medium) : A stress test based on tree search. Of 1000000 nodes, only 10 are unique. Single Player, performance-correlated reward (based on path taken through tree.)

Duplicate-State (Large) : A stress test based on tree search. Of 100000000 nodes, only 15 are unique. Single Player, performance-correlated reward (based on path taken through tree.)

State-Space (Small) : A stress test based on the search of a tree containing 1000 nodes. Single Player, performance-correlated reward (based on path taken through tree.)

State-Space (Medium) : A stress test based on the search of a tree containing 1000000 nodes. Single Player, performance-correlated reward (based on path taken through tree.)

State-Space (Large) : A stress test based on the search of a tree containing 1000000000 nodes. Single Player, performance-correlated reward (based on path taken through tree.)

Week 3

Large Tic-Tac-Toe : Tic-Tac-Toe played on a 5x5 grid. Two player, turn-taking. Zero- sum rewards.

Large Tic-Tac-Toe (Suicide) : Large Tic-Tac-Toe with inverted goals. Two player, turn-taking. Zero-sum rewards.

Connect Four : Players win by forming a line of four or more of their pieces. Two player, turn-taking. Zero-sum rewards.

Connect Four (Suicide) : Connect Four with inverted goals. Two player, turn-taking. Zero- sum rewards.

Bomberman : Players attempt to blow up their opponent by placing bombs, while at the same time avoiding like attacks. Two player, simultaneous move, symmetric roles. Zero-sum rewards.

Quarto : Players win by forming a line of four tiles in which each tile shares one of four attributes. Two player, turn-taking. Zero-sum rewards.

Othello : Players attempt to fill a board with more of their own pieces than their opponent's. Two player, turn-taking.

Breakthrough : Players win by moving one of their pieces to the other end of the board. Two player, turn-taking. Zero-sum rewards. Ties are impossible.

Breakthrough (Suicide) : Breakthrough with inverted goals. Two player, turn-taking. Zero-sum rewards. Ties are impossible.

Week 4

Pentago : Players win by forming a line of five of their pieces. Players can both place pieces and rotate portions of the board. Two player, turn-taking. Positive-sum rewards.

Pentago (Suicide) : Pentago with inverted goals. Two player, turn-taking. Positive-sum rewards.

Amazons : Players win by preventing their opponent from making legal moves. Two player, turn-taking. Zero-sum rewards. Ties are impossible.

Skirmish : Chess with modified rules. Players accumulate points by capturing pieces. Two player, turn-taking. Positive-sum rewards.

Checkers : An important game in the history of AI. Players win by capturing all of their opponent's pieces. Two player, turn-taking. Zero-sum rewards.

Wargame : Military simulation game. Players attempt to capture a flag while avoiding and repelling terrorists. Single player, Zero-sum rewards.

REYKJAVÍK UNIVERSITY

HÁSKÓLINN Í REYKJAVÍK

School of Computer Science
Reykjavík University
Kringlan 1, IS-103 Reykjavík, Iceland
Tel: +354 599 6200
Fax: +354 599 6301
http://www.ru.is