Yngvi Björnsson
Michael Thielscher (Eds.)

The IJCAI-13 Workshop on General Game Playing

General Intelligence in Game-Playing Agents, GIGA'13

Beijing, China, August 2013
Proceedings

IJCAI 2013 WORKSHOPS
August 3-5, 2013, Beijing, China

3rd International General Game Playing Workshop

Preface

Artificial Intelligence (AI) researchers have for decades worked on building game-playing agents capable of matching wits with the strongest humans in the world, resulting in several success stories for games like chess and checkers. The success of such systems has been partly due to years of relentless knowledge-engineering effort on behalf of the program developers, manually adding application-dependent knowledge to their game-playing agents. The various algorithmic enhancements used are often highly tailored towards the game at hand.

Research into general game playing (GGP) aims at taking this approach to the next level: to build 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. In contrast to software systems designed to play one specific game, systems capable of playing arbitrary unseen games cannot be provided with game-specific domain knowledge a priori. Instead, they must be endowed with high-level abilities to learn strategies and perform abstract reasoning. Successful realization of such programs poses many interesting research challenges for a wide variety of artificial-intelligence sub-areas including (but not limited to):

- knowledge representation and reasoning,
- heuristic search and automated planning,
- computational game-theory,
- multi-agent systems,
- machine learning,
- game playing and design,
- artificial general intelligence,
- opponent modeling,
- evaluation and analysis.

These are the proceedings of GIGA'13, the third ever workshop on General Intelligence in Game-Playing Agents following the inaugural GIGA Workshop at IJCAI'09 in Pasadena (USA) and the follow-up event at IJCAI'11 in Barcelona (Spain). This workshop series has been established to become the major forum for discussing, presenting and promoting research on General Game Playing. It is intended to bring together researchers from the above sub-fields of AI to discuss how best to address the challenges and further advance the state-of-the-art of general game-playing systems and generic artificial intelligence.

These proceedings contain the 9 papers that have been selected for presentation at this workshop. All submissions were reviewed by a distinguished international program committee. The accepted papers cover a multitude of topics such as the fast inference for game descriptions, advanced simulation-based methods, general imperfect-information game playing, and automated reasoning about games.

For the first time ever, GIGA'13 proudly presents the award for the Best Student-Only Paper, which comes with a free registration for the presenting author. We congratulate *Michael Schofield* and *Abdallah Saffidine* on winning this inaugural award with their contribution entitled "High Speed Forward Chaining for General Game Playing."

We thank all the authors for responding to the call for papers with their high quality submissions, and the program committee members and other reviewers for their valuable feedback and comments. We also thank IJCAI for all their help and support.

We welcome all our delegates and hope that all will enjoy the workshop and through it find inspiration for continuing their work on the many facets of General Game Playing!

August 2013

Yngvi Björnsson
Michael Thielscher

Organization

Workshop Chairs

Yngvi Björnsson, Reykjavík University, Iceland
Michael Thielscher, The University of New South Wales, Australia

Program Committee

Yngvi	Björnsson	Reykjavík University
Tristan	Cazenave	Université Paris-Dauphine
Stefan	Edelkamp	University of Bremen
Hilmar	Finnsson	Reykjavík University
Michael	Genesereth	Stanford University
Lukasz	Kaiser	Université Paris Diderot
Gregory	Kuhlmann	Apple Inc.
Abdallah	Saffidine	Université Paris-Dauphine
Marius	Schneider	University of Potsdam
Stephan	Schiffel	Reykjavík University
Sam	Schreiber	Google Inc.
Nathan	Sturtevant	University of Denver
Mark	Winands	Maastricht University

Additional Reviewer

Michael Schofield

Table of Contents

A Legal Player for GDL-II Based on Filtering With Logic Programs 7
Michael Thielscher

Sufficiency-Based Selection Strategy for MCTS 15
Stefan Freyr Gudmundsson, Yngvi Bjornsson

Decaying Simulation Strategies 23
M.J.W. Tak, Mark H. M. Winands, Yngvi Bjornsson

High Speed Forward Chaining for General Game Playing 31
Michael Schofield, Abdallah Saffidine

Lifting HyperPlay for General Game Playing to Incomplete-Information Models 39
Michael Schofield, Timothy Cerexhe, Michael Thielscher

Model Checking for Reasoning About Incomplete Information Games 47
Xiaowei Huang, Ji Ruan, Michael Thielscher

Comparison of GDL Reasoners 55
Yngvi Bjornsson, Stephan Schiffel

Online Adjustment of Tree Search for GGP 63
Jean Méhat, Jean Noel Vittaut

Stratified Logic Program Updates for General Game-Playing 71
David Spies

A Legal Player for GDL-II Based on Filtering With Logic Programs*

Michael Thielscher
School of Computer Science and Engineering
University of New South Wales
Australia
mit@cse.unsw.edu.au

Abstract

Motivated by the problem of building a basic reasoner for general game playing with imperfect information, we address the problem of filtering with logic programs, whereby an agent updates its incomplete knowledge of a program by observations. We develop a filtering method by adapting an existing backward-chaining and abduction method for so-called open logic programs. Experimental results show that this provides a basic effective and efficient "legal" player for general imperfect-information games.

Introduction

A general game-playing (GGP) system is one that can understand the rules of arbitrary games and use these rules to play effectively. The annual GGP competition at AAAI has been established in 2005 to foster research in this area [Genesereth *et al.*, 2005]. While the competition in the past has focused on games in which players always know the complete game state, a recent extension [Thielscher, 2011] of the formal game description language GDL also allows the description of general randomized games with imperfect and asymmetric information [Quenault and Cazenave, 2007].

GDL uses normal logic program clauses to describe the rules of a game [Genesereth *et al.*, 2005]. For games with perfect information, standard resolution techniques can be used to build a basic, so-called *legal player* that throughout a game always knows its allowed moves [Love *et al.*, 2006]. Efficient variations exist that use tailored data structures and algorithms for computing moves in classic GDL [Schkufza *et al.*, 2008; Waugh, 2009; Kissmann and Edelkamp, 2010; Saffidine and Cazenave, 2011]. But the generalization to imperfect-information games raises a fundamentally new reasoning challenge even for such a basic player. Computing with all possible states is practically infeasible except for very simple games [Parker *et al.*, 2005]. This is why the only two existing GGP systems described in the literature for imperfect-information GDL [Edelkamp *et al.*, 2012; Schofield *et al.*, 2012] use the more practical alternative of randomly sampling states [Richards and Amir, 2009; Silver and Veness, 2010]. But in so doing these players reason with a mere subset of all models, which is logically incorrect.

In this paper we address the problem of building a logically sound and efficient basic reasoning system for general imperfect-information games by first isolating and addressing the problem of *filtering with logic programs*: Suppose given a logic program with some hidden facts of which we have only partial knowledge. Suppose further that some consequences of this incomplete program can be observed. The question then is, what other conclusions can we derive from our limited knowledge plus the observations? This can be seen as an instance of the general notion of *filtering* as any process by which an agent updates its belief according to observations [Amir and Russell, 2003].

We develop a method for filtering with logic programs under the assumption that incomplete knowledge is represented by two sets containing, respectively, known and unknown atoms, in the sense of 3-valued logic [Kleene, 1952]. Adapting an inference method for abduction in so-called open logic programs [Bonatti, 2001a; 2001b], we show how a method for filtering can be obtained by augmenting standard backward-chaining with the computation of support.

We apply this method for filtering with logic programs to build a legal player for general game playing with imperfect information that, just like its counterpart for perfect-information games, is based on backward-chaining. We prove that the reasoner thus obtained is sound. We also show it to be complete if, as in perfect-information games, the player can observe all other players' moves. Experimental results with all imperfect-information games used at past GGP competitions demonstrate the effectiveness and efficiency of our method for a legal player to always know its allowed moves in almost all games. This in fact supports an argument that can be made for requiring that all games for competitions such as at AAAI be written so that basic backward-chaining is all that is needed to derive a player's legal moves. Interestingly, the experiments also revealed that in many existing game descriptions players are not given enough information to know the outcome after termination.

After the following brief summary of GDL, we define the

*This submission is a slightly extended version of a paper that has been accepted for AAAI'13.

problem of filtering with logic programs. We then develop a filtering method based on backward-chaining and abduction of support. We apply this to build a basic and logically sound legal player, present our experimental results, and conclude.

Background: GDL-II

The science of general game playing requires a formal language for specifying arbitrary games by a complete set of rules. The declarative Game Description Language (GDL) serves this purpose [Genesereth *et al.*, 2005]. It uses the syntax of normal logic programs [Lloyd, 1987] and is characterized by these special keywords:

`role(R)`	`R` is a player
`init(F)`	feature `F` holds in the initial position
`true(F)`	feature `F` holds in the current position
`legal(R,M)`	`R` has move `M` in the current position
`does(R,M)`	player `R` does move `M`
`next(F)`	feature `F` holds in the next position
`terminal`	the current position is terminal
`goal(R,V)`	player `R` gets payoff `V`
`distinct(X,Y)`	terms `X,Y` are syntactically different
`sees(R,P)`	player `R` is told `P` in the next position
`random`	the random player (aka. Nature)

Originally designed for games with complete information [Genesereth *et al.*, 2005], GDL has recently been extended to GDL-II (for: *GDL with incomplete/imperfect information*) by the last two keywords (`sees`, `random`) to describe arbitrary (finite) games with randomized moves and imperfect information [Thielscher, 2010].

Example 1 (Monty Hall) *The GDL-II rules in Fig. 1 formalize a game based on a popular problem where a car prize is hidden behind one of three doors, a goat behind the others, and where a candidate is given two chances to pick a door. The intuition behind the rules is as follows.*[1] *Line 1 introduces the players' names. Lines 3–4 define the features of the initial game state. The allowed moves are specified by the rules for* `legal`*: In step 1, Monty Hall decides where to place the car (lines 6–7) and, simultaneously, the candidate chooses a door (lines 13–14); in step 2, Monty Hall opens one of the other doors (lines 8–11) but not the one with a car behind it; finally, the candidate can either stick to the earlier choice (*`noop`*) or switch (lines 16–17). The candidate's only percepts are: the door opened by the host (line 19) and the location of the car at the end of the game (line 20). Monty Hall, on the other hand, sees all moves by the candidate (line 21). The remaining rules specify the state update (*`next`*), the conditions for the game to end (*`terminal`*), and the payoff for the players depending on whether the candidate picked the right door (*`goal`*).*

Formal Syntax and Semantics

In order to admit an unambiguous interpretation, GDL-II game descriptions must obey certain general syntactic restrictions. Specifically, a valid game description must be *stratified* [Apt *et al.*, 1987] and *allowed* [Lloyd and Topor, 1986]. Stratified logic programs are known to admit a specific *standard model* [Apt *et al.*, 1987], which equals its *unique stable model* [Gelfond and Lifschitz, 1988]. A further syntactic restriction ensures that only finitely many positive instances are true in this model; for details we must refer to [Love *et al.*, 2006] for space reasons. Finally, the special keywords are to be used as follows [Thielscher, 2010]:

- `role` only appears in the head of facts;
- `init` only appears as head of clauses and does not depend on any of `true`, `legal`, `does`, `next`, `sees`, `terminal`, `goal`;
- `true` only appears in the body of clauses;
- `does` only appears in the body of clauses and does not depend on any of `legal`, `terminal`, `goal`;
- `next` and `sees` only appear as head of clauses.

Under these restrictions, any valid GDL-II game description G determines a state transition system as follows.

To begin with, the derivable instances of `role(R)` define the players, and the initial state consists in the derivable instances of `init(F)`. In order to determine the legal moves of a player in any given state, this state has to be encoded first, using the keyword `true`: Let $S = \{f_1, \dots, f_n\}$ be a state (i.e., a finite set of ground terms over the signature of G), then G is extended by the n facts

$$S^{\texttt{true}} \stackrel{\text{def}}{=} \{\, \texttt{true}(f_1). \quad \dots \quad \texttt{true}(f_n).\} \tag{1}$$

Those instances of `legal(R,M)` that follow from $G \cup S^{\texttt{true}}$ define all legal moves `M` for player `R` in position S.

In the same way, the clauses with `terminal` and `goal(R,N)` in the head define, respectively, termination and goal values *relative* to the encoding of a given position.

Determining a position update and the percepts of the players requires the encoding of both the current position and a *joint move*. Specifically, let M denote that players $r_1, \dots, r_k$ take moves $m_1, \dots, m_k$, then

$$M^{\text{does}} \stackrel{\text{def}}{=} \{\, \texttt{does}(r_1, m_1). \quad \dots \quad \texttt{does}(r_k, m_k).\,\} \tag{2}$$

All instances of `next(F)` that follow from $G \cup M^{\text{does}} \cup S^{\texttt{true}}$ compose the updated position; likewise, the derivable instances of `sees(R,P)` describe what a player perceives when the given joint move is done in the given position. All this is summarized below, where "$\models$" denotes entailment wrt. the unique stable model of a stratified set of clauses.

Definition 1 *The* semantics *of a valid GDL-II game description G is the state transition system given by*

- $R = \{r \colon G \models \texttt{role}(r)\}$ *(player names);*
- $s_1 = \{f \colon G \models \texttt{init}(f)\}$ *(initial state);*
- $t = \{S \colon G \cup S^{\texttt{true}} \models \texttt{terminal}\}$ *(terminal states);*
- $l = \{(r, m, S) \colon G \cup S^{\texttt{true}} \models \texttt{legal}(r, m)\}$ *(legal moves);*
- $u(M, S) = \{f \colon G \cup M^{\text{does}} \cup S^{\texttt{true}} \models \texttt{next}(f)\}$ *(update);*
- $\mathcal{I} = \{(r, M, S, p) : G \cup M^{\text{does}} \cup S^{\texttt{true}} \models \texttt{sees}(r, p)\}$ *(players' percepts);*
- $g = \{(r, v, S) \colon G \cup S^{\texttt{true}} \models \texttt{goal}(r, v)\}$ *(goal values).*

[1] For the sake of readability, we write GDL in standard Prolog syntax instead of the prefix notation used at competitions.

```
1 role(monty).  role(candidate).
2
3 init(closed(1)).  init(closed(2)).  init(closed(3)).
4 init(step(1)).
5
6 legal(monty,hide_car(D))    :- true(step(1)),
7                                 true(closed(D)).
8 legal(monty,open_door(D))   :- true(step(2)),
9                                 true(closed(D)),
10                                not true(car(D)),
11                                not true(chosen(D)).
12 legal(monty,noop)           :- true(step(3)).
13 legal(candidate,choose(D))  :- true(step(1)),
14                                true(closed(D)).
15 legal(candidate,noop)       :- true(step(2)).
16 legal(candidate,noop)       :- true(step(3)).
17 legal(candidate,switch)     :- true(step(3)).
18
19 sees(candidate,D) :- does(monty,open_door(D)).
20 sees(candidate,D) :- true(step(3)), true(car(D)).
21 sees(monty,move(R,M)) :- does(R,M).
22 next(car(D))     :- does(monty,hide_car(D)).
23 next(car(D))     :- true(car(D)).
24 next(closed(D))  :- true(closed(D)),
25                     not does(monty,open_door(D)).
26 next(chosen(D))  :- does(candidate,choose(D)).
27 next(chosen(D))  :- true(chosen(D)),
28                     not does(candidate,switch).
29 next(chosen(D))  :- does(candidate,switch),
30                     true(closed(D)),
31                     not true(chosen(D)).
32
33 next(step(2)) :- true(step(1)).
34 next(step(3)) :- true(step(2)).
35 next(step(4)) :- true(step(3)).
36
37 terminal :- true(step(4)).
38
39 goal(candidate,100) :- true(chosen(D)), true(car(D)).
40 goal(candidate,  0) :- true(chosen(D)), not true(car(D)).
41 goal(monty,    100) :- true(chosen(D)), not true(car(D)).
42 goal(monty,      0) :- true(chosen(D)), true(card(D)).
```

Figure 1: A description of the Monty Hall game [Rosenhouse, 2009] adapted from [Schofield *et al.*, 2012].

GDL-II games are played using the following protocol.

1. All players receive the complete game description G.
2. Starting with s_1, in each state S each player $r \in R$ selects a legal move from $\{m : l(r,m,S)\}$. (The predefined role `random`, if present, chooses a legal move with uniform probability.)
3. The update function (synchronously) applies the joint move M to the current position, resulting in the new position $S' = u(M,S)$. Furthermore, the roles r receive their individual percepts $\{p \colon \mathcal{I}(r,M,S,p)\}$.
4. This continues until a terminal state is reached, and then the goal relation determines the result for all players.

Filtering with Logic Programs

The original game protocol for GDL [Love *et al.*, 2006] differs from the above in that players are automatically informed about each other's moves in every round. Since they start off with complete knowledge of the initial state, knowing all moves implies that players have complete state knowledge throughout a game because there never is uncertainty about the facts $S^{\texttt{true}} \cup M^{\texttt{does}}$ (c.f. (1), (2)) that together with the game rules determine everything a player needs to know about the current state (such as the allowed moves as the derivable instances of `legal(R,M)`) and the next one (as the set of derivable instances of `next(F)`). The syntactic restrictions for valid game descriptions ensure that all necessary derivations are finite, so that a basic reasoner for GDL can be built based on standard backward chaining [Genesereth *et al.*, 2005].

In case of GDL-II, however, the situation is very different. Although players also start off with complete knowledge of the initial state, they are not automatically informed about each other's moves. But with only partial knowledge of the set of facts $M^{\texttt{does}}$, players can no longer fully determine the derivable instances of `next(F)` through standard backward chaining. This in turn means that players also lack complete knowledge of the facts $S^{\texttt{true}}$ in later states, which are needed to determine the legal moves and other crucial properties such as termination and goal values.

Rather than getting to see each other's moves, after every round players receive percepts according to the rules for `sees(R,P)`. In other words, they are informed about certain consequences that follow from the game rules and the incompletely known facts $S^{\texttt{true}} \cup M^{\texttt{does}}$. Building a basic player for GDL-II that is logically sound therefore requires a method for reasoning about the consequences of a partially known logic program and for updating this incomplete knowledge according to observations being made. Hence, we first isolate and address the more general problem of *filtering with logic programs*.

Definition 2 *Consider a normal logic program P and two sets, $\mathcal{B}$ and $\mathcal{O}$, of ground atoms called* base relations *and* observation relations*, respectively. A* filter *is a function that maps any given $\Phi \subseteq 2^{\mathcal{B}}$ and $O \subseteq \mathcal{O}$ into a set Filter$[O](\Phi) \subseteq \Phi$. A* correct *filter is one that satisfies* [2]

$$\mathit{Filter}[O](\Phi) \supseteq \{B \in \Phi : \; P \cup B \models o \text{ for all } o \in O \text{ and } P \cup B \not\models o \text{ for all } o \in \mathcal{O} \setminus O\}$$

A filter is complete *if these two sets are equal.*

In this definition, incomplete knowledge about the base relations is represented by a set of possible models Φ. A correct filter retains all models in Φ that entail all observations.

Example 2 *Consider the logic program below, with base relations $\mathcal{B} = \{b(1), b(2)\}$ and $\mathcal{O} = \{\text{obs}\}$.*

```
a   :- b(X).
obs :- not a.
p   :- not a.
q   :- a.
```

Suppose $\Phi = 2^{\{b(1),b(2)\}}$, that is, nothing is known about the base relations. If complete, Filter$[\{\text{obs}\}](\Phi)$ equals $\{\emptyset\}$. It follows that if obs *is observed then, under the only model left after filtering, p is derivable and q is not.*

[2]The definition applies to any chosen entailment relation "$\models$."

Example 3 *Consider the GDL in Fig. 1 with the instances of* **true**`(F)` *and* **does**`(R,M)` *as base relations. Let Φ be such that all of* **true**`(closed(1))`, **true**`(closed(2))`, **true**`(closed(3))`, **true**`(chosen(1))`, **true**`(step(1))`, **does**`(candidate,noop)` *are true in all models in Φ, and let $\mathcal{O} = \{$***sees***`(candidate,2)`, **sees**`(candidate,3)`$\}$.*

*Suppose that we observe $O = \{$***sees***`(candidate,3)`$\}$, then* **does**`(monty,open(3))` *is true in all models resulting from a complete filter (cf. line 19 in Fig. 1), while* **does**`(monty,open(2))` *is false in each of them. It follows, for instance, that in all models remaining after filtering,* **next**`(closed(1))` *and* **next**`(closed(2))` *are derivable but not* **next**`(closed(3))` *(cf. line 24–25).*

A Basic Legal Player for GDL-II

In this section we present a method for constructing a reasoner for GDL-II based on a method for filtering that operates on a compact representation of incomplete information.

Representing Incomplete Information About Facts

Since explicitly maintaining the set of possible states is practically infeasible in most games, we base our approach to filtering on a coarser but practically feasible encoding using two sets of ground atoms, $\mathcal{B}^+ \subseteq \mathcal{B}$ and $\mathcal{B}^0 \subseteq \mathcal{B}$, which respectively contain the base relations that are *known* to be true and those that *may* be true. Any such pair that satisfies $\mathcal{B}^+ \cap \mathcal{B}^0 = \emptyset$ implicitly represents the set of models

$$\Phi_{\mathcal{B}^+,\mathcal{B}^0} \stackrel{\text{def}}{=} \{\mathcal{B}^+ \cup B : B \subseteq \mathcal{B}^0\} \tag{3}$$

This representation allows us to base our filtering method on a derivation mechanism that has been developed in the context of so-called open logic programs [Bonatti, 2001a].

Reasoning with Open Logic Programs

In the following we adapt some basic definitions and results from [Bonatti, 2001a; 2001b] to our setting. Our *open logic programs* are triples $\Omega = \langle P, \mathcal{B}^+, \mathcal{B}^0 \rangle$ where P is a normal logic program and $\mathcal{B}^+, \mathcal{B}^0$ are as above. A program P' is called an *extension*[3] of Ω if $P' = P \cup \mathcal{B}^+ \cup B$ for some $B \subseteq \mathcal{B}^0$. This gives rise to two modes of reasoning:

1. *Skeptical inference*: $\Omega \models^s \varphi$ iff all stable models of all extensions P' of Ω entail φ.
2. *Credulous inference*: $\Omega \models^c \varphi$ iff some stable model of some extension P' of Ω entails φ.

As observed in [Bonatti, 2001a], these two methods of inference are dual to each other: $\Omega \models^s \varphi$ iff $\Omega \not\models^c$ **not** φ and $\Omega \models^c \varphi$ iff $\Omega \not\models^s$ **not** φ. We also make use of the following concepts [Bonatti, 2001b]:

1. A *support* for a ground atom A is a query Q obtained by unfolding A in $P \cup \mathcal{B}^+$ until all the literals in Q either occur in $\mathcal{B}^0$ or are negative.
2. A *countersupport* for a ground atom A is a set of ground literals S such that each $L \in S$ is the complement of some literal belonging to a support of A; and conversely, each support of A contains a literal whose complement is in S.

In the following, for a set S of literals we denote by S^+ the set of positive atoms in S and by S^- the set of atoms that occur negated in S. A support is *consistent* iff $S^+ \cap S^- = \emptyset$.

A Backward-Chaining Proof Method

The definitions from above form the basis of a backward-chaining derivation procedure for computing answer substitutions θ along with supports for literals L wrt. an open program $\Omega = \langle P, \mathcal{B}^+, \mathcal{B}^0 \rangle$ using the following derivation rules.

1. If $L\theta \in \mathcal{B}^+$, return θ along with support $\emptyset$.
2. If $L\theta \in \mathcal{B}^0$, return θ along with support $\{L\theta\}$.
3. If $L = \neg A$ is a negative ground literal and $\mathcal{S}$ the set of computed supports for A, return the empty substitution along with a consistent set containing the complement of some literal from each element in $\mathcal{S}$.
4. If $L = A$ is positive and unifiable with the head of a clause from P, unfold A and return the union, if consistent, of supports for all literals in the resulting query along with the combined answer substitutions.

Recall, for instance, the short program from Example 2 and suppose $\mathcal{B}^+ = \emptyset$ and $\mathcal{B}^0 = \{b(1), b(2)\}$. Query $b(X)$ admits two computed supports: $S = \{b(1)\}$ with $\theta = \{X \setminus 1\}$, and $S = \{b(2)\}$ with $\theta = \{X \setminus 2\}$. Hence, the computed countersupport for query a is $\{\neg b(1), \neg b(2)\}$, which in turn is the (only) support for *obs* under the given sets $\mathcal{B}^+, \mathcal{B}^0$.

The above derivation rules are a subset of the calculus defined in [Bonatti, 2001a; 2001b] but constitute a complete and decidable derivation procedure if the underlying logic program is syntactically restricted.

Proposition 1 *Let $\Omega = \langle P, \mathcal{B}^+, \mathcal{B}^0 \rangle$ be an open logic program with a finite Herbrand base and stratified program P.*

1. *Every computed support θ, S for a query Q satisfies $\langle P, \mathcal{B}^+ \cup S^+, \mathcal{B}^0 \setminus S^- \rangle \models^s Q\theta$.*
2. *If $\langle P, \mathcal{B}^+, \mathcal{B}^0 \rangle \models^c Q\tau$ for some τ, then there exists a computed support θ, S for Q with θ more general than τ.*

In the following, by $\Omega \vdash_{\theta,S} A$ we denote that substitution θ together with some S is a computed support for atom A wrt. open logic program Ω. In particular, $\Omega \vdash_{\theta,\emptyset} A$ means that $A\theta$ follows without additional support, i.e., is necessarily true in all extensions and hence skeptically entailed by Ω.

Filtering Based on Backward Chaining

Next, we use the backward chaining-based method for open logic programs to define a basic method for filtering with logic programs. In the following, by $\mathtt{Supp}(Q)$ we denote the set of all computed supports for query Q. Consider a normal logic program P; sets $\mathcal{B}^+, \mathcal{B}^0$; and a set $O \subseteq \mathcal{O}$ of observations. We compute $\mathit{Filter}[O](\Phi_{\mathcal{B}^+,\mathcal{B}^0})$ as two sets $\mathcal{B}^+_{new}$ and $\mathcal{B}^0_{new}$ as follows.

$$\mathcal{B}^+_{new} = \mathcal{B}^+ \cup \left(\bigcup_{o \in O} \bigcap_{S \in \mathtt{Supp}(o)} S^+ \cup \bigcup_{o \in \mathcal{O} \setminus O} \bigcap_{S \in \mathtt{Supp}(\neg o)} S^+ \right)$$

[3]This is called a *completion* in [Bonatti, 2001a], which however clashes with another concept so named earlier [Shepherdson, 1984].

$$\mathcal{B}^0_{new} = (\mathcal{B}^0 \setminus \mathcal{B}^+_{new}) \setminus \left(\bigcup_{o \in O} \bigcap_{S \in \texttt{Supp}(o)} S^- \cup \bigcup_{o \in \mathcal{O} \setminus O} \bigcap_{S \in \texttt{Supp}(\neg o)} S^- \right)$$

Put in words, for each observation o made (resp. not made) we compute all supports (resp. all supports for $\neg o$) and then "strengthen" $\mathcal{B}^+, \mathcal{B}^0$ by every literal that is contained in all supports. More precisely, if a literal occurs positively in every support for some o (resp. $\neg o$), then it is added to $\mathcal{B}^+$ and removed from $\mathcal{B}^0$. Also removed from $\mathcal{B}^0$ are the literals that occur negatively in every support for some o (resp. $\neg o$).

Example 4 *Recall again the program from Example 2. As we have seen, when $\mathcal{B}^+ = \emptyset$ and $\mathcal{B}^0 = \{b(1), b(2)\}$ then the query* obs *has one support, namely, $\{\neg b(1), \neg b(2)\}$. This yields $\mathcal{B}^+_{new} = \emptyset$ and $\mathcal{B}^0_{new} = \emptyset$. On the other hand, consider the query* $\neg$obs. *It has two supports, $\{b(1)\}$ and $\{b(2)\}$. Their intersection being empty implies $\mathcal{B}^+_{new} = \mathcal{B}^+$ and $\mathcal{B}^0_{new} = \mathcal{B}^0$, i.e., nothing new is learned by* not *seeing* obs*.*

Proposition 2 *Under the assumptions of Proposition 1, the filter defined above is correct.*

Proof: By Definition 2 we need to show that $B \in \Phi_{\mathcal{B}^+_{new}, \mathcal{B}^0_{new}}$ if $B \in \Phi_{\mathcal{B}^+, \mathcal{B}^0}$ and $P \cup B \models o$ for $o \in O$ while $P \cup B \not\models o$ for $o \in \mathcal{O} \setminus O$. So suppose the latter are all true, then for each $o \in O$ (and each $o \in \mathcal{O} \setminus O$, resp.) there must be a computed support $S \in \texttt{Supp}(o)$ (resp., $S \in \texttt{Supp}(\neg o)$) such that $S^+ \subseteq B$ and $S^- \cap B = \emptyset$. By construction of $\mathcal{B}^+_{new}, \mathcal{B}^0_{new}$ this implies $\mathcal{B}^+_{new} \subseteq B \subseteq \mathcal{B}^+_{new} \cup \mathcal{B}^0_{new}$. Hence, $B \in \Phi_{\mathcal{B}^+_{new}, \mathcal{B}^0_{new}}$ according to (3). $\square$

Since the compact representation of incomplete knowledge via (3) does not support reasoning by disjunction, the filter is necessarily incomplete. Recall, for instance, the second case in Example 4. Not observing *obs* means that $b(1)$ or $b(2)$ must be true. Hence, model $\emptyset$ could be filtered out but is not because no two sets $\mathcal{B}^+, \mathcal{B}^0$ can encode $\Phi = \{\{b(1)\}, \{b(2)\}, \{b(1), b(2)\}\}$ via (3).

A Basic Update Method

The method for filtering with logic programs forms the core of our approach to building a basic reasoner for GDL-II. The syntactic restrictions for GDL-II ensure that the underlying open logic program satisfies the conditions Propositions 1 and 2. This will guarantee that the knowledge the player keeps in $\mathcal{B}^+, \mathcal{B}^0$ is always correct.

The procedure for maintaining the player's incomplete knowledge about a state is as follows, where G denotes the GDL-II description of a game whose semantics is given as per Definition 1; and where *my_role* $\in R$ is the role assigned to the player.

1. $\mathcal{B}^+ := \{\texttt{true(F)}\theta : \langle G, \emptyset, \emptyset \rangle \vdash_{\theta,\emptyset} \texttt{init(F)}\}$; $\mathcal{B}^0 := \emptyset$
2. In every round,
   2.1 Compute the possible legal moves of all other roles:
   $$\mathcal{L} := \{\texttt{(R,M)}\theta : \langle G, \mathcal{B}^+, \mathcal{B}^0 \rangle \vdash_{\theta,S} \texttt{legal(R,M)}, \texttt{R} \neq \textit{my_role}\}$$
   2.2 Let *my_move* be the selected move of the basic player and *my_percepts* the player's percepts.
   – Let $\mathcal{B}^+ := \mathcal{B}^+ \cup \{\texttt{does(}\textit{my_role}, \textit{my_move}\texttt{)}\}$
   $\mathcal{B}^0 := \mathcal{B}^0 \cup \{\texttt{does(R,M)} : \texttt{(R,M)} \in \mathcal{L}\}$
   – Now, let
   $$\mathcal{O} := \{\texttt{sees(}\textit{my_role}\texttt{,P)}\theta : \langle G, \mathcal{B}^+, \mathcal{B}^0 \rangle \vdash_{\theta,S} \texttt{sees(}\textit{my_role}\texttt{,P)}\}$$
   $$O := \{\texttt{sees(}\textit{my_role}, p\texttt{)} : p \in \textit{my_percepts}\}$$
   and compute $\mathcal{B}^+_{new}, \mathcal{B}^0_{new}$ as the result of filtering $\mathcal{B}^+, \mathcal{B}^0$ by O wrt. G and $\mathcal{O}$.
   – The knowledge about the next state is obtained as
   $$\mathcal{B}^+ := \{\texttt{true(F)}\theta : \langle G, \mathcal{B}^+_{new}, \mathcal{B}^0_{new} \rangle \vdash_{\theta,\emptyset} \texttt{next(F)}\}$$
   $$\mathcal{B}^0 := \{\texttt{true(F)}\theta : \langle G, \mathcal{B}^+_{new}, \mathcal{B}^0_{new} \rangle \vdash_{\theta,S} \texttt{next(F)}\} \setminus \mathcal{B}^+$$
3. The player knows that the game has terminated in case $\langle G, \mathcal{B}^+, \mathcal{B}^0 \rangle \vdash_{\varepsilon,\emptyset} \texttt{terminal}$.

Put in words, the player starts with complete information about the initial state (step 1). In every round, the player's knowledge is characterized by the skeptical consequences of the open logic program consisting of the game rules plus the incomplete knowledge $\mathcal{B}^+, \mathcal{B}^0$; specifically, this allows us to determine the player's own known legal moves as

$$\{\texttt{M}\theta : \langle G, \mathcal{B}^+, \mathcal{B}^0 \rangle \vdash_{\theta,\emptyset} \texttt{legal(}\textit{my_role}\texttt{,M)}\}^4$$

The incomplete knowledge also allows us to compute credulous consequences, in particular the possible legal moves of all other players (step 2.1). For the update of $\mathcal{B}^+, \mathcal{B}^0$ (step 2.2), we first add to $\mathcal{B}^+$ the knowledge of the player's own move and to $\mathcal{B}^0$ the possible moves by the opponents. This allows us to determine the range of possible observations, $\mathcal{O}$, in order then to filter the player's knowledge by actual observations O. Finally, the player's knowledge of the updated state is determined as the skeptically (for $\mathcal{B}^+$) and credulously (for $\mathcal{B}^0$) entailed instances of `next(F)`.

The incompleteness of the filtering implies that the reasoner for GDL-II thus defined is incomplete in general. It is, however, easy to show that it is complete in case the only `sees`-rule for a player is

```
sees(player,move(R,M)) :- does(R,M).
```

This is so because under this rule the only support for an instance `sees(player,move(r,m))` is `does(r,m)`, and the only countersupport in case the observation is not made is $\neg$`does(r,m)`. Hence, the filter will add the former to $\mathcal{B}^+$ and remove all of the latter from $\mathcal{B}^0$. The update procedure will then result in complete knowledge whenever the player starts with complete knowledge.

Experimental Results

Because the representation of incomplete knowledge and the backward chaining-based filtering are in themselves incomplete, we have run experiments to test both the effectiveness and the efficiency of our method. We have used a simple implementation in the form of a vanilla meta-interpreter in Prolog. We have run experiments with all games that were played

[4] The player knows that it doesn't know all its legal moves if some instance `legal(`*my_role*`,M)` can be derived only with non-empty support, i.e., is credulously but not skeptically entailed.

Game	Legal	Terminal	Goal	Time
Backgammon	✓	✓	✓	8.84
Banker/Thief (role 1)	✓	✓	no	0.42
Banker/Thief (role 2)	✓	✓	no	0.69
Battleships in Fog	no	–	–	–
Battleships in Fog*	✓	no	no	930.04
Blackjack	no	–	–	–
Hidden Connect	✓	✓	no	4.08
Hold your Course II	✓	✓	✓	2.05
Krieg-Tictactoe 5x5	no	–	–	–
Mastermind448	✓	✓	no	0.58
Minesweeper (role 1)	✓	✓	✓	1.56
Minesweeper (role 2)	✓	no	no	199.82
Numberguessing	✓	✓	no	1.53
Monty Hall (role 1)	✓	✓	✓	0.21
Monty Hall (role 2)	✓	✓	✓	0.21
Small Dominion 2	✓	✓	✓	12376.75
Transit (role 1)	✓	no	no	4.18
Transit (role 2)	✓	no	no	5.76
vis_Pacman (role 1,2)	✓	no	no	(706.45)
vis_Pacman (role 3)	✓	no	✓	32.12

Table 1: Experimental results testing the basic player's ability to always know its legal moves, whether a game has terminated, and what its own goal value is in the end.

at past general game playing competitions with imperfect-information track.[5] We ran 1000 simulated random matches each to test whether the legal player always knew its legal moves, and also—in case it did—whether it had sufficient knowledge to know at the end that the game must have terminated and to derive its own goal value.

The results are summarized in Table 1. For games with two or more non-`random` roles that are not symmetric, we have run the basic player for each of them as shown. The times given are the average time, in seconds (cpu time), that the player took for 1000 rounds of updating its incomplete state knowledge on a desktop computer with a 2.66 GHz CPU and 8GB memory running Eclipse Prolog. Overall, the results demonstrate both the effectiveness and the efficiency of our basic backward-chaining method.

Knowledge of Legal Moves The experiments showed that the basic player always knows its legal moves in almost all of the games. An exception is Krieg-Tictactoe, where the uncertainty about the legal moves is due to the rule below.

```
legal(P,mark(M,N)) :-
  role(P), true(cell(M,N,C)), distinct(C,P).
```

This rule says that a player `P` may attempt to mark any cell that is not already marked with his own symbol. From a game-theoretic point of view this rule is correct, because players always knows all cells occupied with their own symbol, that is, for which `true(cell(M,N,P))` holds. Hence, they also know which cells they have *not* yet marked, that is, for which `true(cell(M,N,b))` or `true(cell(M,N,Q))` holds, with `Q` denoting the opponent of `P`. So in principle they can always determine their legal moves. But the basic player does not know which of the other cells are blank and which have been marked by the opponent. Lacking the ability to reason disjunctively means that in this case there is no ground instance of `true(cell(M,N,C))` that is known to satisfy the body of the rule from above.

For a similar reason, the basic player fails to determine its legal moves in Blackjack. In the original version of Battleships in Fog, the reason why the player is uncertain about its legal moves lies in these (slightly simplified) rules:

```
sees(admiral,position(admiral,A,B)) :-
   does(random,setup(A,B,C,D)).
sees(commodore,position(commodore,C,D)) :-
   does(random,setup(A,B,C,D)).

next(position(admiral,A,B)) :-
   does(random,setup(A,B,C,D)).
next(position(commodore,C,D)) :-
   does(random,setup(A,B,C,D)).
```

Here, in a single random move two ships get positioned, one for each player. Despite the given information, however, the legal player is unable to determine the location of its own ship because the observation of some arguments of `setup(A,B,C,D)` does not entail a fully known instance of this move, and hence nothing can be learnt from filtering through an observation like, for example, `sees(admiral,position(admiral,1,2))`. For the sake of experimentation, we have defined a variant of the original game (marked by * in Table 1) where the random move is broken into two moves. With this simple modification, the basic player is able to determine its legal moves throughout that game.

Knowledge of Termination and Goal Values Somehow surprisingly, in a number of games the legal player was not able to derive that a game has terminated and what its goal value was. An inspection of the game rules showed that this is due to the fact that the game rules as such provide players with insufficient information in this regard. Hence, there is an argument to be made for requiring that games in competitions should always be defined so that the percepts suffice for every player to determine their outcome at the end.

Times The runtimes depicted in Table 1 show that basic backward chaining in general is an efficient approach for filtering observations and inferring the updated incomplete state knowledge in a basic player for GDL-II. A notable exception was the 3-person, imperfect-information version of the Pacman game when taking the role of either of the two "ghosts." In this game, the reasoner always slowed down significantly after around move 50 (where the maximal length of that game is 100 moves), and we had to interrupt the experiments a few moves later. We re-ran the experiments with a version of the

[5] The 1st German Open 2011, see `http://fai.cs.uni-saarland.de/kissmann/ggp/go-ggp`; and the 1st Australian Open 2012, see `https://wiki.cse.unsw.edu.au/ai2012/GGP`

basic player that only filters through the observations actually made instead of also filtering through all non-percepts. The results given in Table 1 for "vis_Pacman (role 1,2)" were obtained for this simplified legal player.

Conclusion

We have developed a method for filtering with logic programs and applied it to build a basic legal player for GDL-II based on backward-chaining. Our notion of filtering is similar to [Amir and Russell, 2003; Shirazi and Amir, 2011]; in their case a dynamic system is not described by logic program rules but in the Situation Calculus. For our backward-chaining filtering method we have adapted results for open logic program from [Bonatti, 2001a; 2001b]. Our experiments showed that the method is sufficiently efficient in almost all games from previous GGP competitions. It is worth stressing that even in games where the reasoner is not fast enough to be used at every node of a search tree, it can and in fact should be applied at least at the beginning in order to guarantee that the player submits a legal move. Our method also proved effective in almost all games, which supports an argument that can be made for it to be generally desirable that all GDL-II games for competitions are written so that backward chaining augmented by support computation suffices to always determine a player's legal moves, as in our reformulation of Battleships in Fog.

Acknowledgements. This research was supported under Australian Research Council's *Discovery Projects* funding scheme (project DP 120102023). The author is the recipient of an ARC Future Fellowship (project FT 0991348). He is also affiliated with the University of Western Sydney.

References

[Amir and Russell, 2003] Eyal Amir and Stuart Russell. Logical filtering. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 75–82, Acapulco, Mexico, August 2003. Morgan Kaufmann.

[Apt *et al.*, 1987] Krzysztof Apt, H. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, chapter 2, pages 89–148. Morgan Kaufmann, 1987.

[Bonatti, 2001a] Piero Bonatti. Reasoning with open logic programs. In T. Eiter, W. Faber, and M. Trusczynski, editors, *Proceedings of the International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR)*, volume 2173 of *LNCS*, pages 147–159, Vienna, Austria, September 2001. Springer.

[Bonatti, 2001b] Piero Bonatti. Resolution with skeptical stable model semantics. *Journal of Automated Reasoning*, 156(1):391–421, 2001.

[Edelkamp *et al.*, 2012] Stefan Edelkamp, Tim Federholzner, and Peter Kissmann. Searching with partial belief states in general games with incomplete information. In B. Glimm and A. Krüger, editors, *Proceedings of the German Annual Conference on Artificial Intelligence (KI)*, volume 7526 of *LNCS*, pages 25–36, Saarbrücken, Germany, September 2012. Springer.

[Gelfond and Lifschitz, 1988] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *Proceedings of the International Joint Conference and Symposium on Logic Programming (IJCSLP)*, pages 1070–1080, Seattle, OR, 1988. MIT Press.

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

[Kissmann and Edelkamp, 2010] Peter Kissmann and Stefan Edelkamp. Instantiating general games using Prolog or dependency graphs. In R. Dillmann, J. Beyerer, U. Hanebeck, and T. Schultz, editors, *Proceedings of the German Annual Conference on Artificial Intelligence (KI)*, volume 6359 of *LNCS*, pages 255–262, Karlsruhe, Germany, September 2010. Springer.

[Kleene, 1952] Stephen Kleene. *Introduction to Metamathematics*. Van Nostrand, New York, 1952.

[Lloyd and Topor, 1986] John Lloyd and R. Topor. A basis for deductive database systems II. *Journal of Logic Programming*, 3(1):55–67, 1986.

[Lloyd, 1987] John Lloyd. *Foundations of Logic Programming*. Series Symbolic Computation. Springer, second, extended edition, 1987.

[Love *et al.*, 2006] Nathaniel Love, Timothy Hinrichs, David Haley, Eric Schkufza, and Michael Genesereth. General Game Playing: Game Description Language Specification. Technical Report LG–2006–01, Stanford Logic Group, Computer Science Department, Stanford University, 353 Serra Mall, Stanford, CA 94305, 2006. Available at: `games.stanford.edu`.

[Parker *et al.*, 2005] Austin Parker, Dana Nau, and V.S̃. Subrahmanian. Game-tree search with combinatorially large belief states. In L. Kaelbling and A. Saffiotti, editors, *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 254–259, Edinburgh, UK, August 2005.

[Quenault and Cazenave, 2007] Michel Quenault and Tristan Cazenave. Extended general gaming model. In *Computers Games Workshop*, pages 195–2004, Amsterdam, June 2007.

[Richards and Amir, 2009] Mark Richards and Eyal Amir. Information set sampling in general imperfect information positional games. In *Proceedings of the IJCAI Workshop on General Intelligence in Game-Playing Agents (GIGA)*, pages 59–66, Pasadena, 2009.

[Rosenhouse, 2009] Jason Rosenhouse. *The Monty Hall Problem*. Oxford University Press, 2009.

[Saffidine and Cazenave, 2011] Abdallah Saffidine and Tristan Cazenave. A forward chaining based game description language compiler. In *Proceedings of the IJCAI Workshop*

on General Intelligence in Game-Playing Agents (GIGA), pages 69–75, Barcelona, 2011.

[Schkufza *et al.*, 2008] Eric Schkufza, Nathaniel Love, and Michael Genesereth. Propositional automata and cell automata: Representational frameworks for discrete dynamic systems. In *Proceedings of the Australasian Joint Conference on Artificial Intelligence*, volume 5360 of *LNCS*, pages 56–66, Auckland, December 2008. Springer.

[Schofield *et al.*, 2012] Michael Schofield, Timothy Cerexhe, and Michael Thielscher. HyperPlay: A solution to general game playing with imperfect information. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 1606–1612, Toronto, July 2012. AAAI Press.

[Shepherdson, 1984] John C. Shepherdson. Negation as failure: A comparison of Clark's completed data base and Reiter's closed world assumption. *Journal of Logic Programming*, 1:51–79, 1984.

[Shirazi and Amir, 2011] Afsaneh Shirazi and Eyal Amir. First-order logical filtering. *Artificial Intelligence*, 175(1):193–219, 2011.

[Silver and Veness, 2010] David Silver and Joel Veness. Monte-Carlo planning in large POMDPs. In J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing (NIPS)*, pages 2164–2172, Vancouver, Canada, December 2010.

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

[Thielscher, 2011] Michael Thielscher. The general game playing description language is universal. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1107–1112, Barcelona, July 2011.

[Waugh, 2009] Kevin Waugh. Faster state manipulation in general games using generated code. In *Proceedings of the IJCAI Workshop on General Intelligence in Game-Playing Agents (GIGA)*, pages 91–97, Pasadena, 2009.

Sufficiency-Based Selection Strategy for MCTS *

Stefan Freyr Gudmundsson and Yngvi Björnsson
School of Computer Science
Reykjavik University, Iceland
{stefang10,yngvi}@ru.is

Abstract

Monte-Carlo Tree Search (MCTS) has proved a remarkably effective decision mechanism in many different game domains, including computer Go and general game playing (GGP). However, in GGP, where many disparate games are played, certain type of games have proved to be particularly problematic for MCTS. One of the problems are game trees with so-called optimistic moves, that is, bad moves that superficially look good but potentially require much simulation effort to prove otherwise. Such scenarios can be difficult to identify in real time and can lead to suboptimal or even harmful decisions. In this paper we investigate a selection strategy for MCTS to alleviate this problem. The strategy, called *sufficiency threshold*, concentrates simulation effort better for resolving potential optimistic move scenarios. The improved strategy is evaluated empirically in an n-arm-bandit test domain for highlighting its properties as well as in a state-of-the-art GGP agent to demonstrate its effectiveness in practice. The new strategy shows significant improvements in both domains.

1 Introduction

From the inception of the field of Artificial Intelligence (AI), over half a century ago, games have played an important role as a testbed for advancements in the field, resulting in game-playing systems that have reached or surpassed humans in many games. A notable milestone was reached when IBM's chess program Deep Blue [Campbell *et al.*, 2002] won a match against the number one chess player in the world, Garry Kasparov, in 1997. The 'brain' of Deep Blue relied heavily on both an efficient minimax-based game-tree search algorithm for thinking ahead and sophisticated knowledge-based evaluation of game positions, using human chess knowledge accumulated over centuries of play. A similar approach has been used to build world-class programs for many other deterministic games, including Checkers [Schaeffer, 2009] and Othello [Buro, 1999].

For non-deterministic games, in which moves may be subject to chance, Monte-Carlo sampling methods have additionally been used to further improve decision quality. To accurately evaluate a position and the move options available, one plays out (or samples) a large number of games as a part of the evaluation process. Backgammon is one example of a non-deterministic game, where possible moves are determined by rolls of dice, for which such an approach led to world-class computer programs (e.g., TD-Gammon [Tesauro, 1994]).

In recent years, a new simulation-based paradigm for game-tree search has emerged, Monte-Carlo Tree Search (MCTS) [Coulom, 2006; Kocsis and Szepesvári, 2006]. MCTS combines elements from both traditional game-tree search and Monte-Carlo simulations to form a full-fledged best-first search procedure. Many games, both non-deterministic and deterministic, lend themselves well to the MCTS approach. As an example, MCTS has in the past few years greatly enhanced the state of the art of computer Go [Enzenberger and Müller, 2009], a game that has eluded computer based approaches so far.

MCTS has also been used successfully in General Game Playing (GGP) [Genesereth *et al.*, 2005]. The goal there is to create intelligent agents that can automatically learn how to skillfully play a wide variety of games, given only the descriptions of the game rules (in a language called GDL [Love *et al.*, 2008]). This requires that the agents learn diverse game-playing strategies without any game-specific knowledge being provided by their developers. Most of the strongest GGP agents are now MCTS-based, such as ARY [Méhat and Cazenave, 2011], CADIAPLAYER [Björnsson and Finnsson, 2009; Finnsson and Björnsson, 2011a], MALIGNE [Kirci *et al.*, 2011], and TURBOTURTLE. Although MCTS have proved on average more effective than traditional heuristic-based game-tree search in GGP, there is still a large number of game domains where it does not work nearly as well, for example in non-progressing or highly tactical (chess-like) games. The more general concept of *optimistic actions*, encapsulating among other things tactical traps, is by and large problematic for MCTS [Ramanujan *et al.*, 2010; Finnsson and Björnsson, 2011b].

In this paper we propose an improved selection strategy for MCTS, *sufficiency-threshold*, that is more effective in domains troubled with optimistic actions and, generally speaking, more robust on the whole. We also take steps towards

*This paper was also submitted (and accepted) to the main technical track of IJCAI'13.

better understanding how determinism and discrete game outcomes affect the action-selection mechanism of MCTS, and then empirically evaluate the sufficiency-threshold strategy in such domains, where it shows significant improvements.

The paper is structured as follows. In the next section we provide necessary background material, then we discuss sufficiently good moves and the sufficiency-threshold strategy. This is followed by an empirical evaluation of the strategy in both an n-arm-bandit setup and GGP. Finally, we conclude and discuss future work.

2 Background

Before we introduce our new selection strategy we first provide the necessary background on MCTS, optimistic actions, and n-arm-bandits (which we use as a part of the experimental evaluation of the new selection strategy).

2.1 Monte-Carlo Tree Search

Monte-Carlo Tree Search (MCTS) is a simulation-based search technique that extends Monte-Carlo simulations to be better suited for (adversary) games. It starts by running a pure Monte-Carlo simulation, but gradually builds a game tree in memory with each new simulation. This allows for a more informed mechanism where each simulation consists of four strategic steps: *selection*, *expansion*, *playout*, and *back-propagation*. In the *selection* step, the tree is traversed from the root of the game tree until a leave node is reached, where the *expansion* step expands the leave by one level (typically adding only a single node). From the newly added node a regular Monte-Carlo *playout* is run until the end of the game (or when some other terminating condition is met), at which point the result is *back-propagated* back up to the root modifying the statistics stored in the game tree as appropriate. MCTS continues to run such four step simulations until deliberation time is up, at which point the most promising action of the root node is played.

In this paper we are mainly concerned with the selection step, where *Upper Confidence-bounds applied to Trees (UCT)* [Kocsis and Szepesvári, 2006] is widely used for action selection. At each internal node in the game tree an action a^* to simulate is chosen as follows:

$$a^* = argmax_{a \in A(s)} \left\{ Q(s,a) + C\sqrt{\frac{\ln N(s)}{N(s,a)}} \right\} \quad (1)$$

$N(s)$ stands for the number of samples gathered in state s and $N(s,a)$ for number of samples gathered when taking action a in state s. $A(s)$ is the set of possible actions in state s and $Q(s,a)$ is the expected return for action a in state s, usually the arithmetic mean of the $N(s,a)$ samples gathered for action a. The term added to $Q(s,a)$ decides how much we are willing to explore, where the constant C dictates how much effect the exploration term has versus exploitation. With $C = 0$ our samples would be gathered greedily, always selecting the top-rated action for each playout. When we have values of $N(s,a)$ which are not defined, we consider the exploration term as being infinity.

Although MCTS is effective in many game domains, it has difficulties in other common game structures. Several such properties have been identified, including traps, the non-progression property and optimistic actions [Ramanujan *et al.*, 2010; Finnsson and Björnsson, 2011b].

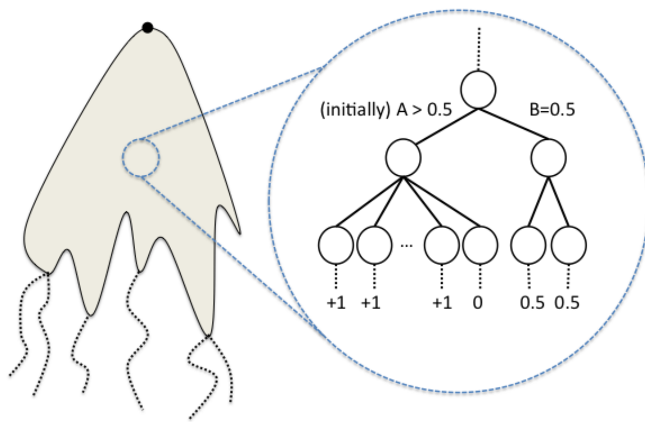


Figure 1: An example of an optimistic action in a MCTS tree: move A initially, but incorrectly, looks much better than move B, because the one move that refutes A (scored as 0) has many siblings that all are winning (for example, in chess, the refutation move could be a trivial recapture of a piece, however, by not recapturing the game is lost).

2.2 Optimistic Actions

Optimistic actions are moves that upon initial investigation look promising, even leading to a win, but are easily refuted in practice. A common source of this problem in simulation-based search are moves leading to positions where the opponent has many legal replies but with only one (or a very few) of them being a refutation. It takes many simulations to explore all the opponent's options and establish the true refutation. Thus, most of the simulations return a positive feedback to start with, labeling the move path leading to that position with a too optimistic score. This can happen at any level in the MCTS game tree, as depicted in Figure 1. Such scenarios are common in many games, for example, recapturing a piece in chess (or other material-dominant games).

When such positions occur in the MCTS game-tree they continue to back-propagate false (too optimistic) values until enough simulations have been performed. In such scenarios it may be better to concentrate simulations on the suspected optimistic move to correct its value. A related scenario is when there are several good looking moves in a given position with a similar value. The standard UCT selection strategy would distribute the simulation effort among them somewhat equally to try to establish a single best move. This may be problematic in the presence of optimistic moves. A better strategy may be to instead commit to one of those sufficiently good moves, at least in discrete outcome deterministic games as we show. This has the benefit of increasing the certainty of the returned value and potentially avoid the optimistic move fallacy. Once the refutation reply has been identified subsequent simulations start to return a radically different value, resulting in the mean score values decreasing.

2.3 N-arm-bandit and the Mean's Path

To simulate a decision process for choosing moves in a game we can think of a one-arm-bandit problem. We stand in front of a number of one-arm-bandits, slot machines, with coins to play with. Each bandit has its own probability distribution which is unknown to us. The question is, how do we maximize our profit? We start by playing the bandits to gather some information. Then, we have to decide where to put our limited amount of coins. This becomes a matter of balancing between exploiting and exploring, i.e. play greedily and try less promising bandits. The selection formula (Equation 1) is derived from this setup [Kocsis and Szepesvári, 2006]. Instead of n one-arm-bandits we can equally talk about one n-arm-bandit and we will use that terminology henceforth.

What do we mean by the bandit's probability distribution? If we only consider the slot machine and discard the game-tree connection we are likely to identify the distribution with its mean, which we believe to be a constant value. With increasing number of samples we gather from the bandit the more the sampled mean approaches the bandit's mean. This happens with more and more certainty thanks to the central limit theorem. Let us re-connect with the game tree. How well does this approach describe what happens in a game tree? In a previous section we defined the optimistic move, i.e. a move which looks promising to begin with when simulations are scarce. Let us assume we are deciding which move to play in a given position and one move shows a very high score after 100 simulations but the scores drops significantly after 1000 simulations, e.g. when we have discovered its refutation further down the game tree. If we play the same position repeatedly, starting from scratch, and measure the score for this move after 100 simulations each time, it would always have a high score. The average of the move for the repeated position would approach the move's correct mean for 100 simulations. However, with additional simulations we would approach a different mean. This is because in a game tree the mean can be moving as new promising moves get established. Therefore, when using an n-arm-bandit to model the behavior of a simulation-based search in a game tree, it is more accurate to accompany each bandit with a path that its mean will follow as opposed to a constant mean. This path we call the mean's path and picture it as a function somewhat related to a discretized random walk.

When dealing with game trees and a selection mechanism such as MCTS the mean can truly change as the Monte-Carlo tree grows larger. For example, in adversary games the MCTS gradually builds up a minimax line of play and discovering a strong reply can drastically change a high sample mean. An important part of Kocsis and Szepesvari's work [Kocsis and Szepesvári, 2006] is that they prove that the selection formula (1) will in the end find the true, game theoretic, value of a position. For the mean's path this equals a stability will be reached after enough number of steps — or simulations. What are then the possible stable values? In deterministic games the true value of a position can only be one of the terminal values, e.g. in a game with binary results, win or loss, the mean's path will only stabilize at the win or loss value. Deterministic games with a few (e.g. two or three) possible terminal values will therefore have the same few possible stable values. This can be exploited as we will show. Non-deterministic games have a different nature as the chance nodes can lead to true values unlike the terminal values.

In [Kocsis and Szepesvári, 2006] the goal is to minimize regret, i.e. we want to minimize our loss of playing the bandits. Using a simple regret would better describe the process of choosing a move in a game [Tolpin and Shimony, 2012]. We can look at it as an n-arm-bandit where we have a fixed amount of coins to use to gather information after which we have to choose one arm to gamble all our money on and the outcome is dictated by the bandit's stable or true value. We only consider simple regret here.

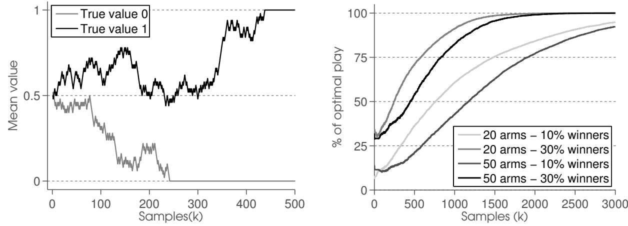
We will not spend many words on the variance of the probability distribution of each arm. The volatility of a position could be evaluated in some games which could be reflected in the value of the standard deviation.

When discussing the action selection for n-arm-bandits we usually talk about UCB (*Upper Confidence Bound*) [Auer *et al.*, 2002] and UCT when working with trees. Avoiding ambiguity we will talk about UCT for both scenarios throughout this paper.

3 Sufficiently Good Moves

Assume that after running a fixed number of simulations in a game, two of the available actions in a position have established themselves as substantially better than the others, say scoring 0.85 and 0.88 respectively where the scoring is between 0 (loss) and 1 win. In a non-deterministic game with a substantial chance element, or in a game where the outcome is scored on a fine grained scale, one might consider spending additional simulations to truly establish which one of the two actions is indeed the better one before committing to either one to play. In contrast, in a deterministic game with a few outcomes this is not necessarily the case. Both moves are likely to lead to a win and no matter which one is played the true outcome is preserved. So, instead of spending additional simulations on deciding between two inconsequential decisions, the resources could be used more effectively. Generally speaking, if there are only win or loss outcomes possible in a deterministic game then once the $Q(s, a)$ values become sufficiently close to a legitimate outcome based on enough simulations, spending additional simulations to distinguish between close values is not necessarily wise use of computing resources. This is even more so true in games suffering from suspected optimistic moves. As mentioned earlier a deterministic game with only win and loss outcomes has only two stable values for the mean's path. We want to take advantage of situations where the possible stable values are easily distinguished and the sample means are close to one of the values. On the other hand when the stable values are unpredictable or close to each others it is possibly better to use other methods [Tolpin and Shimony, 2012; Auer and Ortner, 2000] to gain more accurate estimates of the perceived best moves. We expect this to happen more often in non-deterministic games and deterministic games with many possible outcomes.

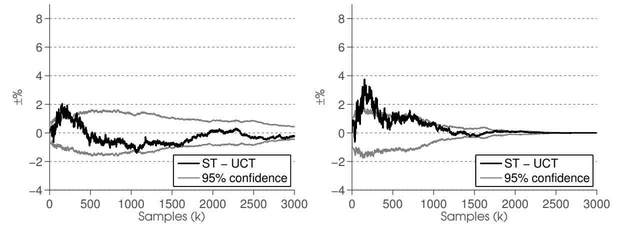
To better understand this concept think of a position in chess where a player can capture a rook or a knight. After a



(a) Two types of mean's path following a random walk

(b) UCT with various number of arms and winners

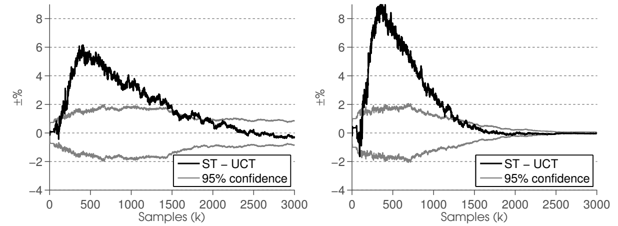
Figure 2: Two examples of mean's paths and ratio of optimal play using UCT



(a) ST vs UCT (10% win)

(b) ST vs UCT (30% win)

Figure 3: 20 arms



(a) ST vs UCT (10% win)

(b) ST vs UCT (30% win)

Figure 4: 50 arms

few simulations we get high estimates for both moves. Probabilities are that both lead to a win, i.e. both might have the same true value as 1. For humans it is possibly easier to secure the victory by capturing the rook but we are more interested in knowing whether there is a dangerous reply lurking just beyond our horizon, i.e. whether one of the moves is an optimistic move. We argue that at this point it is more important to get more reliable information about one of the moves instead of trying to distinguish between, possibly, close to equally good moves. Either our estimate of one of the moves stays high or even gets higher and our confidence increases or the estimate drops and we have proven the original estimate wrong which can be equally important. We introduce a sufficiency threshold α such that whenever we have an estimate $Q(s,a) > \alpha$ from (1) we say that this move is sufficiently good and therefore unplug the exploration. To do so we replace C in Equation (1) by $\hat{C}$ as follows:

$$\hat{C} = \begin{cases} C & \text{when all } Q(s,a) \leq \alpha, \\ 0 & \text{when any } Q(s,a) > \alpha. \end{cases} \quad (2)$$

We call this method *sufficiency threshold (ST)*. When our estimates drop below the sufficiency threshold we go back to the original UCT method. For unclear or bad positions where estimates are less than α most of the time, showing occasional spikes, this approach differs from UCT in temporarily rearranging the order of moves to sample. After such a rearrangement the methods more or less couple back to selecting the same moves to sample from.

4 Experiments

We empirically evaluated the ST selection strategy in three different scenarios: an n-arm-bandit to clearly demonstrate its properties, a sample game position demonstrating its potentials in alleviating problems caused by optimistic moves, and finally in a simulation-based GGP agent to show its effectiveness on a variety of games.

4.1 N-arm-bandit Experiments

In our n-arm-bandits experiment we consider only true values 0 and 1. With each sample we gather for a bandit we move one step further along the mean's path.

Our setup is related to Sutton and Barto's approach (1998) but adapted for deterministic environments. Once a path reaches 0 or 1 it has found its true value and does not change after that. This way we get closer to the true value of a bandit the more samples we gather from it. Figure 2a shows possible paths hitting 0 or 1. We let $M_i(k_i)$ be the mean value for bandit i after k_i samples. The total number of samples is $k = \sum_i k_i$. We use the results from the samples to evaluate the expected rewards of the bandits. Let $Q_i(k)$ play the same role as $Q(s,a)$ in (1), i.e. be the expected reward for bandit i after k samples. For each k we record which arm we would choose for our final bet, i.e. with the highest $Q_i(k)$ value.

We experiment with a bandit as follows. Pulling an arm once is a *sample*. A single *task* is to gather information for k samples, $k \in [1, 3000]$. For each sample we measure which action an agent would take at that point, i.e. which bandit would we gamble all our money on with current information available to us. Let $V(k)$ be the value of the action taken after gathering k samples. $V(k) = 1$ if the chosen bandit has a true value of 1 and $V(k) = 0$ otherwise. A *trial* consists of running t tasks and calculate the mean value of $V(k)$ for each $k \in [1, 3000]$. This gives us one measurement, $\bar{V}(k)$, which measures the ratio of optimal play for an agent with respect to k. There is always at least one bandit with a true value of 1. Each trial is for a single n-arm-bandit, representing one type of a position in a game. In the experiments that follow we compare the performance of ST to UCT, using parameter settings of $C = 0.4$ and $\alpha = 0.6$.

We run experiments on 50 different bandits (models) generated randomly as follows. All the arms start with $M_i(1) = 0.5$ and have randomly generated mean's paths although constrained such that they hit loss (0) or win (1) before taking 500 steps. The step size is 0.02 and each step is equally likely to be positive or negative. One trial consisting of 200 tasks is

run for each bandit, giving us 50 measurements of $\bar{V}(k)$ for each agent and each $k \in [1, 3000]$. In the following charts we show a 95% confidence interval over the models.

In the experiments two dimensions of the models are varied: first the number of arms are either 20 or 50, and second, either 10% or 30% of the arms lead to a win (the remaining to a loss). Figure 2b shows $\bar{V}(k)$ for UCT, which we use as a benchmark. Figures 3 and 4 show the performance of ST relative to UCT when using 20 and 50 arms, respectively. The figures show the increase or decrease in the ratio of optimal play for each k.

ST is overall doing significantly better than UCT except when we have 20 arms and 10% winners. With 50 arms the ST agent is much better than UCT. The general trend is that to begin with there is simply not enough information for neither ST nor UCT to figure out which moves are promising and which are not. After a while ST starts to perform better and only after many more simulations is UCT able to catch up.

4.2 Game Experiments

Using simplified models as we did in the aforementioned experiments is useful for showing the fundamental differences of the individual action-selection schemes. However, an important question to answer is whether the models fit real games. First, we try to get a clearer picture of the optimistic move and how the ST is able to guide the selection strategy out of its optimistic move traps. We have setup a position in the game Breakthrough, frequently played in different variants in GGP competitions. It is highly tactical deterministic game with only win and loss outcomes. It has proved challenging for MCTS to play accurately [Finnsson and Björnsson, 2008]. Each player starts with its first two backranks occupied by pawns of its own color and the goal is to advance a pawn to the opposite end of the board. The first player to achieve that wins. The pawns move forward, one square at a time, both straight and diagonally and can capture opponent's pawns with the diagonal moves.

The position in Figure 5, from a smaller board game variant (the regular game is played on an 8×8 board), showcases the problem at hand, and in a way resembles the types of arms described above. There are two promising moves which turn out to be bad, one that wins, and 10 other moves which do little. In the position, capturing a pawn on a7 or c7 with the pawn on b6 looks promising since all responses from black but one lose. Initially our samples give very high estimates of these two moves until black picks up on capturing back on a7 or c7. There is a forced win for white by playing a6. Black can not prevent white from moving to b7 in the next move, either with the pawn on a6 og b7. From b7 white can move to a8 or c8 and win.

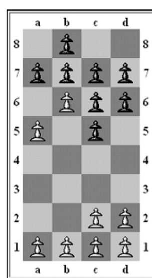


Figure 5: White wins with a5a6

Figure 6 shows how UCT and ST perform in the position in Figure 5. ST clearly outperforms UCT. This position demonstrates clearly the drawbacks of UCT. We are dealing with a problem with an optimistic move where UCT samples more

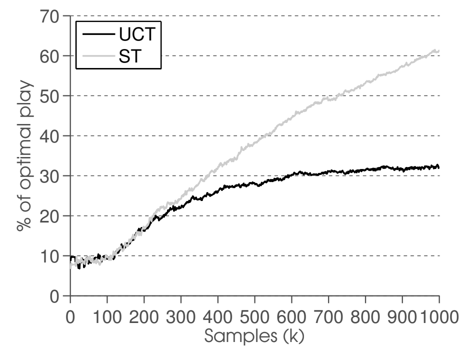


Figure 6: UCT and ST in the position in Figure 5

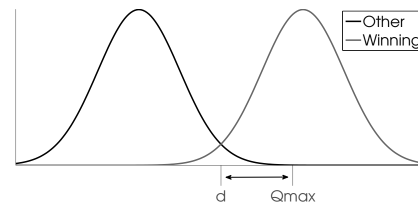


Figure 7: Statistics for the difference between d and $Qmax$

or less equally often for each of the three promising moves. In Figure 6 we see how UCT reaches a plateau around 33%, where the optimal move is played approximately $1/3$ of the time as UCT needs more samples to distinguish between the three promising moves. Being able to disprove optimistic moves early is of a particular interest in GGP where reasoning is somewhat slow, often resulting in decisions being made based on relatively few simulations.

4.3 ST in GGP

We also try the ST selection strategy in a GGP environment using a world class GGP agent, CADIAPLAYER. We test it on four games, *Chomp*, *Runners*, *Connect-4*, and *Breakthrough* [1] with different numbers of simulations for the decision making. For ST to work well in a GGP environment it needs a few adjustments. The GGP agents need to be robust across many different games. Therefore, we need to soften the sufficiecy threshold a bit. First of all it can be difficult to decide an α threshold that works for all games. Also, the simulation results need not be at the correct scale and can be misleading in its absolute values. The strength of MCTS comes from ordering the possible actions reasonably, not necessarily with very accurate values - at least not until near the end of the game. What we want to do is to discover when our best perceived move, the highest $Q(s, a) = Qmax$, is 'close enough' to the winning value. The winning value is not necessarily 1; CADIAPLAYER, for example, discounts the result with the length of the simulation. We treat this as a classification problem, where simulations ending in a victory are labelled as the winning class opposed the other class for non-winning simulations. Both classes form a distribution which then have a

[1] All found in the games repository on the Dresden GGP server

Table 1: ST enhanced CADIAPLAYER vs. CADIAPLAYER

Simulations n	500	1 000	2 000	3 000	5 000	10 000
Runners	48.9 ± 4.8	53.6 ± 4.6	52.4 ± 4.3	$\mathbf{55.8 \pm 4.0}$	52.4 ± 3.7	50.3 ± 2.7
Chomp	49.0 ± 4.9	51.0 ± 4.9	49.5 ± 4.9	51.3 ± 4.9	49.5 ± 4.9	50.3 ± 4.9
Connect 4	47.0 ± 4.8	49.3 ± 4.8	48.4 ± 4.8	47.6 ± 4.8	49.1 ± 4.8	51.3 ± 4.6
Breakthrough	49.5 ± 4.9	$\mathbf{57.3 \pm 4.9}$	$\mathbf{55.0 \pm 4.9}$	$\mathbf{55.0 \pm 4.9}$	52.8 ± 4.9	–

discriminant value, d, where it is equally likely for an unlabeled simulation result to belong to each class. The literature is rich in techniques of this sort (e.g., [Bishop, 2006]). Our approach, for simplicity, assumes the standard deviation of both distributions to be equal. Thus the discriminant value is only dependent on the average values of each class and the number of data points in each class

$$d = \frac{n_{other} \cdot \mu_{win} + n_{win} \cdot \mu_{other}}{n_{other} + n_{win}},$$

where n_{win} is the number of data in the winning class, μ_{win} is their average value and similarly n_{other} and μ_{other} are their counterparts for the non-winning class. The discriminant value, d, does not factor in where $Qmax$ is positioned relative to it. This can vary between games. Therefore, we measure the difference between d and $Qmax$ for each simulation, as depicted in Figure 7. The accumulation of these differences forms a distribution which we assume to be a normal distribution. We then use the statistical Q-function to measure the probability for a random variable from this distribution to have a value larger than the current difference $a = Qmax - d$. This probability is used directly as the probability of unplugging the exploration, although we set a floor of 10%, i.e. it is always at least a 10% chance of choosing an action to simulate with the traditional way. Perhaps, we do not need this floor but for the sake of robustness we chose such an ϵ-greedy approach. We consider all previous simulations as training data.

Table 1 shows the result between ST enhanced CADIAPLAYER versus standard CADIAPLAYER. We ran a match of 400 games between the agents, 200 as each side. We also ran it for different values n which is the fixed number of simulations the agents were given to decide each move. The n values range from 500 to 10 000. The time and space it needs are negligible in our GGP environment. The results suggest that we have windows of simulation counts where ST improves the player and outside these windows it does not seem harmful, as summarized for Breakthrough in Figure 8.

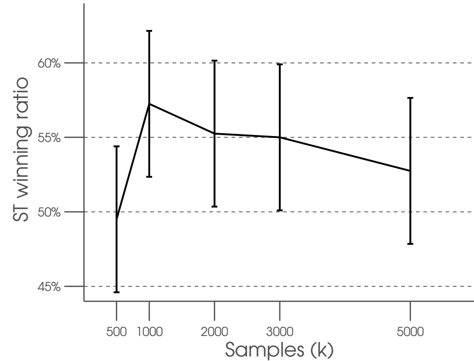


Figure 8: ST winning ratio for Breakthrough

5 Related Work

The current research focus on improving the selection phase in MCTS has been on incorporating domain knowledge to identify good actions earlier, materializing in enhanced schemes such as RAVE [Gelly and Silver, 2007] and Progressive Bias [Chaslot, 2010]. Accelerated UCT and discounted UCT [Hashimoto *et al.*, 2011] are two methods which try to solve the problem of moving mean's path. Although, they use a different terminology. Somewhat surprisingly discounted UCT has not produced positive results. We have also experimented with recency weighted (discounted) average, but with mixed success: whereas it was sometimes helpful it seemed equally often decremental (unlike our ST approach). In [Auer and Ortner, 2000] and [Tolpin and Shimony, 2012] statistical methods are used to guide the selection in favorable directions. Both assume a stable underlying mean's path and give significant improvements.

6 Conclusions and Future Work

We have shown that for certain types of games, where the stable values of the mean's path are predictable and far apart, we can improve the MCTS selection strategy with ST. It seems quite robust across many games, and was never harmful while proving particularly effective in domains suffering from the optimistic move syndrome, where it helps to expedite finding refutations. Furthermore, it seems more effective in games with a large branching factor, however, it also showed in practice promise in a low-branching factor game like Runners. This artifact could be explained by ST being able to search selected positions deeper because committing to a single move, thus finding wins and losses earlier. Furthermore, the ST method comes at little or no cost. It is easy to implement and the time and space it consumes are negligible (not measurable in our experiments).

It would be interesting to see whether we get multiple disjoint windows of this sort as the number of simulations increase. That falls under future work as well as running experiments with more games. It is also interesting to see how ST performs in agents designed for a specific games, such as Go. There, we should be able to figure out the sufficiency threshold offline so ST might come at very little computational cost. The dynamic version of ST needed for GGP could be improved with better classification tools, of which the machine-learning literature has plenty.

References

[Auer and Ortner, 2000] Peter Auer and Ronald Ortner. Ucb revisited: Improved regret bounds for the stochastic multi-armed bandit problem, 2000.

[Auer *et al.*, 2002] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2/3):235–256, 2002.

[Bishop, 2006] C. M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[Björnsson and Finnsson, 2009] Yngvi Björnsson and Hilmar Finnsson. Cadiaplayer: A simulation-based general game player. *IEEE Trans. Comput. Intellig. and AI in Games*, 1(1):4–15, 2009.

[Buro, 1999] Michael Buro. How machines have learned to play Othello. *IEEE Intelligent Systems*, 14(6):12–14, November/December 1999. Research Note.

[Campbell *et al.*, 2002] Murray Campbell, A. Joseph Hoane, Jr., and Feng-Hsiung Hsu. Deep blue. *Artificial Intelligence*, 134(1–2):57–83, 2002.

[Chaslot, 2010] Guillaume Chaslot. *Monte-Carlo Tree Search*. PhD dissertation, Maastricht University, The Netherlands, Department of Knowledge Engineering, 2010.

[Coulom, 2006] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In H. Jaap van den Herik, Paolo Ciancarini, and H. H. L. M. Donkers, editors, *Computers and Games*, volume 4630 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 2006.

[Enzenberger and Müller, 2009] Markus Enzenberger and Martin Müller. Fuego - an open-source framework for board games and go engine based on monte-carlo tree search. Technical Report 09-08, Dept. of Computing Science, University of Alberta, 2009.

[Finnsson and Björnsson, 2008] Hilmar Finnsson and Yngvi Björnsson. Simulation-based approach to general game playing. In Dieter Fox and Carla P. Gomes, editors, *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 259–264, Cambridge, MA, USA, 2008. AAAI Press.

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

[Finnsson and Björnsson, 2011b] Hilmar Finnsson and Yngvi Björnsson. Game-tree properties and mcts performance. In *IJCAI'11 Workshop on General Intelligence in Game Playing Agents (GIGA'11)*, pages 23–30, 2011.

[Gelly and Silver, 2007] Sylvain Gelly and David Silver. Combining online and offline knowledge in uct. In *Proceedings of the 24th international conference on Machine learning*, ICML '07, pages 273–280, New York, NY, USA, 2007. ACM.

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

[Hashimoto *et al.*, 2011] Junichi Hashimoto, Akihiro Kishimoto, Kazuki Yoshizoe, and Kokolo Ikeda. Accelerated UCT and its application to two-player games. In H. Jaap van den Herik and Aske Plaat, editors, *ACG*, volume 7168 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2011.

[Kirci *et al.*, 2011] Mesut Kirci, Nathan R. Sturtevant, and Jonathan Schaeffer. A GGP feature learning algorithm. *KI*, 25(1):35–42, 2011.

[Kocsis and Szepesvári, 2006] Levante Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In *European Conference on Machine Learning (ECML)*, pages 282–293, Berlin / Heidelberg, 2006. Springer.

[Love *et al.*, 2008] Nathaniel Love, Timothy Hinrichs, David Haley, Eric Schkufza, and Michael Genesereth. General game playing: Game description language specification. Technical report, Stanford University, 2008. most recent version should be available at http://games.stanford.edu/.

[Méhat and Cazenave, 2011] Jean Méhat and Tristan Cazenave. A parallel general game player. *KI*, 25(1):43–47, 2011.

[Ramanujan *et al.*, 2010] Raghuram Ramanujan, Ashish Sabharwal, and Bart Selman. On adversarial search spaces and sampling-based planning. In *ICAPS*, pages 242–245, 2010.

[Schaeffer, 2009] Jonathan Schaeffer. *One Jump Ahead:Computer Perfection at Checkers*. Springer, 2009.

[Sutton and Barto, 1998] Richard Sutton and A. Barto. *Reinforcement Learning: An Introduction*. MIT press, Cambridge, MA, USA, 1998.

[Tesauro, 1994] Gerald Tesauro. TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Comput.*, 6(2):215–219, 1994.

[Tolpin and Shimony, 2012] David Tolpin and Solomon Eyal Shimony. Mcts based on simple regret. *CoRR*, abs/1207.5536, 2012.

Decaying Simulation Strategies

Mandy J.W. Tak[1] and **Mark H.M. Winands**[1] and **Yngvi Björnsson**[2]
Games and AI Group, Department of Knowledge Engineering, Maastricht University[1]
Email: {mandy.tak,m.winands}@maastrichtuniversity.nl
School of Computer Science, Reykjavík University[2]
Email: yngvi@ru.is

Abstract

The aim of General Game Playing (GGP) is to create programs capable of playing a wide range of different games at an expert level, given only the rules of the game. The most successful GGP programs currently employ simulation-based Monte Carlo Tree Search (MCTS). The performance of MCTS depends heavily on the simulation strategy used. In this paper we investigate the application of a decay factor for two domain independent simulation stategies: N-Gram Selection Technique (NST) and Move-Average Sampling Technique (MAST). The adjustment is tested in CADIAPLAYER on 20 different games. Three types of games are used, namely: turn-taking, simultaneous-move and multi-player. Experiments reveal that a decay factor significantly improves the NST and MAST simulation strategy.

1 Introduction

Past research in Artificial Intelligence (AI) focused on developing programs that can play one game at a high level. These programs usually rely on human expert knowledge that is brought into the program by the programmers. In General Game Playing (GGP) the aim is to create programs that can play a wide range of games at a high level. The main challenge of GGP is the lack of human expert knowledge. Therefore, all knowledge need to be generated online by the program. Furthermore, it is no longer possible to determine beforehand which search technique is best suited for the game at hand. These challenges entail that a GGP program can only become successful when it incorporates a wide range of different AI techniques, like knowledge representation, knowledge discovery, machine learning, search and online optimization.

The first successful GGP programs, such as CLUNEPLAYER [Clune, 2007] and FLUXPLAYER [Schiffel and Thielscher, 2007a; 2007b], were based on minimax with an automatically learned evaluation function. CLUNEPLAYER and FLUXPLAYER won the International GGP competition in 2005 and 2006, respectively. However, ever since, GGP programs incorporating MCTS based approaches have proved more successful in the competition. In 2007, 2008 and 2012 CADIAPLAYER [Björnsson and Finnsson, 2009; Finnsson, 2012] won; in 2009 and 2010 ARY [Méhat and Cazenave, 2010]; and in 2011 TURBO TURTLE developed by Sam Schreiber. All three programs are based on MCTS, an approach particularly well suited for GGP because no game specific knowledge is required besides the basic rules of the game.

The performance of MCTS depends heavily on the simulation strategy employed in the play-out phase [Gelly and Silver, 2007]. As there is no game dependent knowledge available in GGP, generic simulation strategies need to be developed. Tak *et al.* [2012] proposed a simulation strategy based on N-Grams, called the N-Gram Selection Technique (NST). The new NST strategy was shown to on average outperforms the more established Move-Average Sampling Technique (MAST) [Finnsson and Björnsson, 2008], which was employed by CADIAPLAYER when winning the 2008 International GGP competition.

The information gatherered by NST and MAST are kept between successive searches. On the one hand this reuse of information may bolster the simulation strategy as it is immediately known what the strong moves are in the play-out. On the other hand this information can become outdated as moves that are strong in one phase of the game are weak in another phase. In this paper we investigate the application of a decay factor for NST and MAST statistics. The idea of decaying statistics was already applied in Discounted UCT [Kozelek, 2009] and [Hashimoto *et al.*, 2012]. As these UCT statistics are tied to a particular game position, the information does not get outdated in turn-taking deterministic perfect-information games. Decaying is of limited use here. However, NST and MAST are applied without taking the game position into account. As the game situation changes over time, so does the quality of the NST and MAST statistics.

The paper is structured as follows. First, Section 2 gives the necessary background information about MCTS. Next, the simulation strategies NST and MAST are explained in Section 3. The application of a decay factor is discussed in Section 4. Subsequently, Sections 5 and 6 deal with the experimental setup and results. Finally, Section 7 gives conclusions and an outlook to future research.

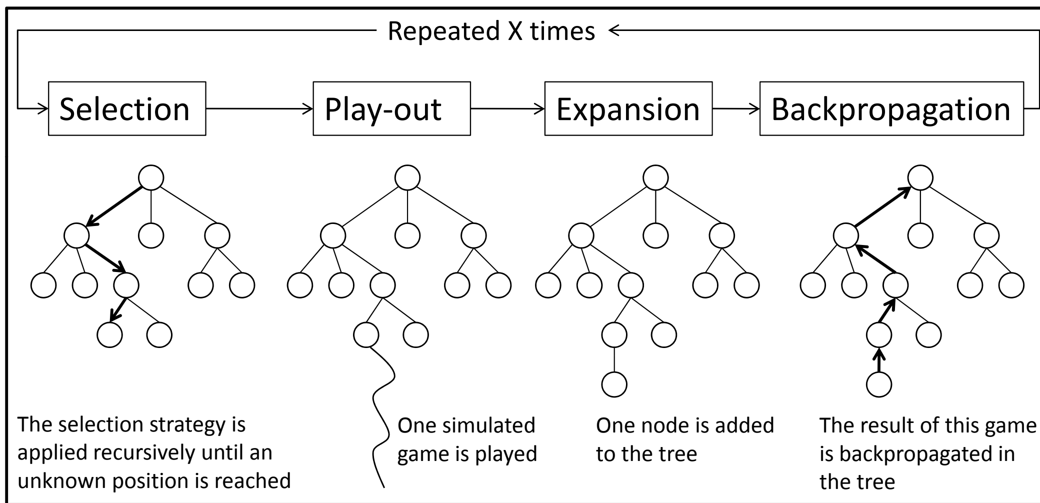


Figure 1: Four strategic steps in Monte Carlo Tree Search

2 Monte Carlo Tree Search

CADIAPLAYER [Björnsson and Finnsson, 2009; Finnsson, 2012] uses Monte Carlo Tree Search (MCTS) [Kocsis and Szepesvári, 2006; Coulom, 2007] to determine which moves to play. The advantage of MCTS over minimax-based approaches is that no evaluation function is required. This makes it especially suited for GGP in which it is difficult to come up with an accurate evaluation function. MCTS is a best-first search strategy that gradually builds up a tree in memory. Each node in the tree corresponds to a state in the game. The edges of a node represent the legal moves in the corresponding state. Moves are evaluated based on the average return of simulated games.

MCTS consists of four strategic steps [Chaslot *et al.*, 2008] which are outlined in Figure 1. (1) The *selection step* determines how to traverse the tree from the root node to a leaf node *L*. It should balance the exploitation of successful moves with the exploration of new moves. (2) In the *play-out step* a random game is simulated from leaf node *L* till the end of the game. Usually a *simulation strategy* is employed to improve the play-out [Gelly and Silver, 2007]. (3) In the *expansion step* one or more children of *L* are added. (4) In the *back-propagation step* the reward *R* obtained is back-propagated through the tree from *L* to the root node.

Below we describe how these four strategic steps are implemented in CADIAPLAYER:

1. In the *selection step* the Upper Confidence Bounds applied to Trees (UCT) algorithm [Kocsis and Szepesvári, 2006] is applied to determine which moves to select in the tree. At each node *s* move a^* selected is given by Formula 1.

$$a^* = \operatorname*{argmax}_{a \in A(s)} \left\{ Q(s,a) + C\sqrt{\frac{\ln N(s)}{N(s,a)}} \right\} \quad (1)$$

   where $N(s)$ is the visit count of *s* and $N(s,a)$ is the number of times move *a* is selected in node *s*. The first term, $Q(s,a)$ is the average return when move a is played in state s. The second term increases when state s is visited and siblings of a are selected. If a state s is visited frequently then even moves with a relatively low $Q(s,a)$ could be selected again at some point, because their second term has risen high enough. Thus, the first term supports the exploitation of successful moves while the second term establishes the exploration of infrequently visited moves. The C parameter influences the balance between exploration and exploitation. Increasing C leads to more exploration.

   If $A(s)$, the set of legal moves in state s, contains moves that are never visited before, then another selection mechanism is utilized, because these moves do not have an estimated value yet. If there is exactly one move that is not visited before, then this one is selected by default. If there are multiple moves that are not visited before, then the same simulation strategies as used in the play-out step are used to determine which move to select. In all other cases Formula 1 is applied.

2. During the *play-out step* a complete game is simulated. The most basic approach is to play plain random moves. However, the play-outs can be improved significantly by playing quasi-random moves according to a *simulation strategy* [Gelly and Silver, 2007]. The aim is to improve the performance of the already existing CADIAPLAYER by introducing new simulation strategies. These simulation strategies are described in Section 3.

3. In the *expansion step* nodes are added to the tree. In CADIAPLAYER, only one node per simulation is added [Coulom, 2007]. This node corresponds to the first position encountered outside the tree. Adding only one node after a simulation prevents excessive memory usage, which could occur when the simulations are fast.

4. In the *back-propagation step* the reward obtained in the play-out is propagated backwards through all the nodes on the path from the leaf node *L* to the root node. The *Q(s,a)* values of all state-move pairs on this path are updated with the just obtained reward. In GGP the reward

lies in the range [0, 100].

More details about the implementation of CADIAPLAYER can be found in Finnsson [2012].

3 Simulation Strategies

This section explains the simulation strategies employed in the experiments. Subsection 3.1 explains the Move-Average Sampling Technique used by CADIAPLAYER when it won the AAAI 2008 GGP competition. Subsection 3.2 explains the N-Gram Selection Technique (NST).

3.1 Move-Average Sampling Technique

The Move-Average Sampling Technique (MAST) [Finnsson and Björnsson, 2008; Finnsson, 2012] is based on the principle that moves good in one state are likely to be good in other states as well. The history heuristic [Schaeffer, 1983], which is used to order moves in $\alpha\beta$ search [Knuth and Moore, 1975], is based on the same principle. For each move a, a global average $Q_h(a)$ is kept in memory. It is the average of the returned rewards of the play-outs in which move a occurred. These values are utilized for selecting moves in the play-out. Furthermore, if in the MCTS tree a node has more than one unvisited legal move then the $Q_h(a)$ values of these unvisited moves are employed by Gibbs measure [Casella and George, 1992] to determine which move to select:

$$P(s,a) = \frac{e^{Q_h(a)/\tau}}{\sum_{b \in A(s)} e^{Q_h(b)/\tau}} \quad (2)$$

$P(s,a)$ is the probability that move a will be selected in state or node s. Moves with a higher $Q_h(a)$ value are more likely to be selected. How greedy the selection is can be tuned with the τ parameter. In order to bias the selection of unexplored moves the initial $Q_h(a)$ value is set to the maximum possible score of 100.

3.2 N-Gram Selection Technique

The N-Gram Selection Technique (NST) was introduced by Tak *et al.* [2012]. NST keeps track of move sequences as opposed to single moves as in MAST. Tak *et al.* [2012] showed that NST often outperforms MAST in GGP.

A method similar to NST is applied successfully in Havannah [Stankiewicz, 2011; Stankiewicz *et al.*, 2012]. Furthermore, NST also bears some resemblance with the simulation strategy introduced by Rimmel and Teytaud [2010], which is based on a tiling of the space of Monte Carlo simulations.

NST is based on N-Gram models, which were invented by Shannon [1951]. An N-Gram model is a statistical model to predict the next word based on the previous N-1 words. N-Grams are often employed in statistical language processing [Manning and Schütze, 1999]. N-Grams also have been applied in various research on computer games. For instance, N-Grams can be used to predict the next move of the opponent [Laramée, 2002; Millington, 2006]. Whereas, Nakamura [1997] uses N-Grams to extract opening moves. N-Grams can also be employed for move ordering [Kimura *et al.*, 2011; Hashimoto, 2011]. Otsuki [2005] applied them in forced move detection.

The N-Grams in NST consist of consecutive move sequences z of length 1, 2 and 3. Similar to MAST the average of the returned rewards of the play-outs is accumulated. However, the average reward for a sequence z, here called $R(z)$, is kept also for longer move sequences as opposed to single moves only.

The N-Grams are formed as follows. After each simulation, starting at the root of the tree, for each player all move sequences of length 1, 2 and 3 that appeared in the simulated game are extracted. The averages of these sequences are updated with the obtained reward from the simulation. It is not checked whether the same move sequence occurred more than once in the simulation. Thus, if there are m occurrences of the same move sequence, then the average of this sequence is updated m times. For each player the extracted move sequences are stored separately.

The move sequences consist of moves from the current player and moves from the opponent(s). The role numbers $0, 1, 2, \cdots, n-1$, which are assigned to the players at the beginning of a game with n players, are employed in order to determine the move of which opponent to include in the sequences. Suppose that the current player has role number i and there are n players, then the sequences are constructed as follows. A sequence of length 1 consists of just one move of the current player. A sequence of length 2 starts with a move of player with role $(i+n-1) \bmod n$ and ends with a move of the current player. A sequence of length 3 starts with a move of player with role $(i+n-2) \bmod n$, followed by a move of the player with role $(i+n-1) \bmod n$ and ends with a move made by the current player. The moves in these sequences are consecutive moves.

Figure 2 gives an example of a play-out. At each step, both players have to choose a move, because all games in GGP are assumed to be simultaneous-move games. The example given here concerns a turn-taking, two-player game, which means that at each step one of the players can only play the *noop* move. The example shows that these *noop* moves are included in the sequences, because NST handles them as regular moves. This does not cause any problem, because these move sequences will only be used during move selection when the player is not really on turn and has the only option of choosing the *noop* move. Therefore, the move sequences containing *noop* moves do not negatively influence the decision process during the play-out.

If the game is truly simultaneous, then at each step all players choose an actual move instead of some players having to choose the *noop* move like in turn-taking games. As explained above, NST includes only one move per step in its sequences. This means that for an n-player simultaneous game, moves of $n-1$ players are ignored each step. Another possibility would have been to include the moves of all players at each step, but that would lead to too specific sequences. The disadvantage of such specific sequences is that fewer statistical samples can be gathered about them, because they occur much more rarely.

In the play-out, and at the nodes of the MCTS tree containing unvisited legal moves, the N-Grams are used to determine which move to select. For each legal move, the player determines which sequence of length 1, which se-

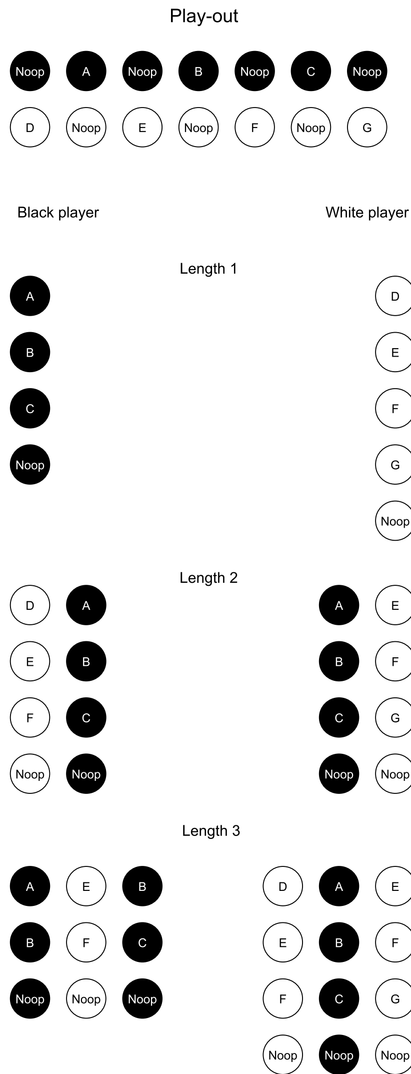


Figure 2: Extracted move sequences from play-out

quence of length 2 and which sequence of length 3, would occur when that move is played. The sequence of length 1 is just the move itself. The sequence of length 2 is the move itself appended to the last move played by player with role $(i + n - 1) \bmod n$. The sequence of length 3 is the move itself appended to the previous last move played by player with role $(i + n - 2) \bmod n$ and the last move played by the player with role $(i + n - 1) \bmod n$. Thus, in total three sequences could occur. The player then calculates a score for a move by taking the average of the $R(z)$ values stored for these sequences. In this calculation, the $R(z)$ values for the move sequences of length 2 and length 3 are only taken into account if they are visited at least k times. In the performed experiments, $k = 7$. This value was determined by manual tuning.

If a move has been played at least once, but the sequences of length 2 and length 3 occurred fewer than k times, then the $R(z)$ value of the move sequence of length 1 (which is the move itself) will be returned.

If a move has never been played before, then no move sequences exist and the calculation outlined above is not possible. In that case the score is set to the maximum possible value of 100 to bias the selection towards unexplored moves.

In this manner, a score $T(a)$ is assigned to each legal move a in a given state. These scores are then used with ϵ-greedy [Sutton and Barto, 1998; Sturtevant, 2008] to determine which move to select. With a probability of $1 - \epsilon$ the move with the highest $T(a)$ value is selected, and with a probability of ϵ a legal move is chosen uniformly at random.

4 Decay Factor

The information gathered by NST and MAST are kept between successive searches. On the one hand this reuse of information may bolster the simulation strategy as it is immediately known what the strong moves are in the play-out. This is especially important in GGP as the number of simulations to gather information is quite low. On the other hand this information can become outdated as moves that are strong in one phase of the game are weak in another phase. Moreover, statistics can be mostly gathered for a particular part of the search tree that subsequently is not reached as the opponent moves differently from what was anticipated. Therefore we propose to introduce a decay factor. For NST in particular, it would mean that the $R(z)$ values, which store the average rewards per move sequence, should change based on the current state of the game. A decay factor would cause more recent simulations to have added weight on the $R(z)$ values. It is implemented such that after a move is applied in the actual game, the visit count of all the stored sequences is multiplied by a decay factor $\gamma \in [0, 1]$. A decay factor of 1 means that there is no decay. During the search no decaying takes place — only after an actual move is made in the current game state are the visit counts of the corresponding $R(z)$ values discounted. A similar scheme is applied for the $Q_h(a)$ values in MAST.

We remark that Stankiewicz [2011] showed that for NST a decay factor of 0 performs best in *Havannah*. A decay factor of 0 means that the results are reset between each move. NST with a decay factor of 0 resembles the Last-Good-Reply Policy (LGRP) [Drake, 2009; Baier and Drake, 2010]. In LGRP the most recent successful replies are stored and a reply is removed from memory when it is no longer successful.

We have tried to apply a decay factor to UCT as well, without success. A similar approach, called Discounted UCB, was evaluated by Hashimoto *et al.* [2012] in Othello, Havannah, and Go, but did not improve performance.

5 Experimental Setup

The N-Gram adjustments are implemented in CADIAPLAYER in order to investigate the effectiveness for GGP. This program is called CP_{NST}. The program using MAST instead of NST is called CP_{MAST}. In Subsection 5.1 brief descriptions are given of the games used in the experiments. In Subsection 5.2 the setup of the experiments is described.

5.1 Games

Below an overview is given of the games used in the experiments. Note that most of the classic games enlisted below are usually a variant of its original counterpart. The most common adjustments are a smaller board size and a bound on the number of steps. The following two-player, turn-taking games are used:

- *Zhadu* is a strategy game consisting of a placement phase and a movement phase. The first piece that is captured, determines what other piece need to be captured in order to win.
- In *GridGame* each player has to find a book, a candle and a bell. A score between 0 and 100 is given, based on how many items where found.
- *3DTicTacToe* is a variant on Tic-Tac-Toe. It is played on a 4×4×4 cube and the goal is to align four pieces in a straight line.
- *TTCC4* stands for: *TicTacChessCheckersFour*. Each player has a pawn, a checkers piece and a knight. The aim of each player is to form a line of three with its own pieces.
- *Connect5* is played on an 8×8 board and the player on turn has to place a piece in an empty square. The aim is to place five consecutive pieces of the own color horizontally, vertically or diagonally, like *Five-in-a-Row*.
- *Checkers* is played on an 8×8 board and the aim is to capture pieces of the opponent.
- *Breakthrough* is played on an 8×8 board. Each player starts on one side of the board and the goal is to move one of their pieces to the other side of the board.
- *Knightthrough* is almost the same as *Breakthrough*, but is played with chess knights.
- *Othello* is played on an 8×8 board. Each turn a player places a piece of its own color on the board. This will change the color of some of the pieces of the opponent. The aim is to have the most pieces of the own color on the board at the end of the game.
- *Skirmish* is played on an 8×8 board with different kind of pieces, namely: bishops, pawns, knights and rooks. The aim is to capture as many pieces from the opponent as possible, without losing to many pieces either.
- *Merrills* is also known as Nine Men's Morris. Both players start with nine pieces each. In order to win, pieces of the opponent need to be captured. The objective is to form a horizontal or vertical line of three pieces, called a mill, because pieces in a mill cannot be captured. The game ends when one player has only two pieces left.
- *Quad* is played on a 7×7 board. Each player has 'quad' pieces and 'white' pieces. The purpose of the 'white' pieces is to form blockades. The player that forms a square consisting of four 'quad' pieces wins the game.
- *Sheep and Wolf* is an asymmetrical game played on an 8×8 board. One player controls the Sheep and the other player controls the Wolf. The game ends when none of the players can move or when the Wolf is behind the Sheep. In this case, if the Wolf is not able to move the Sheep wins. Otherwise, the Wolf wins.

The following three-player, turn-taking games are used:

- *Farmers* is a trading game. In the beginning of the game, each player gets the same amount of money. They can use the money to buy cotton, cloth, wheat and floor. It is also possible to buy a farm or factory and then the player can produce its own products. The player that has the most money at the end of the game wins.
- *TTCC43P* is the same as *TTCC4*, but then with one extra player.
- *Chinese Checkers 3P* is played on a star shaped board. Each player starts with three pieces positioned in one corner. The aim is to move all these three pieces to the empty corner at the opposite side of the board. This is a variant of the original *Chinese Checkers*, because according to the standard rules each player has ten pieces instead of three.

The following two-player, simultaneous-move games are used:

- *Battle* is played on an 8×8 board. Each player has 20 disks. These disks can move one square or capture an opponent square next to them. Instead of a move, the player can choose to defend a square occupied by their piece. If an attacker attacks such a defended square, the attacker will be captured. The goal is to be the first player to capture 10 opponent disk.
- *Chinook* is a variant of *Breakthrough* where two independent games are played simultaneously. One game on the white squares and another one on the black squares. Black and White move their pieces simultaneously like Checkers pawns. As in Breakthrough, the first player that reaches the opposite side of the board wins the game.
- In *Runners* each turn both players decide how many steps they want to move forward or backward. The aim is to reach the goal location before the opponent does.
- *Pawn Whopping* is similar to *Breakthrough*, but with slightly different movement and is simultaneous.

These games were chosen because they are used in several previous CADIAPLAYER experiments [Finnsson, 2007; Finnsson and Björnsson, 2008; 2009; Björnsson and Finnsson, 2009; Finnsson and Björnsson, 2010; 2011; Finnsson, 2012]. *Pawn Whopping* was used during the *German Open in GGP* of 2011 [Kissmann and Federholzner, 2011]. Furthermore, this selection contains different types of games. Namely, two-player games, multi-player games, constant-sum games and general-sum games (e.g. GridGame, Skirmish, Battle, Chinook, Farmers and ChineseCheckers3P).

5.2 Setup

In all experiments two variants of CADIAPLAYER are matched against each other. The ϵ and k parameters of the NST simulation strategy are set to 0.2 and 7, respectively.

Table 1: Win % of CP_{NST} with different values of γ against CP_{NST} with $\gamma = 1$, startclock=60s, playclock=30s.

Game	$\gamma = 0$	$\gamma = 0.2$	$\gamma = 0.4$	$\gamma = 0.6$	$\gamma = 0.8$
Zhadu	26.6 (±3.75)	32.4 (±4.05)	36.5 (±3.80)	47.0 (±3.67)	**47.4** (±4.97)
GridGame	49.4 (±5.38)	49.9 (±3.43)	**50.5** (±4.11)	49.8 (±3.43)	49.3 (±4.58)
3DTicTacToe	66.5 (±4.97)	**69.0**(±3.53)	66.2 (±4.29)	61.8 (±4.82)	58.2 (±4.85)
TTCC4	27.5 (±4.75)	44.4 (±5.04)	47.9 (±4.51)	**52.5** (±5.63)	51.7 (±4.33)
Connect5	61.1 (±4.72)	**69.1** (±4.19)	65.7 (±4.02)	66.2 (±3.69)	59.4 (±4.99)
Checkers	45.6 (±4.76)	54.0 (±4.72)	**63.3** (±4.41)	60.8 (±5.38)	62.6 (±5.46)
Breakthrough	37.3 (±5.22)	41.9 (±4.36)	45.5 (±4.25)	44.6 (±5.18)	**53.6** (±5.62)
Knightthrough	46.4 (±5.62)	38.1 (±4.95)	43.6 (±4.64)	44.1 (±5.58)	**54.6** (±5.60)
Othello	36.1 (±5.29)	44.4 (±4.17)	45.2 (±4.02)	**49.1** (±5.46)	48.1 (±5.58)
Skirmish	51.0 (±5.28)	49.1 (±5.28)	53.2 (±4.85)	**55.1** (±4.40)	52.2 (±4.48)
Merrills	58.3 (±4.32)	58.6 (±5.05)	58.3 (±3.89)	**60.7** (±5.02)	55.6 (±5.22)
Quad	61.5 (±3.87)	**68.7** (±3.63)	67.2 (±3.37)	65.3 (±3.11)	60.4 (±4.25)
Sheep and Wolf	44.3 (±4.11)	44.0 (±3.41)	47.2 (±4.08)	49.0 (±3.46)	**52.2** (±5.47)
Farmers	44.9 (±4.11)	52.7 (±2.62)	**53.0** (±3.13)	50.3 (±2.62)	50.3 (±4.10)
TTCC43P	53.2 (±5.65)	**56.2** (±4.31)	54.6 (±3.96)	54.5 (±3.59)	**56.2** (±5.61)
ChineseCheckers3P	41.4 (±5.09)	50.7 (±4.24)	**53.2** (±4.67)	51.3 (±4.29)	51.0 (±5.66)
Battle	56.6 (±5.32)	**65.6** (±4.46)	63.8 (±4.15)	64.5 (±5.03)	59.3 (±5.15)
Chinook	45.7 (±5.30)	55.0 (±4.75)	**57.3** (±4.36)	56.9 (±5.31)	54.4 (±5.36)
Runners	**55.8** (±4.73)	53.4 (±4.66)	49.7 (±4.66)	49.5 (±4.67)	52.1 (±3.98)
Pawn Whopping	47.2 (±2.72)	50.2 (±2.72)	**51.5** (±2.71)	50.0 (±2.71)	50.3 (±2.30)

Their values were determined by manual tuning. The τ parameter of the Gibbs measure used in CADIAPLAYER was left unchanged to its preset value of 10. The program using NST cuts off simulations that take more than 400 moves. The program using MAST does not cut off.

In the experiments two different time settings are used. When tuning γ the startclock is set to 60s and the playclock is set to 30s. In the validation experiments, the startclock is set to 70s and the playclock is set to 40s. Different time settings are used, because on the one hand we want to have a high number of simulations per move, but on the other hand it takes much computation time to tune γ.

In all experiments, the programs switch roles such that no one has any advantage. For the two-player games, there are two possible configurations. For the three-player games, there are eight possible configurations, where two of them consist of three times the same player. Therefore, only six configurations are employed in the experiments [Sturtevant, 2008]. All experiments are performed on a computer consisting of 64 AMD Opteron 2.2 Ghz cores.

6 Experimental Results

In the experiments it is examined how different decay factors perform. The original N-Gram player, CP_{NST}, is matched against CP_{NST} with a decay factor. After determining the best decay factor, CP_{NST} is matched against CP_{MAST} to further validate whether the decay factor is a genuine improvement. Finally, CP_{MAST} with a decay factor is matched against CP_{NST} to investigate whether decaying is beneficial for MAST as well. All tables show both a win rate, averaged over at least 300 games, and a 95% confidence interval. The win rate is calculated as follows. For the two-player games, each game won gives a score of 1 point and each game that ends in a draw results in a score of $\frac{1}{2}$ point. The win rate is the sum of these points divided by the total number of games played. For the three-player games, a similar calculation is performed except the draws are counted differently. If all three players obtained the same reward, then the draw is counted as $\frac{1}{3}$ point. If two players obtained the same, highest reward, the draw is counted as $\frac{1}{2}$ point for the corresponding players.

Table 2: Win % of CP_{NST} with $\gamma \in \{1; 0.6\}$ against CP_{MAST} with $\gamma = 1$, startclock=70s, playclock=40s

Game	$\gamma = 1$	$\gamma = 0.6$
Zhadu	74.9 (±4.51)	**75.5** (±4.39)
GridGame	52.3 (±3.79)	**52.8** (±4.52)
3DTicTacToe	73.3 (±3.87)	**80.4** (±3.59)
TTCC4	**85.4** (±2.18)	84.4 (±1.69)
Connect5	70.4 (±3.57)	**78.9** (±3.79)
Checkers	68.9 (±5.14)	**80.0** (±4.38)
Breakthrough	63.7 (±3.69)	**72.3** (±2.82)
Knightthrough	47.7 (±5.29)	**50.0** (±5.30)
Othello	**67.4** (±4.54)	67.0 (±4.55)
Skirmish	69.6 (±5.01)	**70.1** (±5.03)
Merrills	44.6 (±2.81)	**50.9** (±2.82)
Quad	79.1 (±2.96)	**92.3** (±2.30)
Sheep and Wolf	61.1 (±3.94)	**61.3** (±4.73)
Farmers	72.2 (±2.64)	**73.1** (±3.11)
TTCC43P	53.2 (±3.66)	**58.1** (±2.43)
ChineseCheckers3P	**57.6** (±4.87)	55.1 (±5.32)
Battle	19.2 (±4.01)	**29.8** (±4.69)
Chinook	73.7 (±2.88)	**79.4** (±1.96)
Runners	35.7 (±4.62)	**36.7** (±4.60)
Pawn Whopping	**52.2** (±2.80)	51.3 (±2.80)

6.1 Decay Factor in NST

Table 1 shows the win rate of CP_{NST} with decay versus CP_{NST} without decay. Note that no decay means that $\gamma = 1$. The win rates in bold indicate that they are the highest win rate of their row. The results show that decay may improve the program. Furthermore, the results demonstrate that simply resetting the NST statistics each move (which means $\gamma = 0$) can decrease the performance significantly in some games

Table 3: Win % of CP_{MAST} with different values of γ against CP_{MAST} with $\gamma = 1$, startclock=60s, playclock=30s

Game	$\gamma = 0$	$\gamma = 0.2$	$\gamma = 0.6$
Zhadu	52.8 (±3.73)	51.2 (±4.44)	**54.2** (±3.78)
GridGame	**50.0** (±3.56)	**50.0** (±4.01)	49.9 (±3.26)
3DTicTacToe	88.0 (±1.95)	**92.4** (±2.00)	87.7 (±2.10)
TTCC4	48.3 (±3.65)	50.4 (±4.54)	**52.0** (±3.81)
Connect5	**77.6** (±3.04)	76.4 (±3.81)	68.8 (±3.44)
Checkers	59.1 (±4.43)	**67.4** (±4.58)	65.0 (±4.45)
Breakthrough	53.2 (±5.03)	53.0 (±4.11)	**58.0** (±4.91)
Knightthrough	53.2 (±3.92)	**54.3** (±3.22)	52.2 (±4.21)
Othello	43.4 (±5.13)	44.9 (±4.22)	**46.2** (±5.11)
Skirmish	49.6 (±4.08)	48.6 (±5.22)	**51.6** (±4.40)
Merrills	53.5 (±3.71)	**54.7** (±3.98)	52.1 (±5.23)
Quad	72.2 (±2.86)	**77.8** (±3.17)	73.9 (±2.80)
Sheep and Wolf	50.0 (±3.92)	**51.2** (±4.40)	49.2 (±3.59)
Farmers	48.3 (±2.90)	**54.3** (±3.25)	53.9 (±4.95)
TTCC43P	51.9 (±3.37)	49.6 (±4.37)	**54.0** (±3.64)
ChineseCheckers3P	52.9 (±4.06)	**53.1** (±5.20)	51.8 (±4.36)
Battle	50.5 (±3.57)	50.2 (±2.93)	**52.6** (±3.82)
Chinook	53.1 (±3.70)	**62.0** (±4.64)	60.2 (±3.95)
Runners	51.8 (±4.42)	**53.8** (±5.03)	52.5 (±4.08)
Pawn Whopping	49.9 (±2.78)	**50.1** (±3.13)	49.5 (±4.78)

(i.e., Zhadu, TTCC4, Breakthrough, Othello, ChineseCheckers3P). The best results are obtained for $\gamma = 0.4$ and $\gamma = 0.6$. In order to validate the results, the CP_{NST} with $\gamma = 0.6$ is matched against CP_{MAST}. The parameter $\gamma = 0.6$ is regarded as the best setting, because unlike for $\gamma = 0.4$ it never performs really worse than the original program.

As a reference experiment, CP_{NST} with $\gamma = 1$ plays against CP_{MAST}. The results of the validation are given in Table 2. Again, win rates in bold indicate that they are the highest win rate of their row. The table reveals that in nine games the performance of the program with a decay factor of $\gamma = 0.6$ is significantly better than the program without a decay factor (i.e., 3DTicTacToe, Connect5, Checkers, Breakthrough, Merrills, Quad, TTCC43P, Battle, Chinook). In the other games, the performance is approximately equal. We suspect that especially games where the quality of a move highly depends on the game state and current phase of the game, can be improved by using a decay factor. Games without this property may profit less from a decay factor. This line of reasoning is supported by the results. Namely in Othello the decay factor did not improve the results. In this game there are certain moves that are always good independent of the game state, like placing a stone in the corner.

Also, as reported previously by Tak *et al.* [2012], we see that NST is mostly superior to MAST as a general move-selection strategy, with the notable exceptions of the simultaneous-move games Battle and Runners. Both these games could be classified as greedy as opposed to strategic, that is, the same greedy action is often the best independent of the current state and the recent move history (for example, in Runners the furthest advancing action is the best one to take in all game states); such situations are best-case scenarios for MAST.

6.2 Decay Factor in MAST

In the final series of experiments CP_{MAST} with decay was matched against CP_{MAST} without decay. The results are shown in Table 3. The win rates in bold indicate that they are the highest win rate of their row. Again, we see that a decay factor may improve the program. In contrast with NST, simply resetting the statistics each move (which means $\gamma = 0$) has approximately the same or better performance than no decay. The table reveals that in five games the performance of the program with a decay factor of $\gamma = 0.2$ is significantly better than the program without a decay factor (i.e., 3DTicTacToe, Connect5, Checkers, Quad, Chinook). The performance stays approximately the same in the other games. Furthermore, notice that there is overlap with NST in the games where decaying is effective (3DTicTacToe, Connect5, Checkers, Quad). This can be explained by the fact that the N-Grams of length 1 are in essence the same as MAST, which means that NST will behave similar as MAST when these techniques are changed in the same way (e.g with a decay factor).

7 Conclusions and Future Work

In this paper we proposed to apply a decay factor to NST and MAST. The experiments revealed that a decay factor of 0.6 and 0.2 for NST and MAST, respectively, improves the program significantly. It appears that decaying works especially well in games where it depends heavily on the current game state which moves are strong (as opposed for a move to be globally good). Furthermore, the experiments revealed that simply resetting the NST statistics after each move harms the performance in some games, while for MAST it does not decrease the performance.

As future work it is interesting to investigate how a decay factor can be applied to the UCT values. We already experimented with a decay factor on the UCT values, but without

success so far. Also of interest is to find out whether there are other methods that can be used to decay the UCT values successfully. Related work is the Discounted UCB, but this was also unsuccessful [Hashimoto *et al.*, 2012]. Another direction of future work would be to investigate whether a decay factor also works within the search iself.

Acknowledgements

This work is funded by the Netherlands Organisation for Scientific Research (NWO) in the framework of the project GoGeneral, grant number 612.001.121.

References

[Baier and Drake, 2010] H. Baier and P.D. Drake. The Power of Forgetting: Improving the Last-Good-Reply Policy in Monte Carlo Go. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):303 –309, 2010.

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

[Casella and George, 1992] G. Casella and E.I. George. Explaining the Gibbs Sampler. *The American Statistician*, 46(3):167–174, 1992.

[Chaslot *et al.*, 2008] G.M.J.-B. Chaslot, M.H.M. Winands, J.W.H.M. Uiterwijk, H.J. van den Herik, and B. Bouzy. Progressive Strategies for Monte-Carlo Tree Search. *New Mathematics and Natural Computation*, 4(3):343–357, 2008.

[Clune, 2007] J. Clune. Heuristic Evaluation Functions for General Game Playing. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence*, pages 1134–1139, Menlo Park, California, 2007. The AAAI Press.

[Coulom, 2007] R. Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In H.J. van den Herik, P. Ciancarini, and H.H.L.M. Donkers, editors, *CG 2006*, volume 4630 of *LNCS*, pages 72–83, Berlin-Heidelberg, Germany, 2007. Springer-Verlag.

[Drake, 2009] P. Drake. The Last-Good-Reply Policy for Monte-Carlo Go. *ICGA Journal*, 32(4):221–227, 2009.

[Finnsson and Björnsson, 2008] H. Finnsson and Y. Björnsson. Simulation-Based Approach to General Game Playing. In D. Fox and C.P. Gomes, editors, *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence*, pages 259–264, Menlo Park, California, 2008. AAAI Press.

[Finnsson and Björnsson, 2009] H. Finnsson and Y. Björnsson. Simulation Control in General Game Playing Agents. In *The IJCAI Workshop on General Game Playing (GIGA'09)*, pages 21–26, Pasadena, California, 2009.

[Finnsson and Björnsson, 2010] H. Finnsson and Y. Björnsson. Learning Simulation Control in General Game-Playing Agents. In M. Fox and D. Poole, editors, *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence*, pages 954–959, Menlo Park, California, 2010. AAAI Press.

[Finnsson and Björnsson, 2011] H. Finnsson and Y. Björnsson. CadiaPlayer: Search Control Techniques. *KI Journal*, 25(1):9–16, 2011.

[Finnsson, 2007] H. Finnsson. CADIA-Player: A General Game Playing Agent. Master's thesis, School of Computer Science, Reykjavík University, Reykjavík, Iceland, 2007.

[Finnsson, 2012] H. Finnsson. *Simulation-Based General Game Playing*. PhD thesis, School of Computer Science, Reykjavík University, Reykjavík, Iceland, 2012.

[Gelly and Silver, 2007] S. Gelly and D. Silver. Combining Online and Offline Knowledge in UCT. In Z. Ghahramani, editor, *Proceedings of the 24th International Conference on Machine Learning*, ICML '07, pages 273–280, New York, New York, 2007. ACM.

[Hashimoto *et al.*, 2012] J. Hashimoto, A. Kishimoto, K. Yoshizoe, and K. Ikeda. Accelerated UCT and Its Application to Two-Player Games. In H.J. van den Herik and A. Plaat, editors, *Advances in Computer Games*, volume 7168 of *LNCS*, pages 1–12, Berlin-Heidelberg, Germany, 2012. Springer-Verlag.

[Hashimoto, 2011] J. Hashimoto. *A Study on Game-Independent Heuristics in Game-Tree Search*. PhD thesis, School of Information Science, Japan Advanced Institute of Science and Technology, Kanazawa, Japan, 2011.

[Kimura *et al.*, 2011] T. Kimura, T. Ugajin, and Y. Kotani. Bigram Realization Probability for Game Tree Search. In *2011 International Conference on Technologies and Applications of Artificial Intelligence (TAAI)*, pages 260–265, 2011.

[Kissmann and Federholzner, 2011] P. Kissmann and T. Federholzner. German Open in GGP 2011, 2011. http://www.tzi.de/∼kissmann/ggp/go-ggp/classical/games/.

[Knuth and Moore, 1975] D.E. Knuth and R.W. Moore. An Analysis of Alpha-Beta Pruning. *Artificial Intelligence*, 6(4):293 – 326, 1975.

[Kocsis and Szepesvári, 2006] L. Kocsis and C. Szepesvári. Bandit Based Monte-Carlo Planning. In J. Fürnkranz, T. Scheffer, and M. Spiliopoulou, editors, *Proceedings of the EMCL 2006*, volume 4212 of *LNCS*, pages 282–293, Berlin-Heidelberg, Germany, 2006. Springer-Verlag.

[Kozelek, 2009] T. Kozelek. Methods of MCTS and the Game Arimaa. Master's thesis, Department of Theoretical Computer Science and Mathematical Logic, Charles University, Prague, Czech Republic, 2009.

[Laramée, 2002] F.D. Laramée. Using N-Gram Statistical Models to Predict Player Behavior. In S. Rabin, editor, *AI Programming Wisdom*, volume 1, chapter 11, pages 596–601. Charles River Media, 2002.

[Manning and Schütze, 1999] C.D. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing*. The MIT Press, USA, 1999.

[Méhat and Cazenave, 2010] J. Méhat and T. Cazenave. Ary, a General Game Playing Program. In *XIIIth Board Games Studies Colloquium*, Paris, France, 2010.

[Millington, 2006] I. Millington. *Artificial Intelligence for Games*, chapter 7, pages 580–591. Morgan Kaufmann, first edition, 2006.

[Nakamura, 1997] T. Nakamura. Acquisition of Move Sequence Patterns from Game Record Database Using N-gram Statistics. In *Proceedings of the 4th Game Programming Workshop in Japan*, pages 96–105, 1997.

[Otsuki, 2005] T. Otsuki. Extraction of 'Forced Move' from N-Gram Statistics. In *Proceedings of the 10th Game Programming Workshop in Japan*, pages 89–96, 2005.

[Rimmel and Teytaud, 2010] A. Rimmel and F. Teytaud. Multiple Overlapping Tiles for Contextual Monte Carlo Tree Search. In C. Di Chio, S. Cagnoni, C. Cotta, M. Ebner, A. Ekárt, A. Esparcia-Alcazar, C-K. Goh, J. Merelo, F. Neri, M. Preuß, J. Togelius, and G. Yannakakis, editors, *Applications of Evolutionary Computation*, volume 6024 of *LNCS*, pages 201–210, Berlin-Heidelberg, Germany, 2010. Springer-Verlag.

[Schaeffer, 1983] J. Schaeffer. The History Heuristic. *ICCA Journal*, 6(3):16–19, 1983.

[Schiffel and Thielscher, 2007a] S. Schiffel and M. Thielscher. Automatic Construction of a Heuristic Search Function for General Game Playing. In *Seventh IJCAI International Workshop on Nonmontonic Reasoning, Action and Change (NRAC07)*, 2007.

[Schiffel and Thielscher, 2007b] S. Schiffel and M. Thielscher. Fluxplayer: A Successful General Game Player. In *Proceedings of the Twenty-Second AAAI Conference on Artificial Intelligence*, pages 1191–1196, Menlo Park, California, 2007. The AAAI Press.

[Shannon, 1951] C.E. Shannon. Prediction and Entropy of Printed English. *The Bell System Technical Journal*, 30(1):50–64, 1951.

[Stankiewicz *et al.*, 2012] J.A. Stankiewicz, M.H.M. Winands, and J.W.H.M. Uiterwijk. Monte-Carlo Tree Search Enhancements for Havannah. In H.J. van den Herik and A. Plaat, editors, *Advances in Computer Games (ACG 13)*, volume 7168 of *LNCS*, pages 60–71, Berlin-Heidelberg, Germany, 2012. Springer-Verlag.

[Stankiewicz, 2011] J.A. Stankiewicz. Knowledge-Based Monte-Carlo Tree Search in Havannah. Master's thesis, Department of Knowledge Engineering, Maastricht University, Maastricht, The Netherlands, 2011.

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

[Sutton and Barto, 1998] R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. Adaptive Computation and Machine Learning. The MIT Press, Cambridge, Massachusetts, 1998.

[Tak *et al.*, 2012] M.J.W. Tak, M.H.M. Winands, and Y. Björnsson. N-Grams and the Last-Good-Reply Policy Applied in General Game Playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(2):73–83, 2012.

High Speed Forward Chaining for General Game Playing

Michael Schofield
CSE, The University of New South Wales, Sydney, Australia
mschofield@cse.unsw.edu.au

Abdallah Saffidine
LAMSADE, Université Paris-Dauphine, Paris, France
abdallah.saffidine@dauphine.fr

Abstract

General Game Playing demands that an AI system be capable of interpreting a set of rules for a previously unseen game and reason about the game state as efficiently as possible. In simulation based reasoners, the number of states that can be visited in a fixed time limit is paramount. One technique for calculating each game state is Forward Chaining; where the system calculates all of the relations that can be calculated from the current state and uses that as a basis for the next state.

In this work we progress some earlier work on Forward Chaining and propose two additional features. Firstly the augmentation of rule processing using reference tables to facilitate high speed instantiation of ground relations into a rule, and secondly an empirical hypothesis ordering strategy utilising data collected from the operation of the system to optimise its performance. This paper proposes and defines these additional features and presents experimental data to support their use.

1 Introduction

Most General Game Playing (GGP) programs can be decomposed into a rule engine and a search engine. The former processes the rules of the game to determine which actions are possible, which game states are terminal, and what the associated utility for each player is. The search engine uses this information together with some tree search algorithms such as variants of A*, alpha-beta, or Monte Carlo Tree Search to decide what action to perform next.

The larger the explored state space, the more informed and better the submitted decisions. However, GGP competitions require that playing programs submit an action within a specific time, say 30 seconds. As a result, the strength of a playing program crucially depends on the speed of the rule engine and that of the search engine. Given that all state-of-the-art algorithms for search in GGP are data-intensive, the speed bottleneck lies on the side of the rule engine. These algorithms include Monte Carlo Tree Search [Finnsson and Björnsson, 2008], Nested Monte Carlo Search [Méhat and Cazenave, 2010], as well as depth-first search [Schiffel and Thielscher, 2007] in the classical track of the GGP competition. In the Imperfect Information track, the Perfect Information Monte Carlo Sampling approach [Long *et al.*, 2010; Schofield *et al.*, 2012] puts even more stress on the rule engine.

A popular measure of raw speed performance of rule engine is the number of random simulations (so-called *playouts*) the engine can generate in given game in a second. As example of the large gap between domain specific engines and GGP engine, one can observe that an optimized program for 19×19 Go can perform millions of simulations per seconds while typical GGP program perform much less than 1000 simuations per seconds on 7×6 Connect Four, a much shorter and simpler game [Saffidine and Cazenave, 2011].

Several approaches have been put forward to process game rules written in the Game Description Language (GDL). The *classic* approach is to use a Prolog engine such as YAP [Costa *et al.*, 2006] to interpret the rules. To this end, a syntactic translation from GDL to Prolog is almost sufficient. While the speed of the resulting engine is far from ideal, the easy setup makes this approach the most popular one by far among competition programs.

A similar but more involved approach is to compile GDL rules to a lower-level language such as C++ or Java so that the resulting program will simulate Prolog's SLD resolution [Waugh, 2009; Möller *et al.*, 2011]. This *SLD compilation* approach leads to programs that are up to an order of magnitude faster than with the classic approach. However, practical use of such a model is hindered by the fact that actual implementations do not handle the full range of the Game Description Language, and in particular nested function constants are typically not supported.

Grounding the rules refers to the process of transforming a game description involving variables into a an equivalent description where variables have been replaced by possible instantiations. Not only is grounding the game rules necessary to apply answer-set programming techniques for solving single agent games [Thielscher, 2009], but a ground description is often faster to execute and interpret with SLD resolution than the corresponding original description. In particular, *propositional automata* can be used when the description is ground [Cox *et al.*, 2009]. The main problem with this approach is that it can lead to an exponential blow-up in the size of the description. Most engines based on grounding fall back to a classic prolog interpreter when the game is too large to

be ground. Techniques have been put forward to widen the range of games that can be efficiently ground [Kissmann and Edelkamp, 2010], but many games remain out of reach with current hardware.

Finally, a *forward chaining* approach that is based on transformations of the source file in the GaDeLaC compiler [Saffidine and Cazenave, 2011]. A few optimizations were proposed and this approach was shown to lead to better performance than the classic approach in some situations. Still, the GaDeLaC compiler relied on generating high-level code and did not perform any grounding, thereby leaving a margin for improvement.

In this paper, we take the forward chaining approach a step further via two distinct contributions. First, we develop *reference tables* as an efficient implementation of a data structure for ground relations. Second we propose an *empirical hypotheses ordering strategy*.

This ordering strategy based on statistics derived from the domain to be compiled is similar to other empirical library optimization techniques [Keller *et al.*, 2008; Frigo and Johnson, 2005; Whaley *et al.*, 2001].

The motivation for this work is the improvement of previous efforts through the design of a High Speed Forward Chaining Engine that is aligned to the strengths of the modern CPU and relies on the following guidelines for the implementation of reference tables.

- Avoid calls to subroutines, functions and complex math;
- Use fixed length arrays rather than vectors;
- Everything reduced to int32;
- Convert repeated calculations to lists;
- Ensure $O(n)$ worst-case complexity for the Knowledge Base (KB).

2 General Game Playing and Forward Chaining

2.1 Game Description Language

The Game Description Language (GDL) was proposed in 2005 to represent in a unified language the rules of variety of games and an extension covering inmperfect information games, GDL-II, was specified shortly after [Love *et al.*, 2006]. It has since been studied from various perspectives. For instance, the connection between GDL and game theory was investigated by Thielscher [2011] and connections between GDL and multi-agent modal logics were studied by Ruan *et al.* [2009]. Finally, a detailed description of GDL with a forward chaining approach in mind was described by Saffidine and Cazenave [2011].

We assume familiarity of the reader with the GDL and refer to the specification and the GaDeLaC paper for further details [Love *et al.*, 2006; Saffidine and Cazenave, 2011]. In particular, we use techniques presented in the latter for ensuring stratification and detecting permanent facts.

2.2 GDL Rule as Engine

The process of forward chaining for a GDL rule may be considered much like the execution of an SQL statement. There is a resulting dataset, there are source datasets, and there is a set of joining conditions. In this work we coin several terms;

Definition 1. We call *result* a ground instance of a relation arising from the execution of a rule, *precondition* a relation that is ground instantiated in the rule by definition, *input* a ground relation, whose instantiation grounds a variable in the rule, and *condition* a relation that is ground instantiated in the rule because all rule variables are already ground.

Let $(\Leftarrow r\ h_1\ \ldots\ h_n)$ be a rule with head r and n hypotheses h_1 through h_n. A ground instance of r is a *result*. A hypothesis relation containing no variable is a *precondition*. A hypothesis h_i containing a variable x that does not appear in any hypothesis h_j with $j < i$ is an *input*. A hypothesis such that all variables appear in previous hypotheses is a *condition*.

Example 1. The following rule from Breakthrough is used for calculating legal moves. It contains one precondition, two inputs and two conditions.

	Relation	Type
$(\Leftarrow$	(legal white (move $?x_1$ $?y_1$ $?x_2$ $?y_2$))	Result
	(true (control white))	Precondition
	(++ $?y_1$ $?y_2$)	Input: (y_1, y_2)
	(++ $?x_2$ $?x_1$)	Input: (x_2, x_1)
	(true (cellholds $?x_1$ $?y_1$ white))	Condition
	(not (true (cellholds $?x_2$ $?y_2$ white))))	Condition

2.3 Executing a Rule

Each rule must be executed as efficiently as possible, ideally with no wasted calculations. As each instance of a relation is read from the knowledge base it must be processed into the rule, or failed. Prima facie, each rule will need to be executed for every permutation of every instance of every relation. That means enumerating each relation list in the knowledge base.

However, conditions and preconditions might be tested to see if they exist (or not exist), and inputs might be remembered between iterations of the rule. And so, we need the knowledge base to provide a *writing*, an *enumerating*, a *clearing*, and a *testing for existence* operations. The demands on the rule processor are equally tough:

- it performs minimal integer calculations,
- it remembers previous calculations,
- it fails inputs and conditions as soon as possible.

3 Knowledge Base

All relations are stored in a knowledge base, including state relations, facts, and auxilliary relations. They are stored according to the name of the relation, as both a list and a boolean array. In the complexity bounds described below, n refers to the number of relations currently stored.

The lists of relations are stored in production order in an integer array using the relation ID[1], with an integer counter giving the length of the list. The length of the integer array is the size of the maximum superset for the relation.[2] It is

[1] Refer to Definition 4.

[2] Refer to Definition 3.

necessary to store the lists in production order so that rules with circular references will calculate all of the relations. The list of relations provides the following operations.

- Writing to the list in $O(n)$;
- enumerating the list in $O(n)$;
- clearing the list in $O(1)$.

The boolean array is an indexed array where the relation ID is used as the index. Agains, its length is the size of the maximum superset for the relation. It provides the following operations.

- Writing to the array in $O(n)$;
- testing for Exists(ID) in $O(1)$;
- clearing the list in $O(n)$.

As a result, this implementation for the knowledge base provides the following operation.

- Writing to a list in $O(n)$;
- enumerating a list in $O(n)$;
- clearing a list in $O(n)$;
- testing for Exists(ID) in $O(1)$.

4 Relations

In the rest of the paper, G will designate a valid GDL signature and L will be the associated Lexicon[3]. Let Q be a relation in G. We denote by $\dim Q \geq 0$, the arity of Q.

4.1 Rule Inputs and Conditions

The motivation for this work dictates we find the most efficient way to process a rule. Having found the maximum superset for each relation we must look at the order that we process the relations inside a rule. Some relations bring new groundings for rule variables, some relations are ground instantiated, some relations are expressed in the negative.

Here we make the distinction between inputs and conditions. An input is a relations whose instantiation grounds a rule variable. A condition is a relation that is ground instantiated, this includes (not ...) and (distinct ...).

Example 2. Observe that if we change the order of the hypotheses from Example 1, their type changes.

	Relation	Type
($\Leftarrow$	(legal white (move $?x_1$ $?y_1$ $?x_2$ $?y_2$))	Result
	(true (control white))	Precondition
	(++ $?y_1$ $?y_2$)	Input: (y_1, y_2)
	(true (cellholds $?x_1$ $?y_1$ white))	Input: (x_1)
	(++ $?x_2$ $?x_1$)	Input: (x_2)
	(not (true (cellholds $?x_2$ $?y_2$ white))))	Condition

[3]Dictionary of terms converting them to Int32.

4.2 Grounding

Grounding is the process of transforming a GDL description into an equivalent one without any variables. To do so, one must identify for each rule R, a superset of the variable instantiations that could fire R. This involves finding supersets of all reachable terms.

The original specification for GDL allows function constants to be arbitrarily nested. However, this possibility is barely used in practice and the vast majority of games only need a bounded nesting depth. We therefore decided to concentrate on the GDL fragment with bounded nesting depth as it makes finding finite supersets of reachable terms possible.

Definition 2. Let $Q = (q\ x_1 \dots x_{\dim Q})$ a relation.

- We denote the domain (actually, a superset of the domain) of the jth variable argument of Q by Δ_Q^j.
- This set of ground terms $\Delta_Q^j \subseteq L$ is a superset of reachable terms that could occur as j^{th} argument to Q.

We can compute the domains by propagating them recursively from relations in the body of rule to the relation in the head of the rule. We take the intersection of all the domains of each variable, excluding relations expressed negatively, in the body of the rule. This intersection is added to the domain for the variable in the head of the rule. Alternative methods for computing supersets of the domains were proposed by Kissmann and Edelkamp [2010].

Example 3. In TicTacToe the rule for legal moves has been altered to highlight the enumeration of the variable arguments. The domain of the 1st argument of legal is the same as the domain for the 1st argument of control.

($\Leftarrow$ (legal $?0$ (mark $?1$ $?2$))
(true (cell $?1$ $?2$ b))
(true (control $?0$)))

It is now possible to define a superset of the instances of a relation based on the domains for the arguments.

Definition 3. Let Q be a relation in the GDL with name q.

- The set of *instances* of Q, $\mathbb{S}(Q)$, can be obtained as the set of ground instances of Q where each argument ranges over its domain.

  $\mathbb{S}(Q) = \{(q\ a_1 \dots a_{\dim Q}), \forall 1 \leq i \leq \dim Q, a_i \in \Delta_Q^i\}$

- For a relation Q, the size of the set of instances of Q is simply the product of the size of the domains:

  $|\mathbb{S}(Q)| = \prod_{i=1}^{\dim Q} |\Delta_Q^i|$.

4.3 Relation ID

Grounding of relations is achieved by assigning each ground instance of a relation a unique identification (ID). This is an integer that can be calculated once the domain of each argument is known. It is a bijective function so the reverse calculation can be made from ID back to ground instance.

Definition 4. Let Q be a relation in the GDL, where;

- $\Delta_Q^i \subseteq L$ is the ordered set of ground terms forming the domain of the ith argument of the relation Q.

- $I(\Delta_Q^i, l) : D \to \mathbb{N}$ is a function that gives the index of a specific grounding of the ith argument of the relation Q. The index is zero based.
- $\chi\,(q\; a_1 \ldots a_{\dim Q}) : Q \to \mathbb{N}$ is a bijective function that gives the unique identification of a ground instance of Q, such that;

$$\chi\,(q\; a_1 \ldots a_{\dim Q}) = \sum_{i=1}^{\dim Q} \left(I(\Delta_Q^i, l) \times \prod_{j=1}^{i-1} \left|\Delta_Q^i\right| \right)$$

Example 4. In TicTacToe there is a relation cell/3 that expresses the contents of a cell in the grid.

		?x	?y	?p
χ (cell 3 1 o)	*0*	1	**1**	x
$= 2 \times 1 + 0 \times 3 + 1 \times 9$	*1*	2	2	**o**
$= 11$	*2*	**3**	3	

The reverse calculation can be made from an ID of 11 back to (cell 3 1 o).

5 Processing Rules

5.1 Failing Inputs and Conditions

In keeping with the motivation for this work the processing of each of the inputs and conditions should be accompanied with a Pass/Fail test. In order to get the optimal performance we must have an estimate of the probability of an input or condition passing (or failing).

For an input we base the probability of Pass/Fail on the size of the rule argument domains being ground by the relation being input, compared to the size of the maximum superset for the relation being input into the rule. For example; a relation bring all new groundings into a rule will always pass; whereas a relation partially grounded by the rule may not.

Definition 5. Let R be a rule in the GDL, and let Q be a relation in the body of the rule, where;

- $\Delta_R^i \subseteq L$ is the set of ground terms forming the domain of the ith variable argument of the rule R.
- $m : Q \times \mathbb{N} \to \mathbb{N}$ is a mapping from the relation variable argument index to the rule variable argument index.
- Probability $P_{\text{pass}}(Q)$ is given by;

$$P_{\text{pass}}(Q) = \frac{\prod_{j=\text{unground}} \left|\Delta_R^{m(Q,j)}\right|}{|\mathbb{S}\,(Q)|}$$

Example 5. In TicTacToe the relation (cell $?m$ 1 $?x$) has a $P_{\text{pass}}(Q) = 0.33$ as the first input in the rule;

($\Leftarrow$ (row $?m$ $?x$)
(true (cell $?m$ 1 $?x$))
(true (cell $?m$ 2 $?x$))
(true (cell $?m$ 3 $?x$)))

For conditions there is only one instance of the relation that will satisfy, so we base the probability on the likelihood of the instance of the relation existing in the knowledge base. This requires some data to be collected from actual games as to the average number of instances occurring in each list of relations.

Definition 6. Let R be a rule in the GDL, and let Q be a relation in the body of the rule. We denote by $\overline{Q}$ the average number of ground instances in the list of Q at the time of processing the rule R. Let $P_{\text{exists}}\,(Q) \;=\; \frac{\overline{Q}}{|\mathbb{S}(Q)|}$ and $P_{\text{not exists}}\,(Q) \;=\; 1 - \frac{\overline{Q}}{|\mathbb{S}(Q)|}$.

Example 6. In Connect4 the relation does/4 only ever has one ground instance in the knowledge base, but can have 14 variations, so
P_{exists}((does red (drop 1))) $= 0.0714$.

5.2 Processing Time

Each rule is executed by enumerating each of the input lists until every permutation of inputs has been processed. Processing involves grounding each variable argument in the rule according to the ground instance of the input relation and testing the existance (or non existance) of each condition, then the writing of the resulting relation to the knowledge base.

The processing of each permutation of inputs is terminated as quickly as possible. As each ID is read from the knowledge base it is tested for agreement with already ground variables and Passed or Failed. Hence it is possible to determine the overall processing time for a rule.

Theorem 1. *Let KB be a knowledge base for the GDL G, and let R be a rule in the GDL, and let Q be a relation in the body of the rule, where;*

- t_r *is the time to read a RelationID from KB, this includes the management of the list pointers.*
- t_e *is the time to test if a specific RelationID is in the KB.*
- t_w *is the time to write a new RelationID to the KB.*
- n *is the number of inputs.*
- c *is the number of conditions.*
- $P_{pass}(Q)$ *is abbreviated to* $\hat{Q}$
- *The total time* T *to process a rule is given by adding the time taken to process inputs, check conditions and post results;*

$$\begin{aligned} T =& t_r \times \sum_{i=1}^{n} \overline{Q_i} \prod_{j=1}^{i-1} \overline{Q_j}\hat{Q}_j \\ &+ t_e \times \prod_{i=1}^{n} \overline{Q_i}\hat{Q}_i \times \sum_{j=1}^{c} \prod_{k=1}^{j-1} \hat{Q}_k \\ &+ t_w \times \prod_{i=1}^{n} \overline{Q_i}\hat{Q}_i \times \prod_{j=1}^{c} \hat{Q}_j \end{aligned}$$

5.3 Optimisation

In order to optimise the performance of the High Speed Forward Chainer, it is necessary to minimise the Total Processing Time. This is done by minimising T, above.

An examination of the details of T in the context of the propsed knowledge base reveals two things;

- the time t_r is much greater than t_e and t_w as it involves the management of the list pointers, and

- the dominant term in each part of the equation counts the number of input permutations processed.

In other words, minimise the number of input permutations and you minimise the processing time.

Corollary 2. *Let KB be a knowledge base for G, let R be a rule in the GDL, and let Q be a relation in the body of the rule, where;*

- *In the KB; $t_r \gg t_w > t_e$.*
- *The total time T to process the rule R is given by Theorem 1.*
- *Minimum processing time is achieved by minimising;*

$$\sum_{i=1}^{n} \overline{Q_i} \prod_{j=1}^{i-1} \overline{Q_j} \hat{Q}_j$$

- *Specifically, input selection and order matters.*

This result cannot be over stressed; changing the selection of inputs to a rule and their processing order can change the processing time[4]. For this reason it is important to collect some real data from the game. We visit between 100 and 1000 states in random simulations to collect these data.

6 Reference Tables

6.1 Reference Tables for Inputs

For inputs we use the reference table to provide a unique number encoded in the language of the domains for each of the rule variable. The number represent a unique ID for all of the rule variables ground so far, similar to the RelationID in Definition 4. It is in fact an enumeration constructed at the same time as the reference table by counting the input combinations that pass the rules domain criteria. This unique Reference Number is combined with the next inputs RelationID to give a new reference index for a new lookup. It should be noted that some of the new input's arguments are already ground; if they disagree with the ground instance being read from the knowledge base we set the Reference Number to Fail. This is consistent with the motivation for this work.

Definition 7. Let R be a rule in the GDL, and let Q be a relation that is an input to the rule, where;

- χ_i is the RelationID for the next ground instance of the relation Q_i being the ith input to the rule.
- $RefNo_i$ is the value retrieved from the reference table for the ith input to the rule. $RefNo_0 = 0$.
- $RefIndex_i$ is the index (offset) for the reference table for the ith input to the rule.
- $RefTable_i$ is an integer array holding $RefNo_i$.

$$RefIndex_i = RefNo_{i-1} \times |\mathbb{S}(Q_i)| + \chi_i$$
$$RefNo_i = RefTable_i[RefIndex_i]$$
$$(RefNo_i = -1) \leftrightarrow Fail$$

[4]Refer to results in Table 1.

6.2 Reference Tables for Conditions

For conditions we use the reference table to provide the RelationID for performing an Exists() test on the knowledge base. The last reference number obtained from the inputs is used as the reference table index. It should be noted that it is possible to write a rule in such a way as to make some ground instances of conditions where the groundings disagree with the argument domains for the relation; in such cases we set the RelationID to Fail.

Definition 8. Let R be a rule in the GDL, and let Q be a relation that is a condition to the rule, where;

- χ_i is the RelationID for a ground instance of the relation Q_i being the ith condition to the rule.
- $LastRefNo$ is the value retrieved from the reference table for the last input to the rule.
- $RefIndex_i$ is the index (offset) for the reference table for the ith condition to the rule.
- $RefTable_i$ is an integer array holding $RelationID_i$.

$$RefIndex_i = LastRefNo$$
$$\chi_i = RefTable_i[RefIndex_i]$$
$$(\chi_i = -1) \leftrightarrow Fail$$

6.3 Reference Table for the Result

For the result we use the reference table to provide the RelationID for performing a write to the knowledge base. The last reference number obtained from the inputs is used as the reference table index. By now it is impossible to have a failure as the resulting relation argument domains, by definition, agree with the argument domains of the rule.

Definition 9. Let R be a rule in the GDL, and let Q be a relation that is the result to the rule, where;

- χ is the RelationID for a ground instance of the relation Q being the result of the rule.
- $LastRefNo$ is the value retrieved from the reference table for the last input to the rule.
- $RefIndex$ is the index (offset) for the reference table for the result of the rule.
- $RefTable$ is an integer array holding $RelationID$.

$$RefIndex = LastRefNo$$
$$\chi = RefTable_i[RefIndex]$$

6.4 Processing a Rule

In keeping with the motivation for this work, we ground every relation to a 4 byte integer and process the integers as tokens. Initially we considered using an n dimensional array to lookup the result of each permutation of inputs, however this was unworkable and not in keeping with the idea of failing each input as soon as possible. So a Reference Table was devised that allowed fast memory efficient lookup which included failure. The process for executing a rule using the reference tables (Lookup) is shown below. It shows how input combinations are retrieved from the knowledge base (KB) and results are posted to the knowledge base.

Efficient coding can reduce this process to a cycle time of around 15 nanoseconds for a simple rule with two inputs and two conditions, this equates to about 50 clock pulses.

Figure 1: Pseudocode for processing a rule. Each Loop process the relavent number of Inputs or Conditions based on the Rule description, this may be none. Finally the Result is posted to the knowledge base.

```
Check PreCondition
Loop
    Loop
        Get Input RelationID from KB
        Calculate ReferenceIndex
        Lookup ReferenceNumber
        if (ReferenceNumber = Fail)
            GoTo NextInputCombination
    End Loop
    ReferenceIndex = ReferenceNumber
    Loop
        Lookup Condition RelationID
        if (Condition Fails)
            GoTo NextInputCombination
    End Loop
    Lookup Result RelationID
    Post Result to KB
NextInputCombination:
    IncrementInputPointers
    If (LastInput) Exit
End Loop
```

7 Experiments

Experiments were conducted to validate the process for High Speed Forward Chaining. Compilation and runtime statistics were gathered for 19 typical GDL games from previous GGP competitions.[5] Each runtime experiment visited many millions of states and the resulting data were almost identical for each run. Therefore we repeated each runtime experiment only 10 times.

7.1 Experimental Setup

The proof of concept had been conducted in the Windows C++ development environment and the final experiments were conducted in the same environment. The CPU was an Intel core i7 with 4 Hyper-threaded cores operating at 3.4GHz. The memory was 8 GB of DDR 3 RAM operating at 1600MHz. The operating system was housed on a Sata3 128 GB solid state hard disk. There was some concern that the Windows 7 operating system would slow the processing and special care was take to monitor the experiments for signs of slowing; with special attention being paid to Hard Page Faults, indicating that the process was paused while data was read into the L3 cache. Experiments were run as a single thread and no Hard Page Faults were detected.

Timing was measured using the internal clock to an accuracy of 1 millisecond, and reference table and knowledge base sizes were calculated to the nearest byte.

[5]Descriptions can be found on `http://games.ggp.org`.

7.2 Performance of the Proposed Approach

The first set of experiments aims at showing the attractivity of the proposed compilation system. First, we show that compilation can be performed within the typical initialization time at the start of GGP matches. We timed the process through three stages of initialisation; reading and processing the GDL, optimising the rules by running the game manually, grounding the relations and rules into the reference tables.

Then we show that the resulting rule engines are fit for competitive programs. This requires ensuring that the runtime memory overhead is small so as to allow the search engine as much resource as possible, so we give the size of the Reference Table and the size of the Knowledge Base. Finally, we need to demonstrate that the rule engines can process games fast enough. For each game rule in the benchmark, we ran Monte Carlo simulations from the initial state to a final state for 30 seconds. We counted the number of visited states and we display the average number of thousands of states (kilo-states) visited in 1 second. We also show the total number of complete games played out in 30 seconds (in thousands).

We have laid a full set of results in Table 1. It is unwise to make general statement about the results as a whole, so we highlight individual results in the discussion.

Amazons This was the most challenging game. The optimisation time of 21 seconds was for a single state to be fully calculated using a forward chaining process with no groundings or reference tables, at which point the optimiser terminated as it was over the 10 second time limit. However once optimised and ground each new state could be fully calculated in less than five milliseconds. That is a 4 orders of magnitude improvement.

Connect4 The GDL for this game defines a diagonal line. This rule cycles through many 100s of permutations for each round in the game, but only delivers a result once every 200 rounds. Whilst necessary, it is the limiting factor in the speed of processing the rules.

Pancakes6 The astronomical number of 2 million states visited in one seconds in driven by the large number of permanent facts that have been reduced to Pass/Fail entries in the reference tables. This is typical of many of the fast games.

TicTacToe This is the most popular game in the literature and became our benchmark during development; the fact that we can visit 9 hundred thousand states in one seconds is a testament to the success of this work.

7.3 Hypothesis Ordering

We have seen in Example 1 that different relation orderings in a given rule could lead to the relations being assigned a different type. In Section 5.2 we have argued that as a result of the different possible types the rule engine speed depended on a good ordering of the relations. We now provide experimental data to substantiate that claim. Namely, we measured the rule

Table 1: Compilation time and performance of the resulting engine.

Game	Compilation time (sec)			Runtime footprint (kB)		Runtime speed	
	GDL	Optimise	Ground	Reference Tables	Knowledge Base	kStates in 1 sec	kPlayouts in 30 sec
Amazons	14.27	20.96	12.76	333,324	29,795	0.2	0.047
Asteroidserial	1.00	0.07	0.06	336	5	370	118
Beatmania	0.30	0.04	0.01	23	3	366	180
Blocker	0.24	0.08	0.01	4	2	638	1,955
Breakthrough	0.48	0.41	4.47	20,619	103	168	88
Bunk t	0.30	0.03	0.02	8	2	595	1,913
Chomp	0.31	0.02	0.01	130	3	694	3,862
Connect4	0.44	0.64	0.02	62	5	111	142
Doubletictactoe	0.28	0.03	0.01	8	2	636	2,044
Hanoi	0.31	0.06	0.03	164	6	1,375	1,289
Lightsout	0.22	0.02	0.01	6	2	728	1,040
Minichess	0.73	1.31	0.12	1,583	41	75	227
Nim1	0.25	0.17	0.03	481	6	234	895
Pancakes6	0.38	0.01	0.09	2,196	461	2,046	1,507
Peg bugfixed	0.73	5.03	12.54	16,420	28	139	164
Roshambo2	0.24	0.02	0.01	35	1	1,612	5,040
Sheep and Wolf	0.49	3.63	1.96	38,799	85	94	72
Tictactoe	0.21	0.02	0.01	7	1	898	3,118
Tictactoex9	0.50	0.16	0.06	839	6	112	109

engine speed in terms of thousands of processed states per second for the game rules of the benchmark with 3 different rule ordering strategies.

Table 2 displays the results we obtained. In the *Original* column, we kept the ordering present in the source file. In the two other columns, we used Corollary 2 and its dual to define the estimated *Best* and *Worst* orderings.

We can see from the results that the ordering provided by the game rule author could generally be improved upon. Although many authors spend effort improving the GDL files they write, we can not expect that their ordering would be optimal for every approach for processing GDL. Additionally, we can see that the *Best* often provides significant improvement over the two other tested orderings. This gives practical evidence that Corollary 2 provides a good approximation of the optimal ordering.

Amazons and Breakthrough In some games the less than optimal configuration of the rules produced a reference table too large to be stored in RAM. This is shown in the table as Failed. It is worth noting that rules that are two big to be ground can always be processed without reference tables. This is may be many orders of magnitude slower, but it is possible.

Sheep and Wolf The Best configuration did process the rules with fewer permutations of inputs and hence fewer steps, but this required substantially more memory for the reference tables. In this case the movement of memory pages from RAM to the Cache produced a less than optimal performance. This outcome suggests some future work.

Table 2: Impact of the ordering of hypotheses in rules. We consider the original ordering as dictated by the input source file, and the estimated worst and best orderings according to Theorem 1. For each game in our benchmark, we provide the number of thousands of states (kStates) processed per second.

Game	Worst	Original	Best
Amazons	Failed	Failed	0.2
Asteroidserial	351	378	370
Beatmania	342	366	366
Blocker	530	608	638
Breakthrough	Failed	145	168
Bunk t	471	539	595
Chomp	77.5	693	694
Connect4	41.3	78.7	111
Doubletictactoe	457	548	636
Hanoi	188	1,370	1,380
Lightsout	537	551	728
Minichess	62.8	62.7	74.8
Nim1	136	225	234
Pancakes6	1,920	1,960	2,050
Peg bugfixed	3.2	3.6	139
Roshambo2	362	1,480	1,610
Sheep and Wolf	76.9	101	94.3
Tictactoe	771	868	898
Tictactoex9	94.3	109	112

8 Conclusion

Clearly we have improved the speed of the forward chaining approach to the Game Description Language with the approach outlined in this paper.

The closest comparitive work in this field is Kissmann and Edelkamp [2010]. It is difficult to give a single figure that defines the improvement in performance over the results published by Kissmann and Edelkamp [2010] as each game varied by a different, and in some cases dramatic, amount. If we exclude the highest and lowest differences we can say that we are generally 3 times faster then their approach.

However, the two approaches are not incompatible and future work could investigate how to best take advantage of both. Indeed, Kissmann and Edelkamp [2010] obtain much smaller sets of instances than we do. In turn, this would lead to smaller Reference Tables and Knowledge Base which could improve the overall performance significantly in larger domains.

References

Vítor Santos Costa, Luís Damas, Rogério Reis, and Rúben Azevedo. *YAP Prolog user's manual*. Universidade do Porto, 2006.

Evan Cox, Eric Schkufza, Ryan Madsen, and Michael Genesereth. Factoring general games using propositional automata. In *IJCAI Workshop on General Intelligence in Game-Playing Agents (GIGA)*, pages 13–20, 2009.

Hilmar Finnsson and Yngvi Björnsson. Simulation-based approach to general game playing. In Dieter Fox and Carla P. Gomes, editors, *Twenty-Third AAAI Conference on Artificial Intelligence*, pages 259–264. AAAI Press, July 2008.

Matteo Frigo and Steven G Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

Gabriele Keller, Hugh Chaffey-Millar, Manuel MT Chakravarty, Don Stewart, and Christopher Barner-Kowollik. Specialising simulator generators for high-performance Monte-Carlo methods. In *Practical Aspects of Declarative Languages*, pages 116–132. Springer, 2008.

Peter Kissmann and Stefan Edelkamp. Instantiating general games using prolog or dependency graphs. In *KI 2010: Advances in Artificial Intelligence*, pages 255–262. Springer, 2010.

Jeffrey Long, Nathan R. Sturtevant, Michael Buro, and Timothy Furtak. Understanding the success of perfect information Monte Carlo sampling in game tree search. In *24th AAAI Conference on Artificial Intelligence (AAAI)*, pages 134–140, 2010.

Nathaniel C. Love, Timothy L. Hinrichs, and Michael R. Genesereth. General Game Playing: Game Description Language specification. Technical report, LG-2006-01, Stanford Logic Group, 2006.

Jean Méhat and Tristan Cazenave. Combining UCT and nested Monte-Carlo search for single-player general game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):271–277, 2010.

Maximilian Möller, Marius Schneider, Martin Wegner, and Torsten Schaub. Centurio, a General Game Player: Parallel, java- and ASP-based. *KI - Künstliche Intelligenz*, 25:17–24, 2011.

Ji Ruan, Wiebe Van Der Hoek, and Michael Wooldridge. Verification of games in the game description language. *Journal of Logic and Computation*, 19(6):1127–1156, 2009.

Abdallah Saffidine and Tristan Cazenave. A forward chaining based game description language compiler. In *IJCAI Workshop on General Intelligence in Game-Playing Agents (GIGA)*, pages 69–75, Barcelona, Spain, July 2011.

Stephan Schiffel and Michael Thielscher. Fluxplayer: A successful general game player. In *22nd AAAI Conference on Artificial Intelligence (AAAI)*, pages 1191–1196, 2007.

Michael Schofield, Tim Cerexhe, and Michael Thielscher. Hyperplay: A solution to general game playing with imperfect information. In *26th AAAI Conference on Artificial Intelligence (AAAI)*, 2012.

Michael Thielscher. Answer set programming for single-player games in general game playing. In *Logic Programming*, pages 327–341. Springer, 2009.

Michael Thielscher. The general game playing description language is universal. In *22nd International Joint Conference on Artificial Intelligence*, IJCAI, pages 1107–1112, July 2011.

Kevin Waugh. Faster state manipulation in general games using generated code. In *IJCAI-09 Workshop on General Game Playing (GIGA'09)*, 2009.

R Clint Whaley, Antoine Petitet, and Jack J Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1):3–35, 2001.

Lifting HyperPlay for General Game Playing to Incomplete-Information Models

Michael Schofield and **Timothy Cerexhe** and **Michael Thielscher**
School of Computer Science and Engineering
The University of New South Wales
{mschofield,timothyc,mit}@cse.unsw.edu.au

Abstract

General Game Playing is the design of AI systems able to understand the rules of new games and to use such descriptions to play those games effectively. Games with imperfect information have recently been added as a new challenge for existing general game-playing systems. The only published solution to this challenge, HyperPlay, maintains a collection of complete information models. In doing so it grounds all of the unknown information thereby valuing all information gathering moves at zero—a well-known criticism of such sampling-based or particle filter systems.

We have extended HyperPlay to better reason about its knowledge. This escalates reasoning from complete-information models to incomplete-information models, and correctly values information gathering moves. In this paper we describe the new HyperPlay-II technique, show how it was adapted for use with a Monte Carlo decision making process, and give experimental results demonstrating its superiority over its predecessor.

1 Introduction

General Game Playing (GGP) is concerned with the design of AI systems able to understand the rules of new games and to use such descriptions to play those games effectively. GGP with perfect information has made advances, thanks largely to the standardisation of the Game Description Language [Love *et al.*, 2006] and its widespread adoption, particularly in the AAAI GGP Competition [Genesereth *et al.*, 2005]. Successful players typically employ either automatically generated evaluation functions [Clune, 2007; Schiffel and Thielscher, 2007] or some form of Monte Carlo technique such as the modern UCT [Björnsson and Finnsson, 2009]. The relative ease of Monte Carlo parallelisation has also rewarded distributed and cluster-based hardware in this domain [Méhat and Cazenave, 2011].

Games with *imperfect information* have recently been added as a new challenge for existing general game-playing systems [Thielscher, 2010]. However little progress has been made on these types of games beyond a specification of what their rules should look like [Quenault and Cazenave, 2007; Thielscher, 2011]. This was confirmed at the recent Australasian Joint Conference on Artificial Intelligence where the game descriptions were all designed to test a specific property, ie. games that are hard to play for a clear reason. One such game was Number Guessing, where a player must guess a random number after asking the fewest number of binary questions it can. Other games tested concealing private information, opponent modelling, managing large probabilistic domains, and decomposition.[1]

Beyond GGP, Frank and Basin [2001] have investigated imperfect-information games with a focus on Bridge, presenting a 'game-general' tree search algorithm that exploits a number of imperfect-information heuristics. This may effectively complement existing work applying theorem-proving techniques [Schiffel and Thielscher, 2009]. The Alberta Computer Poker Research Group has developed systems at the forefront of computer Poker players [Billings *et al.*, 2006]—a challenging domain combining incomplete and misleading information, opponent modelling, and a large state space. While not explicitly interested in GGP, they do describe several techniques that could generalise to this field, including *miximix*, fast opponent modelling, and Nash equilibrium solutions over an abstracted state space. Meanwhile, our work is motivated by set sampling [Richards and Amir, 2009] and by particle system techniques [Silver and Veness, 2010]. Similar special-case applications of sampling to reduce imperfect- to perfect-information can be found in [Ginsberg, 2011; Kupferschmid and Helmert, 2007].

Despite these advances in related fields, the only published work we are aware of that attempts to model and play *general* imperfect-information games is HyperPlay [Schofield *et al.*, 2012], which presents a partial solution to this challenge by maintaining a collection of models of the true game as a foundation for reasoning and move selection. HyperPlay's strength is in sacrificing a complete representation of all possible worlds in the information set of an imperfect-information game. By restricting its attention to a subset of possible worlds, it is able to correctly reason about a large class of imperfect-information games without becoming intractable. However this strength becomes a weakness

[1] 1st Australian Open 2012, see `https://wiki.cse.unsw.edu.au/ai2012/GGP`

```
1 role(agent).
2 role(random).
3
4 colour(red).
5 colour(blue).
6
7 init(round(0)).
8
9 legal(random,arm(C)) :- colour(C), true(round(0)).
10 legal(random,noop)   :- not true(round(0)).
11 legal(agent,noop)    :- true(round(0)).
12 legal(agent,ask)     :- true(round(1)).
13 legal(agent,wait)    :- true(round(1)).
14 legal(agent,cut(C))  :- true(round(2)), colour(C).
15
16 sees(agent,C) :- does(agent,ask), true(armed(C)).
```

```
17 next(round(1))    :- true(round(0)).
18 next(round(2))    :- true(round(1)).
19 next(round(3))    :- true(round(2)).
20 next(armed(C))    :- does(random,arm(C)).
21 next(armed(C))    :- true(armed(C)).
22 next(score(90))   :- does(agent,ask).
23 next(score(100))  :- does(agent,wait).
24 next(score(S))    :- true(score(S)), not explodes.
25 next(score(0))    :- explodes.
26
27 explodes :- true(armed(C)), does(agent,cut(C)).
28
29 terminal :- true(round(3)).
30 goal(agent, S) :- terminal, true(score(S)).
31 goal(agent, 0) :- not terminal.
32 goal(random,0).
```

Figure 1: GDL-II description of the Exploding Bomb game.

when reasoning about its own knowledge is critical to successful play, since the sampling technique 'solves' imperfect-information by, sometimes erroneously, escalating hypothesis to fact. Thus, while all samples (known as hypergames) may agree on which move to do next, the individual reasons may be contradictory.

For example in Number Guessing all hypergames 'already know' the secret number, so they all agree to guess. In the next round, however, it is revealed that there was only a 'superficial agreement' between the hypergames. This is a criticism shared with the broader category of particle filter systems [Silver and Veness, 2010].

In this paper we propose HyperPlay-II, an extended version of the original technique able to play a much larger class of imperfect-information games by reasoning on incomplete-information models. This new technique values information correctly according to the expected cost/benefit, performs information-gathering moves when appropriate, is protective of information that should remain discreet, and requires no additional resources over its predecessor.

The remainder of the paper is organised as follows. In the next section, we recapitulate syntax and operational semantics of the Game Description Language GDL-II, which provides the formal basis for General Game Playing [Genesereth *et al.*, 2005; Thielscher, 2010]. We subsequently review the previous technique and describe the general HyperPlay-II technique. Thereafter, we report on experiments between the new and the old algorithms. We conclude with a short discussion.

2 Background: Game Description Language

The science of General Game Playing requires a formal language that allows an arbitrary game to be specified by a complete set of rules. The declarative Game Description Language (GDL) serves this purpose [Genesereth *et al.*, 2005]. It uses a logic programming-like syntax and is characterised by the special keywords listed in Table 1.

Originally designed for games with complete information [Genesereth *et al.*, 2005], GDL has recently been extended to GDL-II (for: *GDL with incomplete/imperfect information*) by the last two keywords (`sees`, `random`) to describe arbitrary (finite) games with randomised moves and imperfect information [Thielscher, 2010].

Keyword	Meaning
`role(R)`	`R` is a player
`init(F)`	`F` holds in the initial position
`true(F)`	`F` holds in the current position
`legal(R,M)`	`R` can do `M` in the current position
`does(R,M)`	player `R` does move `M`
`next(F)`	`F` holds in the next position
`terminal`	the current position is terminal
`goal(R,V)`	`R` gets payoff `V`
`sees(R,P)`	`R` perceives `P` in the next position
`random`	the random player (aka. Nature)

Table 1: GDL-II keywords

Since the number guessing game would take too much space, we use a much simpler game called Exploding Bomb as our running example. As with number guessing, this game puts an emphasis on the value of knowledge and information-gathering moves.

Example 1. The GDL-II rules in Fig. 1 formalise a simple game that commences with the random player choosing a red or blue wire. This arms a bomb accordingly. Next, the agent may choose whether or not to ask which wire was used; asking carries a cost of 10% to the final score. Finally, the agent must then cut one of the wires to either disarm—or detonate—the bomb.

The intuition behind the GDL rules is as follows.[2] Line 1 introduces the players' names. Line 7 defines the single feature that holds in the initial game state. The possible moves are specified by the rules for `legal`: in the first round, the `random` player arms the bomb (line 9); then the agent gets to choose whether to ask or wait (lines 12–13), followed by cutting a wire of his choice (line 14). The agent's only percept is the true answer if he decides to enquire about the right wire (line 16). The remaining rules specify the state update (rules for `next`); the conditions for the game to end (rule for `terminal`); and the payoff (rule for `goal`).

[2] A word on the syntax: We use infix notation for GDL-II rules as we find this more readable than the usual prefix notation. Variables are denoted by uppercase letters.

GDL-II comes with some syntactic restrictions—for details we must refer to [Love *et al.*, 2006; Thielscher, 2010] due to lack of space—that ensure that every valid game description has a unique interpretation as a state transition system as follows. The **players** in a game are determined by the derivable instances of `role(R)`. The **initial state** is the set of derivable instances of `init(F)`. For any state S, the **legal moves** of a player `R` are determined by the instances of `legal(R,M)` that follow from the game rules *augmented by an encoding of the facts in* S using the keyword `true`. Since game play is synchronous in the Game Description Language,[3] states are updated by *joint* moves (containing one move by each player). The **next position** after joint move $\vec{m}$ is taken in state S is determined by the instances of `next(F)` that follow from the game rules *augmented by an encoding of* $\vec{m}$ *and* S using the keywords `does` and `true`, respectively. The **percepts** (aka. information) a player `R` gets as a result of joint move $\vec{m}$ being taken in state S is likewise determined by the derivable instances of `sees(R,P)` after encoding $\vec{m}$ and S using `true` and `does`. Finally, the rules for `terminal` and `goal` determine whether a given state is **terminal** and what the players' **goal values** are in this case.

On this basis, game play in GDL-II follows this protocol:

1. Starting with the initial state, which is completely known to all players, in each state each player selects one of their legal moves. By definition `random` must choose a legal move with uniform probability.
2. The next state is obtained by (synchronously) applying the joint move to the current state. Each role receives their individual percepts resulting from this update.
3. This continues until a terminal state is reached, and then the goal relation determines the result for all players.

3 Lifting HyperPlay

The motivation for this work is the weakness formally identified in Definition 3 below, in that the original technique seeks to maximise the expected outcome from a sample across the information set for the current round. By sampling the information set, all unknown information is ground and the maximisation process will select against any information gathering move that has a cost. This is HyperPlay's Achilles' heel.

For example HyperPlay is unable to play the Number Guessing game because it incorrectly extrapolates individual samples of the information set (hypergames) to fact rather than treating multiple samples in concert—one hypergame 'knows' the number is three, another 'knows' it is seven. Each hypergame then chooses to guess rather than incur the cost of asking a question. These hypergames are never forced to justify their decisions.

To remedy this weakness, we present the HyperPlay-II technique, a refinement of the original technique to include an Imperfect Information Simulation (IIS) in the decision making process. This allows reasoning directly with imperfect information, exploring the consequences of every action given its context, and using these outcomes to make a decision. This allows the new technique to encompass larger (ie. non-singleton) subsets of the information set. The result is that HyperPlay-II places the correct value (based on cost/benefit) on knowledge and will choose information gathering moves when appropriate.

The Original Technique HyperPlay is described as a solution to the challenge of imperfect information games play, by maintaining a bag of models of the true game as a foundation for reasoning and move selection. It provides existing game players with a bolt-on solution to convert from perfect information games to imperfect-information games [Schofield *et al.*, 2012]. Effectively it maintains these models (hypergames) by updating them after every move so that they agree with all of the percepts received by the player and ground everything that remains unknown. Move selection is done by maximising the expected reward across all of the hypergames using a weighting factor based on the probability that the hypergame is the true game.

Decision Making Process As with the original approach, the new technique requires a bag of models of the information set (hypergames), representing a weighted sample. Very similar to a weighted particle filter in that all unknowns are grounded, and each model is updated based on move choices (actions) and percepts (signals).

Unlike the original approach, the expected payoff values reflect the rewards from the optimal strategy being executed in an imperfect-information game. This addresses the principle failing of the original technique, which used the reward values from the optimal strategy being executed in a perfect-information game.

We now formally define the HyperPlay-II decision making process, adapting notation from [Osborne, 2004].

Definition 1. Let G be an imperfect information game as described in the Game Description Language (GDL)

- N is a set of players in G.
- V is a set of nodes on the game tree of G.
- T is a set of terminal nodes.
- $D = V \backslash T$ is a set of decision nodes.
- $\mathcal{H}$ is the information partition of D, and $H \in \mathcal{H}$ is the information set for the current round in the game.
- $A_n(H)$ is a set of moves available to player $n \in N$ in the current information set H. Sometimes referred to as equivalence classes.

Definition 2. Let G be an imperfect information game, with a game tree, information set H and equivalence classes $A_n(H)$

- $a_n \in A_n(H)$ is a move available to player n.
- $\alpha = \langle a_1 ... a_n \rangle$ is a move vector (tuple of moves with one move for each role) for the current round.
- $do : \alpha, D \to 2^V$ is the successor function defined by the game G.
- $d_{i+1} = do(\alpha, d_i)$ where $d_i \in D$ is how the game progresses from one decision node to the next.

[3] Synchronous means that all players move simultaneously. Turn-taking games are modelled by allowing players only one legal move without effect (such as `noop`) if it is not their turn.

- $h_{i+1} = do(\alpha, h_i)$ where $h_i \in H$ move the game from a node in the current information set to a node in the next information set.

We first show the move selection process for the original technique so as to make a comparison with the new technique.

Definition 3. Let G be an imperfect information game, with a game tree, information set H, equivalence classes $A_n(H)$ and a successor function do. The expected value of information set is given by

$$E(H) = max_{a \in A_n(H)}[avg_{h \in H}[E(do(a,h))]]$$

where

- $h \in H$ is a hypergame;
- $do(a,h)$ is an action with perfect information, and the expected value is provided by the embedded perfect information player.

By comparison the new technique calculates the expected value of the information set recursively until all of the paths have terminated, then maximises that value.

Definition 4. Let G be an imperfect information game, with a game tree, information set H, equivalence classes $A_n(H)$ and a successor function do. The expected value of information set is given by

$$E(H) = max_{a \in A_n(H)}[E(do(a,H))]$$

recursively,

- terminating with $E(\tau) = \frac{1}{|\tau|}\sum_{v \in \tau} Reward(v,n)$ where $\tau \subseteq T$;
- $do(a,H)$ is an action with **imperfect information**, and the expected value cannot be provided by the embedded perfect information player.

Both techniques find the move in the current round that maximises the expected value of the information set. The old technique collapses the information set into a sample of hypergames, by grounding the unknown values. The other calculates the true expected value of the information set. For this reason, the new technique always places the correct value on information gathering moves, whereas the old technique revalues information gathering moves at zero when it grounds all of the unknowns. This is the key distinction between the techniques.

Example We now present a worked example of how expected values are calculated in the Exploding Bomb game (cf. Fig. 1). Recall that play commences with the random player choosing a red or blue wire. Next, the agent chooses whether or not to ask which wire was used; asking carries a cost of 10% to the final score. Finally, the agent must then cut one of the wires to either disarm—or detonate—the bomb.

Fig. 2 shows the expected outcomes $E(a_j)$ in round 1 for each action in the game tree. For brevity we refer to actions (cut/arm) blue as b, (cut/arm) red as r, ask as a, wait as w, and to states as $do(\text{cut blue}, do(\text{ask}, do(\text{arm red}, v_0))) = rab$. Note the change in order.

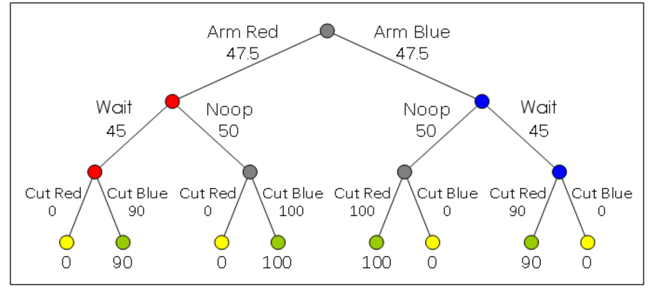


Figure 2: The Exploding Bomb game tree

The reward sequences are $R(rwb) = R(bwr) = 100$, $R(rab) = R(bar) = 90$, and $R(bwb) = R(rwr) = R(bab) = R(rar) = 0$. The information set in round 1 is $H = \{b, r\}$.

The old player considers both hypergames equally likely and the policy is uniform, hence $E(ask|H)$ is the average of $R(bab) \times 0.5 + R(bar) \times 0.5$ and $R(rab) \times 0.5 + R(rar) \times 0.5$ is 45. The policy for the new player recognises the correct move, so π^*_{HPII} gives probability 1.0 to hypergames rab and bar, and probability 0.0 to the other two states. Thus $E(ask|H)$ is now the average of $R(bab) \times 0 + R(bar) \times 1.0$ and $R(rab) \times 1.0 + R(rar) \times 0$ is 90. Similarly for $wait$, the old player favours its chances at guessing correctly in preference to paying the penalty for asking, so $E(wait|H) = 50$ and $a^*_{\text{HP}} = \textit{wait}$. The new player arrives at the same expected value since the information set cannot be divided, and so chooses the more promising *ask* action: $a^*_{\text{HPII}} = \textit{ask}$.

agent does	HyperPlay	HyperPlay-II
Round 1:		
ask	45	**90**
wait	**50**	50
Round 2:		
cut armed	**50**	**90**
cut unarmed	**50**	0

Table 2: Expected outcomes for Exploding Bomb

This result can be compared to the experimental results given in Table 3.

4 Experiments

A series of experiments were designed to test the capabilities of the new technique using the high-throughput computer facilities at the School of Computer Science and Engineering. We used the games played at the recent Australasian Joint Conference on Artificial Intelligence as inspiration for the experiments that would validate our claim that the new technique correctly values moves that seek information. The conference organisers had specially designed games that would challenge the state of the art of GDL-II players so as to encourage research and development in this field.

4.1 Player Resources

As with the original technique, the resources were varied to demonstrate the change in performance as resources were increased. However the new technique carries four resource parameters, compared to two parameters in the original. The original technique contained hypergames each with an embedded GDL player (we used Monte Carlo simulations). Now we have hypergames each with an embedded Imperfect Information Simulation (IIS), each with a GDL-II player for each role, each containing hypergames, each with an embedded GDL player (again we used Monte Carlo simulations). Hence a resource number of eight means the player has eight models of the true game, each running eight IIS, where each player in the IIS is modelled by an original HyperPlayer, each with eight models of IIS game and running eight Monte Carlo simulations for each move choice. For a two-player game with an average of ten moves there would be $8 \times 8 \times 2 \times 10 \times 8 = 81920$ Monte Carlo simulations being run for each move choice in the true game.

This exponential growth in the number of Monte Carlo simulations presented a challenge since a fully-resourced original player typically required resources of ~32 hypergames, each running ~32 simulations for each possible move in each round. A similar level of resourcing would make the new technique intractable. However experiments showed otherwise—the data below shows that the new player can function optimally[4] with the same number of total resources as the original player. That is, where the original player might need a resource index of 32 ($= 32^2$ simulations), the new player only requires a resource index of $\sqrt{32}$. Which translates to exactly the same CPU time for both players. This result emphasises the contribution made by the new technique in both efficacy and efficiency.

4.2 Games and Simulations

We have implemented a version of the new technique (the 'HyperPlayer-II') to validate our claim that it places the correct value on knowledge and will seek information appropriately. As with the previous research, we modelled the game server and both players in a single thread so it could be parallelised across many CPUs. Each datum below is supported by one hundred games; a 99% confidence interval is calculated and shown as an error bar.

4.3 Equal Resources

Some experiments were conducted with two-player games, pitting the original player against the new player using equal resources. For a resource index of four, the new player would have four hypergames, each running an IIS with four hypergames, while the old player would have 16 hypergames.

Some games show a player resource index of zero. This represents random decision making and serves to provide a basis for improvement.

[4] A player is 'optimal' when increasing its resources will not lead to better play. We refer to this as 'adequate resourcing'.

5 Results

The results of each experiment are given below, along with a commentary on their significance.

5.1 Exploding Bomb

The old player never asks the question in this game since it carries a penalty that it thinks it can avoid (due to superficial agreement of its hypergames). This contrasts with the new player which correctly identifies that asking the question gives the best expected outcome. Table 3 shows the experimental results of calculations made by each technique when choosing an action. Remember that the action with the highest expected score is chosen for each round (shown in bold). Note the HyperPlayer chooses randomly in round 2.

agent does	HyperPlay	HyperPlay-II
Round 1:		
ask	45.04 ± 0.09	$\mathbf{90.00 \pm 0.00}$
wait	$\mathbf{49.98 \pm 0.10}$	49.91 ± 0.64
Round 2:		
cut armed	$\mathbf{50.60 \pm 1.19}$	$\mathbf{90.00 \pm 0.00}$
cut unarmed	$\mathbf{49.40 \pm 1.19}$	0.00 ± 0.00

Table 3: Experimental score calculations during the Exploding Bomb decision making process

5.2 Spy vs. Spy

A simple variant of the Bomb Maker game is to change which direction the information flows. In the Bomb Maker game the disarming agent could ask for the answer, in this version the arming agent—who chooses which wire arms the bomb—also decides whether to tell the other player which wire to cut. Withholding this information carries a penalty of 20%. This tests the value both players place on giving away information.

arming agent does	HyperPlay	HyperPlay-II
arm blue and tell	$\mathbf{60.00 \pm 0.15}$	20.00 ± 0.00
arm red and tell	$\mathbf{60.04 \pm 0.14}$	20.00 ± 0.00
arm blue and hide	39.98 ± 0.16	$\mathbf{40.36 \pm 1.22}$
arm red and hide	39.99 ± 0.14	$\mathbf{39.45 \pm 1.33}$

Table 4: Expected score calculations for the arming agent in round one of the Spy vs. Spy decision making process

Table 4 shows experimental results in the form of calculated expected outcomes (the chosen action is bold). When the original HyperPlayer is the arming agent it always tells to avoid the penalty. HyperPlayer-II recognises that hiding this information yields a better expected outcome.

5.3 Number Guessing

The original player reasoning with perfect information always announces it is 'ready to guess', but then guesses randomly resulting in a 6.25% chance of guessing correctly. The new player only guesses the number when all hypergames agree on the result.

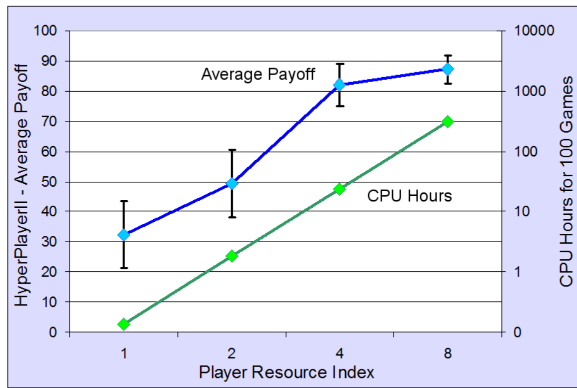


Figure 3: The Number Guessing Results for HyperPlay-II. The dark line represents the average payoff (with error bars for 99% confidence interval). The light line shows increased CPU resources required for increased player resources (note the log-log scale).

Binary search is a perfect player here, able to guess after four questions. The new player used in the experiments approached this score with a resource index of eight, as seen in Fig. 3.

5.4 Banker and Thief

This game tests a player's ability to keep secrets, ie. to value withholding information from their opponent. The banker is given ten $10 notes to deposit arbitrarily into two banks, and is (randomly) assigned a 'target' bank. The banker scores all the money left in that bank at the end of the game. The thief can steal all the money from one bank and is scored by the amount stolen *provided* it came from the target bank—which they do not know. Guessing the wrong bank yields a score of zero for the thief. The challenge for the banker is not to reveal the target bank by over-depositing.

Fig. 4 shows that the original technique adopts a greedy policy and places all of the money in the target bank, only to have it stolen. The new technique, adequately resourced, will deposit 40% of the money in the target bank, relying on a greedy thief to attempt to steal the 60% in the other bank.

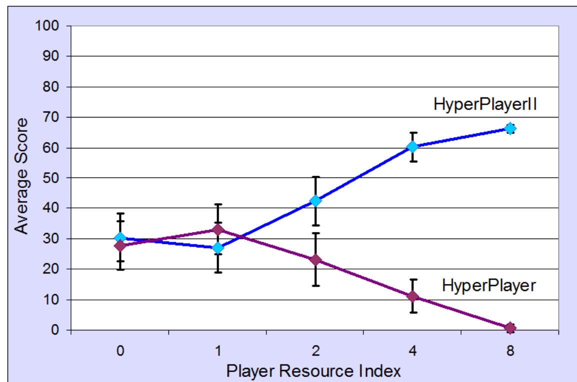


Figure 4: The Banker and Thief results

The new player reaches the optimal performance with a resource index of eight. At this level it models the behaviour of both roles correctly and avoids giving away the location of the target bank.

5.5 Battleships In Fog

This turn-taking, zero-sum game was designed to test a player's ability to gather information and to be aware of information collected by its opponent. Two battleships occupy the same grid but cannot see each other—even when they occupy the same square. A player can fire a missile to any square on the grid, move to an adjacent square, or scan for their opponent. If they scan they will get the exact location, and their opponent will know that they have been scanned.

The original player sees no value in scanning as all of the hypergames already 'know' where the opponent is. It doesn't value moving after being scanned as it knows its opponent always knows where it is. Its only strategy is to randomly fire missiles giving it a 6.25% chance of a hit on a 4x4 board. The new player, adequately resourced, will scan for the opponent and fire a missile. When the original player was challenged by the new player the difference was clear.

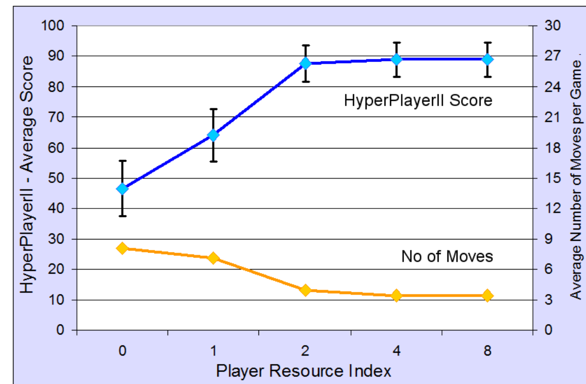


Figure 5: The Battleships In Fog results for HyperPlay-II versus HyperPlay

Note that a resource index of four is all that is required for new player to have complete dominance over old in this turn-taking game: HyperPlay has a 9.4% chance of winning with a random shot (12.5% if it goes first, half that if it plays second). This is reflected in the experimental results (Fig. 5). Note also that HyperPlayer-II requires only three rounds to finish the game: scan, noop, fire.

6 Conclusion

The experimental results show the value HyperPlay-II places on knowledge, and how it correctly values information-gathering moves by it and its opponents. It is able to collect information when appropriate, withhold information from its opponents, and keep its goals secret. The use of the Imperfect Information Simulations is an efficacious and efficient tool for reasoning with imperfect information. A HyperPlayer-II was easily able to outperform an equally resourced HyperPlayer in all of the experiments.

We intend to explore additional features of the HyperPlay-II technique as they pertain to general artificial intelligence through the development of the HyperWorlds technique. This will be a more generalised solution, capable of dealing with imperfect information in real-world scenarios.

We also intend to implement the HyperPlayer-II for the General Game Playing arena as a benchmark competitor for other researchers to challenge in GDL-II games.

Acknowledgements. We thank the anonymous reviewers of an earlier version for their constructive comments. This research was supported under Australian Research Council's (ARC) *Discovery Projects* funding scheme (project DP 120102023). Michael Thielscher is the recipient of an ARC Future Fellowship (project FT 0991348). He is also affiliated with the University of Western Sydney.

References

[Billings *et al.*, 2006] Darse Billings, Aaron Davidson, Terence Schauenberg, Neil Burch, Michael Bowling, Robert Holte, Jonathan Schaeffer, and Duane Szafron. Game-tree search with adaptation in stochastic imperfect-information games. In *Proc. Computers and Games*, pages 21–34, 2006.

[Björnsson and Finnsson, 2009] Y. Björnsson and H. Finnsson. CadiaPlayer: A simulation-based general game player. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(1):4–15, March 2009.

[Clune, 2007] Jim Clune. Heuristic evaluation functions for general game playing. In *Proc. AAAI*, pages 1134–1139, Vancouver, July 2007.

[Frank and Basin, 2001] Ian Frank and David Basin. A theoretical and empirical investigation of search in imperfect information games. *Theoretical Computer Science*, 252(1-2):217–256, 2001.

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

[Ginsberg, 2011] Matthew Ginsberg. GIB: Imperfect information in a computationally challenging game. *CoRR*, 2011.

[Kupferschmid and Helmert, 2007] Sebastian Kupferschmid and Malte Helmert. A Skat player based on Monte-Carlo simulation. In *Proc. Computers and Games*, pages 135–147, 2007.

[Love *et al.*, 2006] Nathaniel Love, Timothy Hinrichs, David Haley, Eric Schkufza, and Michael Genesereth. General game playing: Game description language specification. Technical Report LG–2006–01, Stanford Logic Group, 2006.

[Méhat and Cazenave, 2011] Jean Méhat and Tristan Cazenave. A parallel general game player. *KI Journal*, 25(1):43–47, 2011.

[Osborne, 2004] Martin J. Osborne. *An introduction to game theory*, volume 3. Oxford University Press, New York, NY, 2004.

[Quenault and Cazenave, 2007] M. Quenault and T. Cazenave. Extended general gaming model. In *Computer Games Workshop*, pages 195–204, 2007.

[Richards and Amir, 2009] Mark Richards and Eyal Amir. Information set sampling for general imperfect information positional games. In *Proc. IJCAI-09 Workshop on GGP (GIGA'09)*, pages 59–66, 2009.

[Schiffel and Thielscher, 2007] Stephan Schiffel and Michael Thielscher. Fluxplayer: A successful general game player. In *Proc. AAAI*, pages 1191–1196, 2007.

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

[Schofield *et al.*, 2012] Michael Schofield, Timothy Cerexhe, and Michael Thielscher. HyperPlay: A solution to general game playing with imperfect information. In *Proc. AAAI*, pages 1606–1612, Toronto, July 2012. AAAI Press.

[Silver and Veness, 2010] David Silver and Joel Veness. Monte-Carlo planning in large POMDPs. In *Proc. NIPS*, pages 2164–2172, 2010.

[Thielscher, 2010] Michael Thielscher. A general game description language for incomplete information games. In *Proc. AAAI*, pages 994–999, Atlanta, July 2010.

[Thielscher, 2011] Michael Thielscher. The general game playing description language is universal. In *Proc. IJCAI*, pages 1107–1112, Barcelona, July 2011.

Model Checking for Reasoning About Incomplete Information Games *

Xiaowei Huang, Ji Ruan, Michael Thielscher
University of New South Wales, Australia
{xiaoweih, jiruan, mit}@cse.unsw.edu.au

Abstract

GDL-II is a logic-based knowledge representation formalism used in general game playing to describe the rules of arbitrary games, in particular those with incomplete information. In this paper, we show how *model checking* can be used to automatically verify that games specified in GDL-II satisfy desirable temporal and knowledge conditions. We present a systematic translation of GDL-II to a model checking language, prove the translation to be correct, and demonstrate the feasibility of applying model checking tools for GDL-II games by four case studies.

1 Introduction

The general game description language GDL, which has been established as input language for general game-playing systems [Genesereth *et al.*, 2005], has recently been extended to GDL-II to incorporate games with nondeterministic actions and where players have incomplete/imperfect information [Thielscher, 2010]. However, not all GDL-II descriptions correspond to games, let alone meaningful, non-trivial games. Genesereth *et al.* [2005] list a few properties that are necessary for well-formed GDL games, including guaranteed termination and the requirement that all players have at least one legal move in non-terminal states. The introduction of incomplete information raises new questions, e.g., can players *always know* their legal moves in non-terminal states or *know* their goal values in terminal states?

Temporal logics have been applied to the verification of computer programs, and more broadly computer systems [Manna and Pnueli, 1992; Clarke and Emerson, 1981]. The programs are in certain states at each time instance, and the correctness of the programs can be expressed as temporal specifications. An example is the formula "$AG\,\neg deadlock$" meaning *the program can never enter a deadlock state*. Epistemic logics, on the other hand, are formalisms for reasoning about knowledge and beliefs. Their application in verification was originally motivated by the need to reason about communication protocols. One is typically interested in what knowledge different parties to a protocol have before, during and after a run (i.e., an execution sequence) of the protocol. Fagin *et al.* [1995] give a comprehensive study on epistemic logic for multi-agent interactions.

Ruan and Thielscher [2011] have shown that the situation at any stage of a game in GDL-II can be characterized by a multi-agent epistemic (i.e., S5-) model. Yet, this result only provides a static characterization of what players know (and don't know) at a certain stage.

Our paper extends this recent analysis with a temporal dimension, and also provides a practical method for verifying temporal and epistemic properties using a model checker MCK [Gammie and van der Meyden, 2004]. We present a systematic translation from GDL-II into equivalent specifications in the model specification language of MCK. Verifying a property φ for a game description G is then equivalent to checking whether φ holds for the translation $\mathsf{trs}(G)$. The latter can be automatically checked in MCK.

The paper is organized as follows. Section 2 introduces GDL-II and MCK. Section 3 presents the translation along with possible optimizations and a proof of its correctness. Experimental results for four case studies are given in Section 4. The paper concludes with a discussion of related work and directions for further research.

2 Background

2.1 Game Description Language GDL-II

A complete game description consists of the names of (one or more) players, a specification of the initial position, the legal moves and how they affect the position and the players' knowledge thereof, and the terminating and winning criteria. The emphasis of game description languages is on *high-level, declarative game rules* that are easy to understand and maintain. Background knowledge is not required—a set of rules is all a player needs to know to be able to play a hitherto unknown game. Meanwhile, GDL and its successor GDL-II have a precise semantics and are fully machine-processable.

The GDL-II rules in Fig. 1 formalize a simple but famous game called *Monty Hall* where a car prize is hidden behind one of three doors and where a candidate is given two chances to pick a door. Highlighted are the pre-defined *keywords* of GDL-II. The intuition behind the rules is as follows. Line 1 introduces the players' names (the game host is modelled

*This paper is an extended version of a paper presented at the ECAI'12 Computer and Games Workshop

```
1  role(candidate). role(random).
2  init(closed(1)). init(closed(2)).
3  init(closed(3)). init(step(1)).
4
5  legal(random,hide_car(?d))  <=
6     true(step(1)), true(closed(?d)).
7  legal(random,open_door(?d)) <=
8     true(step(2)),     true(closed(?d)),
9     not true(car(?d)), not true(chosen(?d)).
10 legal(random,noop)          <= true(step(3)).
11 legal(candidate,choose(?d)) <=
12    true(step(1)), true(closed(?d)).
13 legal(candidate,noop)       <= true(step(2)).
14 legal(candidate,noop)       <= true(step(3)).
15 legal(candidate,switch)     <= true(step(3)).
16
17 next(car(?d))     <= does(random,hide_car(?d)).
18 next(car(?d))     <= true(car(?d)).
19 next(closed(?d)) <= true(closed(?d)),
20                     not does(random,open_door(?d)).
21 next(chosen(?d)) <= does(candidate,choose(?d)).
22 next(chosen(?d)) <= true(chosen(?d)),
23                     not does(candidate,switch).
24 next(chosen(?d)) <= does(candidate,switch),
25    true(closed(?d)), not true(chosen(?d)).
26 next(step(2)) <= true(step(1)).
27 next(step(3)) <= true(step(2)).
28 next(step(4)) <= true(step(3)).
29 sees(candidate,?d) <= does(random,open_door(?d)).
30 sees(candidate,?d) <= true(step(3)), true(car(?d)).
31
32 terminal <= true(step(4)).
33 goal(candidate,100) <= true(chosen(?d)),true(car(?d)).
34 goal(candidate,  0) <= true(chosen(?d)),
35                        not true(car(?d)).
```

Figure 1: G_{MH} - a GDL-II description of the Monty Hall game adapted from [Thielscher, 2011].

by pre-defined role called random). Lines 2–3 define the four features that comprise the initial game state. The possible moves are specified by the rules for legal: in step 1, the random player must decide where to place the car (lines 5–6) and, simultaneously, the candidate chooses a door (lines 11–12); in step 2, random opens a door that is not the one that holds the car nor the chosen one (lines 7–9); finally, the candidate can either stick to their earlier choice (noop) or switch to the other, yet unopened door (line 14 and 15, respectively). The candidate's only percept throughout the game is to see the door opened by the host (line 29) and where the car is after step 3 (line 30). The remaining rules specify the state update (rules for next), the conditions for the game to end (rule for terminal), and the payoff for the player depending on whether they got the door right in the end (rules for goal).

GDL-II is suitable for describing synchronous n-player games with randomness and imperfect information. Valid game descriptions must satisfy certain syntactic restrictions, which ensure that all deduction problems "$\vdash$" needed in Definition 1 are finite and decidable; see [Love *et al.*, 2006] for details. In the following, we assume the reader to be familiar with basic notions and notations of logic programming, as can be found in e.g. [Lloyd, 1987].

A state transition system can be obtained from a valid GDL-II game description by using the notion of the *stable models* of logic programs with negation [Gelfond and Lifschitz, 1988]. The syntactic restrictions in GDL-II ensure that all logic programs we consider have a *unique* and *finite* stable model [Love *et al.*, 2006; Thielscher, 2010]. Hence, the state transition system for GDL-II has a finite set of players, finite states, and finitely many legal moves in each state. By $G \vdash p$ we denote that ground atom p is contained in the unique stable model, denoted as $\text{SM}(G)$, for a stratified set of clauses G. In the following definition of the game semantics for GDL-II, *states* are identified with the set of ground atoms that are true in them.

Definition 1. *[Thielscher, 2010] Let G be a valid GDL-II description. The state transition system $(R, s_0, \tau, l, u, \mathcal{I}, \Omega)$ of G is given by*

- *roles $R = \{i \mid \texttt{role}(i) \in \text{SM}(G)\}$;*
- *initial position $s_0 = \text{SM}(G \cup \{\texttt{true}(f) \mid \texttt{init}(f) \in \text{SM}(G)\})$;*
- *terminal positions $\tau = \{s \mid \texttt{terminal} \in s\}$;*
- *legal moves $l = \{(i, a, s) \mid \texttt{legal}(i, a) \in s\}$;*
- *state update function $u(M, s) = \text{SM}(G \cup \{\texttt{true}(f) \mid \texttt{next}(f) \in \text{SM}(G \cup s \cup M)\})$, for all* joint *legal moves M (i.e., where each role in R takes one legal move) and states s;*
- *information relation $\mathcal{I} = \{(i, M, s, p) \mid \texttt{sees}(i, p) \in \text{SM}(G \cup s \cup M)\}$;*
- *goal relation $\Omega = \{(i, n, s) \mid \texttt{goal}(i, n) \in s\}$.*

Note that a state s contains all ground atoms that are true in the state, which includes the "fluent atoms" $\texttt{true}(f)$ in, respectively, $\{\texttt{true}(f) \mid \texttt{init}(f) \in \text{SM}(G)\}$ (for the initial state) and $\{\texttt{true}(f) \mid \texttt{next}(f) \in \text{SM}(G \cup s \cup M)\}$ (for the successor state of s and M), and all other atoms that can be derived from G *and* these fluent atoms.

Different runs of a game can be described by *developments*, which are sequences of states and moves by each player up to a certain round. A player *cannot distinguish* two developments if the player has made the same moves and perceptions in both of them.

Definition 2. *[Thielscher, 2010] Let $(R, s_0, \tau, l, u, \mathcal{I}, \Omega)$ be the state transition system of a GDL-II description G, then a* development *δ is a finite sequence*

$$\langle s_0, M_1, s_1, \ldots, s_{d-1}, M_d, s_d \rangle$$

such that for all $k \in \{1, \ldots, d\}$ ($d \geq 0$), M_k is a joint move and $s_k = u(M_k, s_{k-1})$.

A terminal development is a development such that the last state is a terminal state, i.e., $s_d \in \tau$. The length *of a development δ, denoted as len(δ), is the number of states in δ. By $M(i)$ we denote agent i's move in the joint move M. Let $\delta|_k$ be the prefix of δ up to length $k \leq$ len(δ).*

A player $i \in R \setminus \{\texttt{random}\}$ cannot distinguish *two developments $\delta = \langle s_0, M_1, s_1, \ldots \rangle$ and $\delta' = \langle s_0, M'_1, s'_1 \ldots \rangle$ (written as $\delta \sim_i \delta'$) iff len(δ) = len(δ') and for any $1 \leq k \leq$ len(δ) $- 1$:*

- *$M_k(i) = M'_k(i)$, and*
- *$\{p \mid (i, M_k, s_{k-1}, p) \in \mathcal{I}\} = \{p \mid (i, M'_k, s'_{k-1}, p) \in \mathcal{I}\}$.*

2.2 Model Checker MCK

In this paper, we will use MCK, for "Model Checking Knowledge", which is a model checker for temporal and knowledge specifications [Gammie and van der Meyden, 2004]. The overall setup of MCK supposes a number of agents acting in an environment. This is modelled by an *interpreted system*, formally defined below, where agents perform actions according to protocols. Actions and the environment may be only partially observable at each instant in time. In MCK, different approaches to the temporal and epistemic interaction and development are implemented. Knowledge may be based on current observations only, on current observations and clock value, or on the history of all observations and clock value. The last corresponds to *synchronous perfect recall* and is used in this paper. In the temporal dimension, the specification formulas may describe the evolution of the system along a single computation, i.e., using linear time temporal logic; or they may describe the branching structure of all possible computations, i.e., using branching time or computation tree logic. We give the basic syntax of Computation Tree Logic of Knowledge (CTL*K$_n$).

Definition 3. *The language of CTL***K*$_n$ *(with respect to a set of atomic propositions* Φ*), is given by the following grammar:*

$$\varphi ::= p \mid \neg\varphi \mid \varphi \vee \psi \mid A\varphi \mid X\varphi \mid \varphi\,\mathcal{U}\,\psi \mid K_i\varphi.$$

The other logic constants and connectives $\top, \bot, \vee, \rightarrow$ are defined as usual. In addition, $F\varphi$ (read: finally, φ) is defined as $\top\,\mathcal{U}\,\varphi$, and $G\varphi$ (read: globally, φ) is defined as $\neg F\neg\varphi$.

The semantics of the logic can be given using *interpreted systems* [Fagin *et al.*, 1995]. Let S be a set, which we call the set of environment states, and Φ be the set of atomic propositions. A *run* over environment states S is a function $r : \mathbf{N} \rightarrow S \times L_1 \times \ldots \times L_n$, where each L_i is called the set of *local states of agent* i. These local states are used to concretely represent the information on the basis of which agent i computes its knowledge. Given run r, agent i, and time m, we write $r_i(m)$ for the $(i+1)$-th component (in L_i) of $r(m)$, and $r_e(m)$ for the first component (in S). An *interpreted system* over environment states S is a tuple $\mathcal{IS} = (\mathcal{R}, \pi)$, where $\mathcal{R}$ is a set of runs over environment states S, and $\pi : \mathcal{R} \times \mathbf{N} \rightarrow \mathcal{P}(\Phi)$ is an interpretation function. A *point* of $\mathcal{IS}$ is a pair (r, m) where $r \in \mathcal{R}$ and $m \in \mathbf{N}$.

Definition 4. *Let* $\mathcal{IS}$ *be an interpreted system,* (r, m) *be a point of* $\mathcal{IS}$*, and* φ *be a CTL***K*$_n$ *formula.* Semantic entailment $\models$ *is defined inductively as follows:*

- $\mathcal{IS}, (r, m) \models p$ *iff* $p \in \pi(r, m)$*;*
- *the propositional connectives* $\neg, \wedge$ *are defined as usual;*
- $\mathcal{IS}, (r, m) \models A\varphi$ *iff for all runs* $r' \in \mathcal{R}$ *with* $r'(k) = r(k)$ *for all* $k \in [0..m]$*, we have* $\mathcal{IS}, (r', m) \models \varphi$*;*
- $\mathcal{IS}, (r, m) \models X\varphi$ *iff* $\mathcal{IS}, (r, m+1) \models \varphi$*;*
- $\mathcal{IS}, (r, m) \models \varphi\,\mathcal{U}\,\psi$ *iff* $\exists m' \geq m$ *s. t.* $\mathcal{IS}, (r, m') \models \psi$ *and* $\mathcal{IS}, (r, k) \models \varphi$ *for all* $k \in [m..m')$*;*
- $\mathcal{IS}, (r, m) \models K_i\varphi$ *iff for all points* (r', m') *with* $r_i(m) = r'_i(m')$*, we have* $\mathcal{IS}, (r', m') \models \varphi$.

We now describe the syntax and semantics of the input language of MCK, following [van der Meyden *et al.*, 2012].

Syntax of MCK Input Language

An MCK description consists of an environment and one or more agents. An environment model represents how states of the environment are affected by the actions of the agents. A protocol describes how an agent selects an action under certain environment.

Formally, an *environment model* is a tuple $\mathcal{M}_e = (Agt, Acts, Var_e, Init_e, Prog_e)$ where Agt is a set of agents, $Acts$ is a set of actions available to the agents, Var_e is a set of environment variables, $Init_e$ is an initial condition, in the form of a boolean formula over Var_e, and $Prog_e$ is a standard program for the environment e to be defined below.

Let $ActVar(\mathcal{M}_e) = \{i.a \mid i \in Agt,\ a \in Acts\}$ be a set of *action variables* generated for each model $\mathcal{M}_e$. An atomic statement in $Prog_e$ is of the form $x := expr$, where $x \in Var_e$ and $expr$ is an expression over $Var_e \cup ActVar(\mathcal{M}_e)$.

A *protocol for agent* i in environment $\mathcal{M}_e$ is a tuple $Prot_i = (PVar_i, OVar_i, Acts_i, Prog_i)$, where $PVar_i \subseteq Var_e$ is a set of *parameter variables*, $OVar_i \subseteq PVar_i$ is a set of *observable variables*, $Acts_i \subseteq Acts$, and $Prog_i$ is a standard program. An atomic statement in $Prog_i$ is either of the form $x := expr$, or of the form $\ll a \gg$ with $a \in Acts_i$.

A *standard program* over a set Var of variables and a set A of atomic statements is either the terminated program ϵ or a sequence P of the form $stat_1\,;\,\ldots\,;\,stat_m$, where the $stat_k$ are simple statements and ';' denotes sequential composition.

Each simple statement $stat_k$ is an atomic statement in A; or a nondeterministic branching statement of the form

$$\text{if } g_1 \rightarrow a_1 \,[]\, g_2 \rightarrow a_2 \,[]\ldots[]\, g_m \rightarrow a_m \text{ fi;}$$

or a nondeterministic iteration statement of the form

$$\text{do } g_1 \rightarrow a_1 \,[]\, g_2 \rightarrow a_2 \,[]\ldots[]\, g_m \rightarrow a_m \text{ od;}$$

where each a_k is an atomic statement in A and each g_k is a boolean expressions over Var called *guard*.

Each atomic statement a_k can be executed only if its corresponding guard g_k holds in the current state. If several guards hold simultaneously, one of the corresponding actions is selected nondeterministically. The last guard g_m can be $otherwise$, which is shorthand for $\neg g_1 \wedge \cdots \wedge \neg g_{m-1}$. An *if*-statement executes once but a *do*-statement can be repeatedly executed.

Semantics of MCK Input Language

Based on a set of agents running particular protocols in the context of a given environment, we can define an interpreted system as follows.

Definition 5. *A* system model $\mathcal{S}$ *is a pair* $(\mathcal{M}_e, Prot)$ *with* $\mathcal{M}_e = (Agt, Acts, Var_e, Init_e, Prog_e)$ *and* $Prot$ *a joint protocol, i.e., with* $Prot_i = (PVar_i, OVar_i, Acts_i, Prog_i)$ *for all* $i \in Agt$.

Let a state with respect to $\mathcal{S}$ *be an assignment* s *over the set of variables* Var_e*. A* transition model *over* $\mathcal{S}$ *is* $\mathcal{M}(\mathcal{S}) = (S, I, \{O_i\}_{i \in Agt}, \rightarrow, V)$*, where* S *is the set of states of* $\mathcal{S}$*;* I *is the set of initial states* s *such that* $s \models Init_e$*;* $O_i(s) = s \restriction OVar_i$ *is the partial assignment given on the observable variables of agent* i*,* $\rightarrow$ *is a transition relation on* $S \times S$*;*[1] *and*

[1]More precisely, $s \rightarrow s'$ if s' is obtained by executing the parallel program $Prog_e \,||_{i \in Agt}\, Prog_i$ on s; see [van der Meyden *et al.*, 2012] for details.

a valuation function V is given by: for any boolean variable x, $x \in V(s)$ iff $s(x) = true$. [2]

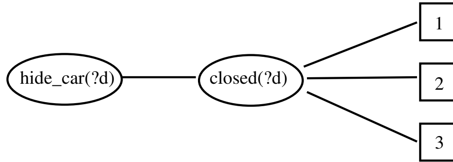
An infinite sequence of states $s_0 s_1 ...$ is an initialized computation *of $\mathcal{M}(\mathcal{S})$ if $s_0 \in I$, $s_k \in S$ and $s_k \to s_{k+1}$ for all $k \geq 0$. An* **interpreted system over** *$\mathcal{S}$ is $\mathcal{IS}(\mathcal{S}) = (\mathcal{R}, \pi)$, where $\mathcal{R}$ is the set of runs such that each run r corresponds to an initialized computation $s_0 s_1 ...$ with $r_e(m) = s_m$, and $r_i(m) = O_i(s_0)O_i(s_1)\ldots O_i(s_m)$; and $\pi(r, m) = V(s_m)$.*

3 Translation from GDL-II to MCK

Our main contribution in this paper is a systematic translation from a GDL-II description G into an MCK description $\mathtt{trs}(G)$. The translation is provably correct in that the game model derived from G using the semantics of GDL-II satisfies the exact same formulas as the model that is derived from $\mathtt{trs}(G)$ using the semantics of MCK. This will be formally proved later in this section.

We use the GDL-II description of the Monty Hall game from Fig. 1, denoted as G_{MH}, to illustrate the whole process. The translation trs can be divided into the following steps.

Preprocessing The first step is to obtain a variable-free (i.e., ground) version of the game description G. We can compute the domains, or rather supersets of the domains, of all predicates and functions of G by generating a domain dependency graph from the rules of the game description, following [Schiffel and Thielscher, 2007]. The nodes of the graph are the arguments of functions and predicates in game description, and there is an edge between two nodes whenever there is a variable in a rule of the game description that occurs in both arguments. Connected components in the graph share a (super-)domain. E.g., lines 2–6 in G_{MH} give us the domain graph as follows, from which it can be seen that the arguments of both closed() and hide_car() range over the domain $\{1, 2, 3\}$.



Once we have computed the domains, we instantiate all the variables in G. This gives us all ground atoms, e.g., true(closed(1)), legal(random, hide_car(1)), etc. Our following translation operates on the variable-free version of G, which for convenience we still refer to as G.

Deriving Environment Variables This step derives all the environment variables Var_e. Let $\mathtt{AT}$ be the set of ground atoms in G. Define the following subsets of $\mathtt{AT}$ according to the keywords: $\mathtt{AT_t} = \{\mathtt{h} \in \mathtt{AT} \mid \mathtt{h} = \mathtt{true(p)}\}$, $\mathtt{AT_n} = \{\mathtt{h} \in \mathtt{AT} \mid \mathtt{h} = \mathtt{next(p)}\}$, $\mathtt{AT_d} = \{\mathtt{h} \in \mathtt{AT} \mid \mathtt{h} = \mathtt{does(i,a)}\}$, $\mathtt{AT_i} = \{\mathtt{h} \in \mathtt{AT} \mid \mathtt{h} = \mathtt{init(p)}\}$ and $\mathtt{AT_s} = \{\mathtt{h} \in \mathtt{AT} \mid \mathtt{h} = \mathtt{sees(r,p)}\}$.

Define t as follows:

- $t(\mathtt{true(p)}) = p_{old}$ and $t(\mathtt{next(p)}) = p$;
- $t(\mathtt{init(p)}) = p$, and $t(\mathtt{does(i,a)}) = did_i$;
- $t(\mathtt{p}) = p$ for all $\mathtt{p} \in \mathtt{AT} \setminus (\mathtt{AT_t} \cup \mathtt{AT_n} \cup \mathtt{AT_d} \cup \mathtt{AT_i})$.

Note that the ground atoms with keywords legal, terminal, goal are all in $\mathtt{AT} \setminus (\mathtt{AT_t} \cup \mathtt{AT_n} \cup \mathtt{AT_d} \cup \mathtt{AT_i})$. The set of environment variable Var_e is then $\{t(\mathtt{p}) \mid \mathtt{p} \in \mathtt{AT}\}$. For convenience, we denote $t(A)$ as $\{t(x) \mid x \in A\}$.

The type of each variable $did_i \in t(\mathtt{AT_d})$ is the set of legal moves of agent i plus two additional moves, INIT and STOP, that do not appear in G, i.e., $\{\mathtt{a} \mid \mathtt{legal(i,a)} \in \mathtt{AT}\} \cup \{\mathtt{INIT}, \mathtt{STOP}\}$. The type of variables in $Var_e \setminus t(\mathtt{AT_d})$ is Bool.

Initial Condition This step specifies the environment initial condition $Init_e$, which essentially is an assignment over Var_e. By using the semantics of G and $\mathtt{AT_i}$, we first compute the initial state s_0 (see Definition 1). Then for any $\mathtt{p} \in \mathtt{AT_i}$, we add boolean expression "$t(\mathtt{p}) == true$" to $Init_e$ as a conjunct; and for all $did_i \in t(\mathtt{AT_d})$, we add "$did_i == \mathtt{INIT}$". For the rest, add "$t(\mathtt{p}) == true$" if $\mathtt{p} \in s_0$, and "$t(\mathtt{p}) == false$" if $\mathtt{p} \notin s_0$.

Agent Protocols This step specifies the agents and their protocols. The names of the agents are read off the role() facts. Let $Prot_i = (PVar_i, OVar_i, Acts_i, Prog_i)$ be the protocol of agent i, such that $PVar_i = Var_e$, $OVar_i = \{sees_i_p \mid sees_i_p \in t(\mathtt{AT_s})\} \cup \{did_i\}$ includes all the variables representing i's percept and i's move, and $Acts_i = \{\mathtt{a} \mid \mathtt{legal(i,a)} \in G\}$ includes all the legal moves of agent i. Note that $Acts_i$ does not include the two special moves in the protocol. The last component $Prog_i$ is a standard program of the following format:

```
begin do neg terminal ->
   if legal_i_a1 -> <<a1>> []
      legal_i_a2 -> <<a2>> []
       ...
   fi
od end
```

This program intuitively means that if the current state is not terminal, then a legal move is selected non-determinstically by i. The statements between do $\cdots$ od are executed repeatedly.

State Transition This step specifies the environment program $Prog_e$. Each environment variable is updated corresponding to the rules in G. The main task is *to translate these rules into MCK statements in a correct order.* In GDL-II, the order of the rules does not matter as the stable models semantics always gives a unique model, but MCK uses the imperative programming style in which the order of the statements does matter; e.g., executing "$x := 0; x := 1;$" results in a different state than "$x := 1; x := 0;$". To take care of the order, we separate the program $Prog_e$ into three parts.

[2] For simplicity, we assume x is a boolean; this can be easily extended to enumerated type variables. Suppose x is a variable with type $\{e_1, \ldots, e_m\}$, we can use m booleans $x.e_1, \ldots, x.e_m$ such that $x.e_k \in V(s)$ iff $s(x) = e_k$.

The first part is to update the variables in $t(\mathtt{AT_d})$ using the following template (for agent i):

```
if  i.a1      -> did_i := a1  []
    i.a2      -> did_i := a2  []
    ...
    otherwise -> did_i := STOP
fi;
```

The second part of $Prog_e$ updates the variables in $t(\mathtt{AT_t})$ and $t(\mathtt{AT_n} \cup \mathtt{AT_s})$. For all $p_{old} \in t(\mathtt{AT_t})$, an atomic statement of the form $p_{old} := p$ is added to ensure that the value of p is remembered before it is updated. For any atom $h \in t(\mathtt{AT_n} \cup \mathtt{AT_s})$, suppose $h = t(\mathtt{h})$ and $Rules(\mathtt{h})$ is the set of rules in G with head $\mathtt{h}$:

$$\begin{array}{llll} r_1: & \mathtt{h} & \Leftarrow & b_{11}, \cdots, b_{1j} \\ \ldots & \ldots & \ldots & \ldots \\ r_k: & \mathtt{h} & \Leftarrow & b_{k1}, \cdots, b_{kj} \end{array}$$

where b_{xy} is a literal over $\mathtt{AT}$.

Define a translation tt as follows:

- $tt(\mathtt{does(i,a)}) = did_i == a$;
- $tt(\mathtt{not\ x}) = neg\ tt(\mathtt{x})$; and other cases are same as t.

The translation of $Rules(\mathtt{h})$ has the following form:

$$h := (tt(b_{11}) \wedge \cdots \wedge tt(b_{1j})) \vee \cdots \vee (tt(b_{k1}) \wedge \cdots \wedge tt(b_{kj}))$$

This simplifies to $h := true$ if one of the bodies is empty. Essentially, this is a form of the standard Clark [1978] Completion, which captures the idea that $\mathtt{h}$ will be false in the next state unless there is a rule to make it true. The statements with $t(\mathtt{AT_t})$ should be given before those with $t(\mathtt{AT_n} \cup \mathtt{AT_s})$.

The third part deals the variables in $t(\mathtt{AT} \setminus (\mathtt{AT_t} \cup \mathtt{AT_n} \cup \mathtt{AT_s} \cup \mathtt{AT_d} \cup \mathtt{AT_i}))$. Pick such an atom h and take $Rules(\mathtt{h})$. The literals in the body of these rules are translated differently from the last case, as $\mathtt{h}$ refers to the current instead of the next state. Define a new translation tt' as follows:

- $tt'(\mathtt{true(p)}) = p$ and all other cases are identical to tt.

The translation of $Rules(\mathtt{h})$ is similar to the above by replacing tt by tt'. The statements in the third part are ordered according to the dependency graph. If $\mathtt{h}'$ depends on $\mathtt{h}$, then the statement of $tt'(\mathtt{h})$ must appear before that of $tt'(\mathtt{h}')$. The fact that GDL rules are stratified ensures that a desirable order can always be found.

3.1 Optimizations

The above translation can be further optimized to make the model checking more efficient by reducing the number of variables.

(1) Using definitions. The variables in $t(\mathtt{AT} \setminus (\mathtt{AT_t} \cup \mathtt{AT_n} \cup \mathtt{AT_d} \cup \mathtt{AT_i}))$ (refer to the third part of state transition step) can be represented as definitions to save memory space for variables. The assignment statement $h := expr$ is swapped with definition $define\ h = expr$. MCK replaces h using the boolean expression $expr$ during its preprocessing stage, so h does not occupy memory during the main stage.

(2) Removing static atoms. We distinguish three special kinds of atoms in GDL-II: those (a) appearing in the rules with empty bodies, (b) never appearing in the heads of rules, (c) only appearing in the rules with (a) and (b). Under the GDL-II semantics, atoms in (a) are always true, those in (b) are always false, and those in (c) do not change their value during gameplay. Therefore we can replace them universally with their truth values. E.g., consider the following rules:

```
succ(1,2). succ(2,3).
next(step(2)) <= true(step(1)), succ(1,2).
next(step(3)) <= true(step(2)), succ(2,3).
```

Both $\mathtt{succ(1,2)}, \mathtt{succ(2,3)}$ are always true, so we replace them using their truth values. Then we can further simplify this by removing the "*true*" conjuncts universally (and by removing the rules with a "*false*" conjunct in the body):

```
next(step(2)) <= true(step(1)).
next(step(3)) <= true(step(2)).
```

(3) Converting booleans to typed variables. The atoms in $\mathtt{AT} \setminus \mathtt{AT_d}$ are translated to booleans in our non-optimized translation. There often are sets of booleans B such that at each state exactly one of them is true. We can then convert the booleans in B into one single variable v_B with the type $\{b_1, \ldots, b_{|B|}\}$, where $|B|$ is the size of B. This results in a logarithmic space reduction on B: $2^{|B|}$ is reduced to $|B|$. Reusing the example just discussed, we can create a variable v_{step} with type $\{1, 2, 3\}$.

3.2 Translation Soundness

The above completes the translation from G to $\mathsf{trs}(G)$. As our main theoretical result, we show that our translation is correct as follows: first the game model derived from a GDL-II description G is isomorphic to the interpreted system that is derived from its translation $\mathsf{trs}(G)$, then a CTL*K$_n$ formula has an equivalent interpretation over these two models (i.e., having the same truth value).

We first extend the concept of finite developments in Definition 2 to infinite ones.

Definition 6 (Infinite Developments and GDL-II Game Models)**.** *Let $\langle R, s_0, t, l, u, \mathcal{I}, g\rangle$ be the state transition system of G, and $\delta = \langle s_0, M_1, s_1, \ldots, M_d, s_d\rangle$ be a finite terminal development of G, then an* infinite extension *of δ is an infinite sequence $\langle s_0, M_1, s_1, \ldots, M_d, s_d, M_{d+1}, s_{d+1}, \ldots\rangle$ such that M_{d+k} is the joint move where all players take a special move* STOP *and $s_{d+k} = s_d$ for all $k \geq 1$.*

Given a GDL-II description G, the game model $\mathsf{GM}(G)$ *is a tuple $(D, \{\sim_i |i \in Agt\})$, where D is the set of infinite developments δ such that either δ is an infinite development without terminal states, or δ is an infinite extension of a finite terminal development; and $\sim_i$ is agent i's indistinguishability relation defined on the finite prefixes of $\delta|_k$ as in Definition 2.*

For a given δ, let $\delta(k)$ denote the k-th state s_k; $\delta(k)^M$ the k-th joint move M_k; and (δ, k) the pair (M_k, s_k).

Definition 7 (Isomorphism)**.** *Let* $\mathsf{GM} = (D, \{\sim_i |i \in Agt\})$ *be a game model and $\mathcal{IS} = (\mathcal{R}, \pi)$ an interpreted system.* GM *is* isomorphic *to $\mathcal{IS}$ if there is a bijection* w *between the ground atoms of* GM *and the atomic propositions of $\mathcal{IS}$, and a bijection* z *between D and $\mathcal{R}$ satisfying the following:* $\mathsf{z}(\delta) = r$ *iff for any ground atom* p: $\mathtt{p} \in \delta(k)$ *iff* $\mathsf{w}(\mathtt{p}) \in \pi(r, k)$*, and* $\mathtt{does}(i, a) \in \delta(k)^M$ *iff $did_i == a$ is true in (r, k).*

Intuitively, z associates a point (δ, k) in a development to a point (r, k) in a run such that they coincide in the interpretation of basic and move variables. The following proposition is the first step in showing the correctness of our translation.

Proposition 1. *Given a GDL-II description G, let* trs *be the translation from GDL-II to MCK, then the game model* $\mathsf{GM}(G)$ *is isomorphic to the interpreted system* $\mathcal{IS}(\mathsf{trs}(G))$.

Proof. (Sketch) Bijection w is obtained from the step of deriving variables in trs. Then define a map z from an arbitrary infinite development δ to a run r, and show z is a bijection by induction on the move-state pairs of δ. The base case is the initial state, and in the inductive step, we first use the fact that the moves M_{k+1} chosen in the current state s_k should all be $legal$, and then derive that the corresponding legal variables should be true in (r, k), then use the joint protocol of agents to get the execution path with the corresponding moves in M_{k+1}. Then the environment program guarantees that all other variables are updated accordingly. For the technical details of the proof we must refer to [Ruan and Thielscher, 2012]. □

Let w be a bijection from the set of ground atoms of G to the set of atomic propositions of CTL*K$_n$ and w^{-1} be its inverse. The semantics of CTL*K$_n$ over GDL-II Game Models can be given as relation $\mathsf{GM}(G), (\delta, m) \models \varphi$ in analogy to the semantics of CTL*K$_n$ over interpreted systems; e.g., $\mathsf{GM}(G), (\delta, m) \models p$ iff $\mathsf{w}^{-1}(p) \in \delta(m)$, and $\mathsf{GM}(G), (\delta, m) \models K_i\varphi$ iff for all states (δ', m') of $\mathsf{GM}(G)$ that satisfy $\delta|_m \sim_j \delta'|_{m'}$ we have $\mathsf{GM}(G), (\delta', m') \models \varphi$.

The following proposition then shows that checking φ against a game model of G is equivalent to checking φ against the interpreted system of $\mathsf{trs}(G)$.

Proposition 2. *Given a GDL-II description G, let* trs *be the translation from GDL-II to MCK;* φ *a CTL***K*$_n$ *formula over the set of atomic propositions in* $\mathsf{trs}(G)$*; and* w, z *the bijections from the isomorphism between* $\mathsf{GM}(G)$ *and* $\mathcal{IS}(\mathsf{trs}(G))$ *then:*

$$\mathsf{GM}(G), (\delta, m) \models \varphi \textit{ iff } \mathcal{IS}(\mathsf{trs}(G)), (\mathsf{z}(\delta), m) \models \varphi.$$

This follows from Proposition 1 by an induction on the structure of φ and completes the proof of our main result.

Our optimization techniques do not affect the isomorphism. So we can follow a similar argument as Proposition 1 and 2 to show that the optimized translation is also sound.

4 Experimental Results

We present experimental results on four GDL-II games from the repository at *www.general-game-playing.de*: Monty Hall (MH), Krieg-TicTacToe (KTTT), Transit, and Meier. MCK (v1.0.0) runs on Intel Core i5-2500 CPU 3.3 GHz and 8GB RAM with GNU Linux OS 2.6.32.

Temporal and epistemic specifications The temporal logic formulas can be used to specify the *objective* aspects of a game. The following three properties represent the basic requirements from [Genesereth *et al.*, 2005]. (Let $Legal_i$ and $Goal_i$ be the set of legal moves and goals of i respectively.)

$$AF\ terminal \tag{1}$$

$$AG(\neg terminal \rightarrow \bigwedge_{i \in Agt} \bigvee_{p \in Legal_i} p) \tag{2}$$

$$\bigwedge_{i \in Agt} \neg AG \neg goal_i_100 \tag{3}$$

Property (1) says that the game always terminates. Property (2) expresses *playability*: at every non-terminal state, each player has a legal move. Property (3) expresses *fairness*: every player has a chance to win, i.e., eventually achieving the maximal goal value 100. These properties apply both to GDL and GDL-II games. The next three properties concern the *subjective* views of the players under incomplete-information situations, hence are specific to GDL-II games.

$$\bigwedge_{i \in Agt} G(terminal \rightarrow K_i terminal) \tag{4}$$

$$\bigwedge_{i \in Agt} G\ (\neg terminal \rightarrow \bigwedge_{p \in Legal_i} (K_i p \vee K_i \neg p)) \tag{5}$$

$$\bigwedge_{i \in Agt} G\ (terminal \rightarrow \bigwedge_{p \in Goal_i} (K_i p \vee K_i \neg p)) \tag{6}$$

Property (4) says that once the game has terminated, all players know this. Property (5) says that any player always knows its legal moves in non-terminal states; and property (6) says that in a terminal state, all players know their outcome.

Table 1 shows the runtimes on five translations. The first four translations use all three optimization techniques on the four games. The last translation Meier′ is partially optimized with the third technique applied only for the variables in $t(\mathtt{AT_s})$. As a consequence, Meier′ uses 126 booleans that in the fully optimized Meier are represented by 4 enumerated type variables of a size equivalent to about 22 booleans, i.e., the state space of Meier is only $(1/2)^{104}$ of the state space of Meier′. The time is measured in seconds and "NA" means MCK does not return a result after 10 hours. A comparison of the two translations of Meier shows that our optimization can be very effective. Somehow surprisingly, the result shows that the game Meier is not well-formed as it does not satisfy property (1). The last three properties were also checked in [Haufe and Thielscher, 2012], but we were able to get results on Transit and Meier that are beyond the expressivity of their approach. Note that although we only show the experiment results for four games, our approach is not a specialised solution for these four games only. It is general enough to deal with all GDL games.

	MH	KTTT	Transit	Meier	Meier′
(1)	0.47	1864.81	12.17	6.41	8079.52
(2)	0.48	3528.14	7.54	9.75	13192.91
(3)	0.67	303.04	11.02	17.06	15056.29
(4)	0.60	22847.06	14.91	7.00	NA
(5)	0.56	22643.12	14.39	23.28	NA
(6)	0.43	5498.03	45.15	11.01	NA

Table 1: Experimental results on 5 translations.

5 Related Work and Further Research

There are a few papers on reasoning about games in GDL and GDL-II. Haufe *et al.* [2012] use Answer Set Programming for verifying temporal invariance properties against a given game description by structural induction. Haufe and Thielscher [2012] extend [Haufe *et al.*, 2012] to deal with epistemic properties for GDL-II. Their approach is restricted to positive-knowledge formulas while ours does not and can handle more expressive epistemic and temporal formulas.

Ruan *et al.* [2009] provide a reasoning mechanism for strategic and temporal properties but it is restricted on the original GDL for complete information games. Ruan and Thielscher [2011] examine the epistemic logic behind GDL-II and in particular show that the situation at any stage of a game can be characterized by a multi-agent epistemic (i.e., S5-) model. Ruan and Thielscher [2012] provide both semantic and syntactic characterizations of GDL-II descriptions in terms of a strategic and epistemic logic, and show the equivalence of these two characterizations. The current paper does not handle strategies but is able to provide practical results by using a model checker.

Kissmann and Edelkamp [2011] instantiate GDL descriptions and utilise BDDs to construct a symbolic search algorithm to solve single- and two-player turn-taking games with complete information. This is related to our work in the sense that we also do an instantiation of GDL descriptions and uses the BDD-based symbolic model checking algorithms of MCK to verify properties. But our approach is more general and also deals with games with incomplete information.

Other existing work is related to this paper in terms of dealing with declarative languages. Chang and Jackson [2006] show the possibility of embedding declarative relations and expressive relational operators into a standard CTL symbolic model checker. Whaley *et al.* [2005] propose to use Datalog (which GDL is based upon) with Binary Decision Diagrams (BDDs) for program analysis.

We conclude by pointing out some directions for further research. Firstly our results suggest that the optimization we have applied allows us to verify some formulas quickly, but it is still difficult to deal with a game like Krieg-TicTacToe. However a hand-made version of Krieg-TicTacToe (with more abstraction) in MCK does suggest that MCK has no problem to cope with the amount of reachable states of Krieg-TicTacToe. So the question is, what other optimization techniques can we find for the translation? Secondly, we would like to investigate how to make MCK language more expressive by allowing declarative relations such as shown in [Chang and Jackson, 2006]. Our current translation maps GDL-II to MCK's input and MCK internally encodes that into BDDs for symbolic checking. So a more direct map from GDL-II to BDDs may result in a gain in efficiency. Thirdly, we would like to explore the use of bounded model checking to check if a property holds for a partial game model. MCK has implemented some bounded model checking algorithms, but they are not yet particularly effective in dealing with perfect recall semantics that we used in our investigation. We leave all these for future work.

Acknowledgements We thank the anonymous reviewers for their helpful comments. This research was supported under Australian Research Council's (ARC) *Discovery Projects* funding scheme (project DP 120102023). Michael Thielscher is the recipient of an ARC Future Fellowship (project FT 0991348). He is also affiliated with the University of Western Sydney.

References

[Chang and Jackson, 2006] Felix Sheng-Ho Chang and Daniel Jackson. Symbolic model checking of declarative relational models. In Leon J. Osterweil, H. Dieter Rombach, and Mary Lou Soffa, editors, *ICSE*, pages 312–320. ACM, 2006.

[Clark, 1978] Keith Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[Clarke and Emerson, 1981] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In D. Kozen, editor, *Logics of Programs — Proceedings 1981 (LNCS Volume 131)*, pages 52–71. Springer-Verlag: Berlin, Germany, 1981.

[Fagin *et al.*, 1995] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning About Knowledge*. The MIT Press: Cambridge, MA, 1995.

[Gammie and van der Meyden, 2004] P. Gammie and R. van der Meyden. MCK: Model checking the logic of knowledge. In R. Alur and D. Peled, editors, *Proceedings of the 16th International Conference on Computer Aided Verification (CAV 2004)*, pages 479–483. Springer, 2004.

[Gelfond and Lifschitz, 1988] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *Proceedings of the International Joint Conference and Symposium on Logic Programming (IJCSLP)*, pages 1070–1080, Seattle, OR, 1988. MIT Press.

[Genesereth *et al.*, 2005] M. Genesereth, N. Love, and B. Pell. General game playing: Overview of the AAAI competition. *AI Magazine*, 26(2):62–72, 2005.

[Haufe and Thielscher, 2012] S. Haufe and M. Thielscher. Automated verification of epistemic properties for general game playing. In *Proceedings of the 13th International Conference on Principles of Knowledge Representation and Reasoning (KR 2012)*, 2012.

[Haufe *et al.*, 2012] S. Haufe, S. Schiffel, and M. Thielscher. Automated verification of state sequence invariants in general game playing. *Artificial Intelligence Journal*, 187–188:1–30, 2012.

[Kissmann and Edelkamp, 2011] Peter Kissmann and Stefan Edelkamp. Gamer, a general game playing agent. *KI*, 25(1):49–52, 2011.

[Lloyd, 1987] J. Lloyd. *Foundations of Logic Programming*. Series Symbolic Computation. Springer, second, extended edition, 1987.

[Love *et al.*, 2006] N. Love, T. Hinrichs, D. Haley, E. Schkufza, and M. Genesereth. General Game Playing: Game Description Language Specification. Technical Report LG–2006–01, Stanford Logic Group, Computer Science Department, Stanford University, 2006.

[Manna and Pnueli, 1992] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag: Berlin, Germany, 1992.

[Ruan and Thielscher, 2011] J. Ruan and M. Thielscher. The epistemic logic behind the game description language. In *Proceedings of the Conference on the Advancement of Artificial Intelligence (AAAI)*, pages 840–845, San Francisco, 2011.

[Ruan and Thielscher, 2012] J. Ruan and M. Thielscher. Model Checking Games in GDL-II: the Technical Report CSE-TR-201219, (2012)

[Ruan and Thielscher, 2012] J. Ruan and M. Thielscher. Strategic and epistemic reasoning for the game description language GDL-II. In *Proceedings of the European Conference on Artificial Intelligence (ECAI 2012)*, 2012.

[Ruan *et al.*, 2009] J. Ruan, W. van der Hoek, and M. Wooldridge. Verification of games in the game description language. *Journal Logic and Computation*, 19(6):1127–1156, 2009.

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

[Thielscher, 2010] M. Thielscher. A general game description language for incomplete information games. In *Proceedings of AAAI*, pages 994–999, 2010.

[Thielscher, 2011] M. Thielscher. The general game playing description language is universal. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, pages 1107–1112, Barcelona, 2011.

[van der Meyden *et al.*, 2012] R. van der Meyden, P. Gammie, K. Baukus, J. Lee, C. Luo, and X. Huang. User manual for MCK 1.0.0. Technical report, University of New South Wales, 2012.

[Whaley *et al.*, 2005] J. Whaley, D. Avots, M. Carbin, and M. S. Lam. Using datalog with binary decision diagrams for program analysis. In Kwangkeun Yi, editor, *APLAS*, volume 3780 of *Lecture Notes in Computer Science*, pages 97–118. Springer, 2005.

Comparison of GDL Reasoners *

Yngvi Björnsson and Stephan Schiffel
School of Computer Science / CADIA
Reykjavik University, Iceland
{yngvi,stephans}@ru.is

Abstract

The variety of open-source GDL reasoners available to newcomers to General Game Playing (GGP) lowers the technical barrier of entering the field. This variety, however, also makes it more complicated to decide on a fitting reasoner for a given GGP project, considering the project's objectives, ambitions, and technical constraints. This paper gives an overview of available GDL reasoners, discusses their main pros and cons, and most importantly quantifies their relative reasoning performance on a number of games (in terms of nodes searched per second), showing an order of magnitude difference in some cases. We similarly quantify the performance difference between game-playing systems specifically designed for playing a single game on the one hand and general game-playing systems on the other, witnessing up to several orders of magnitude difference.

1 Introduction

Games have played an important role as a testbed for advancements in the field of artificial intelligence ever since its inception. The focus was initially on developing general problem-solving systems, but gradually shifted towards building specialized high-performance game-playing systems capable of matching wits with the best humans. These highly specialized systems were engineered and optimized towards playing a single particular game at a world-class level. Examples of such game-playing systems achieving fame are CHINOOK (checkers), DEEPBLUE (chess), and LOGISTELLO (Othello) [Schaeffer and van den Herik, 2002].

The auspicious effect of increased computing speed was recognized early as deeper lookahead could greatly improve programs' playing strength. Thus, in addition to developing pure algorithmic enhancements much effort was invested in developing compact and efficient data-structures and inventive code optimization tricks for the games at hand (as a case in point, "magic bitboards" in chess allow the board representation to be transformed from being row-aligned to diagonally-aligned simply by performing a couple of carefully chosen multiplications — an extremely useful feature for efficiently generating the legal moves of the sliding pieces). It is not uncommon for specialized game-playing programs to explore millions of *nodes per second (nps)* on modern computer hardware, even on a single CPU. Massive parallel processing has the potential of scaling this already impressive performance up by a few orders of magnitude.

The International General Game Playing Competition [Genesereth *et al.*, 2005] renewed interest in more general approaches to computer game playing. In *General Game Playing (GGP)*, as opposed to creating highly-efficient game-playing agents for playing one specific game, the goal is to create agents capable of autonomously learning how to play a wide variety of games skillfully. The games can take various disparate shapes and forms. The principal game rules, such as what the goal of the game is and how the pieces move, are communicated to the GGP agents using a language called *Game Description Language (GDL)*. The responsibility is then on the agents to learn —without any human intervention— strategies for playing that game well. Obviously such general game-playing systems cannot be expected to achieve the same level of performance as their game-specific counterparts. It is nonetheless important for them to be as efficient as possible, not the least because the learning and reasoning mainly takes place in real-time during play.

In GGP, as any other new research field, it is important to attract new practitioner. One potential obstacle of entry is the sizable software infrastructure needed for having simply a functional GGP agent. Fortunately, the community provides software tools for lowering this technical barrier. For example, there are several open-source GGP agents available, online game servers for playing other agents, as well as various GDL reasoners that can be plugged directly into new projects. This frees newcomers from having to implement their own GDL interpreters, allowing them instead to concentrate on any of the other challenging aspects of GGP.

In this paper we give an overview of available GDL reasoners, discuss their main pros and cons, and quantify their relative reasoning efficiency on a number of games. We hope that this work not only provides added insights into how to further improve GDL reasoners' efficiency, but also makes it easier for both new and old GGP practitioners to select a proper reasoner for their projects or tasks at hand. We fur-

*The support of the Icelandic Centre for Research (RANNIS) is acknowledged.

```
1 (role xplayer)
2 (role oplayer)
3
4 (init (cell 1 1 b))
5 (init (cell 1 2 b))
6 ...
7 (init (control xplayer))
8
9 (<= (legal ?w (mark ?x ?y))
10   (true (cell ?x ?y b))
11   (true (control ?w)))
12 (<= (legal oplayer noop)
13   (true (control xplayer)))
14 ...
15 (<= (next (cell ?m ?n x))
16   (does xplayer (mark ?m ?n))
17   (true (cell ?m ?n b)))
18 (<= (next (control oplayer))
19   (true (control xplayer)))
20 ...
21 (<= (row ?m ?x)
22   (true (cell ?m 1 ?x))
23   (true (cell ?m 2 ?x))
24   (true (cell ?m 3 ?x)))
25 ...
26 (<= (line ?x)
27   (row ?m ?x))
28 (<= (line ?x)
29   (column ?m ?x))
30 (<= (line ?x)
31   (diagonal ?x))
32 ...
33 (<= (goal xplayer 100)
34   (line x))
35 (<= (goal xplayer 0)
36   (line o))
37 ...
38 (<= terminal
39   (line x))
```

Figure 1: A partial Tictactoe GDL description.

thermore provide initial benchmarks quantifying the (search speed) performance gap between game-specific and general game-playing systems, showing the efficiency of GGP systems leaving much to be desired.

The paper is organized as follows. Section 2 gives a brief background of GDL and common approaches for interpreting it. This is followed in Section 3 by an overview of publicly available GDL reasoners. Their efficiency is evaluated in Section 4, and, finally we conclude and discuss future work.

2 Game Description Language (GDL)

Games in GGP are described in a first-order-logic based language called *Game Description Language (GDL)* [Love *et al.*, 2008]. It is an extension of Datalog permitting negations and function constants. The extensions are restricted such that the language still has an unambiguous declarative interpretation. The expressiveness of GDL permits a large range of deterministic perfect-information simultaneous-move games with arbitrary number of adversaries to be described. Turn-based games are modeled by having the players that do not have a turn return a special *no-op* move. Special relations have a game-specific semantic, such as for describing the initial game state ($init$), detecting ($terminal$) and scoring ($goal$) terminal states, and for generating ($legal$) and playing ($next$) legal moves. A game state is represented by the set of facts that are true in the state (e.g., $cell(1, 1, b)$). Figure 1 shows a partial GDL description for the game Tictactoe. The official GDL specification [Love *et al.*, 2008] defines the language's syntax and semantics.

The three main approaches used by GGP agents for interpreting GDL game descriptions are: 1) using a custom-made GDL interpreter; 2) translating GDL to Prolog and then use an off-the-shelf Prolog engine to handle the interpretation; or, 3) translate GDL into some other alternative representation that the agents knows how to manage.

The first approach of using a custom-made GDL interpreter is probably the most straightforward for integrating a GDL reasoner into a new GGP project. Building a robust and efficient GDL interpreter from scratch is of course a highly involved task. However, there already exists several GDL interpreters written in popular imperative programming languages. These interpreters are though still in their infancy and tend to be quite inefficient (as we see later). Although they are typically modeled after Prolog interpreters, for example, by using SLD-NF (Selective Linear Definite–Negation as Failure) resolution [Apt and van Emden, 1982], they still lack many of the standard optimization techniques commonly found in Prolog interpreters. We evaluate three custom-made GDL interpreters, one written in C++ and two in Java.

The second approach of translating GDL game descriptions to Prolog and offload the interpretation to an (highly-optimized) off-the-shelf Prolog engine is most popular among established GGP agents. The translation of GDL to Prolog is mostly straightforward as both languages are first-order-logic based and share a similar syntax, however, their semantics differ somewhat. In particular, the ordering of clauses is inconsequential for the semantic in GDL (which is, as Datalog, fully declarative), whereas clause ordering is essential for determining a program's semantic in Prolog. Thus some precautions are necessary during the translation process to ensure that the correct semantic interpretation of GDL is preserved. For example, one must make certain that negated clauses in generated Prolog implication rules are ordered such that their variable arguments are surely grounded before execution. In contrast to the first approach, the integration and interfacing of a Prolog engine into a GGP project can be a somewhat more involved task. For example, not all Prolog engines provide a convenient or efficient application-programming interface to programs written in an imperative programming language. Another downside is that most publicly available Prolog engines are non-reentrant and as such cannot be safely used by host programs using thread-based parallelism. We evaluate three GDL reasoners using Prolog as a backend, one written in Prolog, another in C++, and the third in Java.

The third approach is to translate the GDL game description into an alternative representation (other than Prolog), efficiently managed by the GGP agent. A few GGP

agents do this, for example by translating GDL into propositional nets [Schkufza *et al.*, 2008] or binary decision diagrams [Bryant, 1985]. Whereas this can result in highly efficient reasoners for particular games, the main drawback is that such translations typically require the grounding of all possible GDL logic clauses, often leading to an exponential blowup of the size of the representation. Such an approach is thus feasible for only a subset of (typically the more simpler) games. These GGP agents do thus also rely on one of the two aforementioned approaches as a fallback. We chose to exclude these non-general reasoners in our comparison study; however, it would be interesting to include them in a more comprehensive future study.

3 Reasoners

This section catalogs available GDL reasoners.

3.1 CadiaPlayer

The CADIAPLAYER [Björnsson and Finnsson, 2009; Finnsson and Björnsson, 2008] agent is developed by the CADIA research lab at Reykjavik University. It is written in C++, but uses YAP Prolog [Costa *et al.*, 2012] as a backend for GDL game state reasoning. Although the agent is under constant development, the lowest level the code for interfacing the Prolog backend has not changed much since 2007.

The translation from GDL into Prolog is straightforward: Apart from syntactical changes the only change is that negative literals in the rules bodies are moved to the end. This is done to ensure safe evaluation of negations in Prolog, which uses a left to right evaluation and negation-as-failure semantics. For example, the following rule from Tictactoe

```
1 (<= (next (cell ?m ?n x))
2   (does xplayer (mark ?m ?n))
3   (true (cell ?m ?n b)))
```

is transformed to

```
1 next(cell(M, N, x)) :-
2   does(xplayer, mark(M, N)),
3   state(cell(M, N, b)).
```

As a consequence, these Prolog rules can only be used to reason about one particular state at a time. To set this state, facts of the form `state(F)` are asserted in Prolog – one for each fluent `F` in the current state. Similarly, to set the moves that the players have done, CADIAPLAYER asserts facts of the form **`does`**`(R,M)`. This allows to reason about the game. For example, to compute the legal moves of `xplayer` in the current state, CADIAPLAYER executes the query `findall(M,`**`legal`**`(xplayer,M),Moves)`. To change the state, the facts `state(F)` have to be retracted again before asserting the facts for the new state.

Asserting and retracting facts from the Prolog rules is a somewhat expensive operation. Depending on the Prolog interpreter that is used, it requires to actually compile the facts in to machine code of a Warren Abstract Machine [Warren, 1983]. However, this machine code is optimized for fast reasoning, such that inferences that involve lookup of facts from the current state will benefit from this approach. In fact, by taking advantage of YAP Prolog's indexing mechanism, lookup of facts can often be done in a small constant time, i.e., independent on the size of state.

3.2 FluxPlayer

FLUXPLAYER [Schiffel and Thielscher, 2007] was developed by the Computational Logic group at Technische Universität Dresden. It is written entirely in ECLiPSe Prolog [ecl, 2013], except for a small part handling the communication which is written in Java.

Consequently, FLUXPLAYER also converts the GDL rules into Prolog. However, the approach differs from CADIAPLAYER's by not needing to assert and retract facts. Instead states and players' moves are modeled as Prolog lists. Each predicate in the game rules that depends on the state or on the players' moves is extended by additional arguments. For example, the following rule from Tictactoe

```
1 (<= (next (cell ?m ?n x))
2   (does xplayer (mark ?m ?n))
3   (true (cell ?m ?n b)))
```

is transformed to

```
1 next(cell(M, N, x), State, Moves) :-
2   does(xplayer, mark(M, N), Moves),
3   true(cell(M, N, b), State).
```

The two predicates **`true`**`(F, State)` and **`does`**`(R, M, Moves)` are implemented similar to Prolog's built-in `member` predicate.

The advantage of this approach is that modification and storing of Prolog lists (and hence states) is efficient compared to asserting facts. However, the lookup time on a list is linear in its size. Thus, inferences that involve lookup of facts in a state can be more expensive than with asserted facts.

In addition to this standard transformation of GDL rules into Prolog rules, FLUXPLAYER comes with a multitude of game analysis algorithms (see, e.g., [Haufe *et al.*, 2012; 2011]), some of which are used to modify the game rules to decrease inference time. In the interest of a fair comparison of the reasoners, these improvements were turned off for the experiments presented in this paper.

3.3 JavaProver

JAVAPROVER [Halderman *et al.*, 2006] was written by students in Stanford University's cs227b General Game Playing course. It was provided by Stanford University as a reference player. JAVAPROVER is also embedded in GameController and GGPServer [Günther *et al.*, 2013], two popular programs for running matches and tournaments between general game players.

JAVAPROVER is a custom implementation of SLD-NF resolution for GDL. As the name suggests, JAVAPROVER is written in Java. It does not contain optimizations such as indexing or tabling that can be found in off-the-shelf Prolog systems.

For the experiments we used JAVAPROVER embedded in the GameController code. This added a small overhead to the runtime (1–2%), but allowed to reuse the code of the test-driver (see section 4.1) because GameController allows for an easy way to exchange the GDL reasoner.

3.4 JavaEclipse

JAVAECLIPSE is essentially an adapter allowing the GameController code-base to access a simplified version of FLUXPLAYER through ECLiPSe Prolog's Java interface. Therefore, the actual reasoner is very similar to FLUXPLAYER. However, the interface uses socket communication between the Java and ECLiPSe processes which incurs overhead, including having to encode/decode all Prolog clauses passed over the interface. A similar approach is used by the CENTURIO player [Möller *et al.*, 2011].

3.5 C++Reasoner

C++REASONER [Schultz *et al.*, 2008] was implemented by students the General Game Playing course at Technische Universität Dresden in 2008. Similar to JAVAPROVER, it is a custom implementation of SLD-NF resolution, completely implemented in C++.

3.6 GGPBaseProver

The ggp-base project [Schreiber, 2013] is an open source project with the goal of providing a common infrastructure for general game player. GGPBASEPROVER uses the SLD-NF resolution part of ggp-base embedded in the GameController code base. As with JAVAPROVER, this embedding incurs some small runtime overhead (1 to 2% depending on the game), but allows to reuse the code of the test-driver.

3.7 Others

We chose not to include in this study specialized GDL reasoners that translate GDL into an alternative representation as they are typically applicable for only a small subset of games (it would be of interest to get insights into their efficiency and expressiveness, but it's left as future work). We list below the reasoners omitted because of this or some other reason.

JOCULAR [joc, 2007] is another Java-based player that uses its own resolution based reasoner. However, it crashes for Amazons and has errors for some of the other games.

The *GGP-Base Project* [Schreiber, 2013] also provides an reasoner based on propositional networks [Schkufza *et al.*, 2008]. This approach can speed up reasoning by several orders of magnitude. However, it requires to propositionalize the game description. In general, this results in a significant increase of representation size, to the extend that propositionalizing games is only feasible for a small subset of games. For example, with the basic approach provided by ggp-base, only 2 out of the 12 games we tested could be propositionalized in a reasonable amount of time (5 minutes) and without running out of memory (2 GB). Since the approach is currently only usable for some of the games, we excluded it from the experiments.

GADELAC [Saffidine and Cazenave, 2011] is a compiler for GDL that generates a forward chaining reasoner in OCAML. One of the main differences to the other specialized approaches [Schreiber, 2013; Kissmann and Edelkamp, 2011] is that it does not require to propositionalize the game description. The forward chaining reasoner that is produced from a game description is similar in size to the original game description. In [Saffidine and Cazenave, 2011] the authors reported an improvement of up to 50% in number of random simulations of a game per time interval compared to a YAP Prolog-based reasoner. However, the results were mixed. For some games GaDeLaC performed considerably worse than YAP Prolog. We did not include the compiler in our comparison because our test-driver framework does not currently support the OCAML programming language; however, we plan to do so for future comparisons.

TOSS [Kaiser and Stafiniak, 2013] is a system in which games are defined using mathematical structures and structure rewriting rules. [Kaiser and Stafiniak, 2011] show how to translate GDL game descriptions into the Toss formalism. However, the translation depends on certain structures in the game rules and it might incur a prohibitively large increase in the size of the state representation.

GAMER [Kissmann and Edelkamp, 2011] is a general game player that propositionalizes game descriptions and use those propositional descriptions to solve games by classifying game states using binary decision diagrams [Edelkamp and Kissmann, 2008]. As reported in [Kissmann and Edelkamp, 2011], this can increase the performance by as much as two orders of magnitude. However, not all games can be propositionalized within reasonable time and memory constraints.

Kevin Waugh [Waugh, 2009] developed GDLCC, a compiler that transforms GDL rules into a game specific C++ library for performing state manipulation tasks. The generated C++ library implements a game specific backward-chaining reasoner that does not require a propositionalized game description. The author reports speed-up factors of up to 18 for state generations per second on some games compared to YAP Prolog. However, reportedly the system did not work for game descriptions with recursive terms and compilation time for the generated C++ code could be long for some games. A similar approach compiling GDL into Java code is implemented in the CENTURIO player [Möller *et al.*, 2011]. The authors report speed-up factors of up to three compared to the Prolog based version of CENTURIO.

4 Empirical Evaluation

In here we empirically evaluate the GDL reasoners. We describe the experimental setup followed by a performance comparison of the different reasoners, both in absolute and relative terms. This is reported in two separate subsections because the former comparison is mainly helpful for contrasting the GGP reasoners efficiency with their game-specific counterparts, whereas the latter is mostly helpful for gaining insights into the different GDL reasoning approaches. Finally, we also briefly report on the reasoners' robustness.

4.1 Experimental Setup

We built a test driver module for running homologous experiments across the different reasoners. The driver implements basic iterative-deepening minimax (MM) and Monte-Carlo (MC) search algorithms, as well as providing functionality for reading pre-recorded game records and for gathering search statistics. It was designed to be as low overhead as possible and with only minimal logic embedded (simplest possible implementations of MM and MC), such that the performance timing would be solely focused on the GDL reasoning

part. For example, neither minimax nor Monte-Carlo back up the values of the terminal states, however, minimax detects whether all leafs are terminal as in such cases further depth iterations are unnecessary. The driver updates the game state in between searches with the move from the pre-recorded game (as opposed to the best move based on the search) to ensure that the same benchmarking data is used for all reasoners throughout an entire match. The exact implementation of the test driver can differ slightly between reasoners to accommodate their different game-logic interfaces and programming languages; the test-driver implementations are, however, all functionally equivalent. Figure 2 gives a pseudo-code outline of the driver (the source code will be made available online).

```
1  boolean mm(Game g, int depth) {
2    if ( timeout() ) { return false; }
3    if (g.isTerminal()) {g.goals(); return true;}
4    if ( depth <= 0 ) { return false; }
5    boolean isTerminal = true;
6    for ( Move move : g.getActions() ) {
7      g.make(move);
8      isTerminal = mm(g, depth-1) && isTerminal;
9      g.retract(move);
10     if ( timeout() ) { return false; }
11   }
12   return isTerminal;
13 }
14
15 void algorithmIterativeDeepeningMM(State g) {
16   boolean isTerminal = false;
17   for (int d=0; !timeout()&&!isTerminal; ++d) {
18     isTerminal = mm(g, d);
19   }
20 }
21
22 void randomSimulation(Game g) {
23   if ( timeout() ) { return; }
24   if (g.isTerminal()) {g.goals(); return;}
25   List<Move> moves = g.getActions( );
26   int i = Random.nextInt(moves.size());
27   g.make(moves.get(i));
28   randomSimulation(g);
29   g.retract(moves.get(i));
30 }
31
32 void algorithmMonteCarlo(Game g) {
33   while ( !timeout() ) {
34     randomSimulation(g);
35   }
36 }
37
38 void runTest(Game g, Algorithm algorithm) {
39   List<String> record = readGameRecord();
40   g.reset();
41   for ( String moveStr : record ) {
42     algorithm( g );
43     Move move = g.strToMove( moveStr );
44     g.make( move );
45   }
46 }
```

Figure 2: The outline of the test-driver.

A test-suit of games consisting of the following two-player games is used for the experiments: *Amazons*, *Breakthrough*, *Chinese checkers*, *Connect4*, *Othello*, *Skirmish*, and *Tictactoe*. The complexity of the games span a wide spectrum and are thus representative of the type of games one would expect to encounter in GGP competitions (they have all been used in previous GGP competitions in one form or another). *Chinese checkers* can also be played with only a single player or as a multi-player game with up to six players. We include the 1, 3, 4, and 6 player variants in our test-suit as well as a 6-player simultaneous move variant. The games' GDL descriptions are all available on the Dresden GGP server [Schiffel, 2013].

For each game 100 complete match records were pre-recorded and used for the testing. Three experiments were executed for each reasoner: the first two using a time-limited iterative-deepening minimax (MM) and a time-limited Monte-Carlo search (MC), respectively, and then a third using a fixed-depth minimax search. Time limits for MM and MC were set to 60 seconds. The depth limit for the fixed-depth minimax search was determined on a per-game basis ranging from as low as 1 for *Amazons* up to 6 for *Tictactoe*. The former two experiments allow us to compare the relative performance of the different reasoners under tournament-like conditions, whereas the last experiment also helps with evaluating the correctness of the reasoners, as their node count statistics should (for the most part) match.

All experiments were run on a Linux (Ubuntu 12.04) server using a single-core of a quad-core 2.5 GHz Intel Xeon CPU (with 2 x 3 MB L2 cache and 1333 Mhz FSB). The computer had 4 GB of memory, but the computations were first and foremost CPU bound. Java 1.6 (OpenJDK), Gcc 4.6.3, YAP Prolog 5.1.1, and ECLiPSe Prolog 6.0#188 were used for compiling (and running) the programs.

4.2 Absolute Performance

We first establish a baseline for each game/algorithm combination. The baseline is the best (highest) score achieved by any reasoner for that particular combination and is used in subsequent graphs for normalizing all scores to the range $[0, 1]$. Table 1 shows the baseline values.

First, it is worth noting how slow even the fastest GDL reasoners are, with a typical search speed measured in mere thousands of *nodes per second (nps)*. For contrast, a typical search speed of programs specifically designed for playing these particular games would be measure in hundreds of thousand or even millions of nodes per second, resulting in a difference of at least two to three orders of magnitude. For a concrete performance comparison we implemented (in Java) game-specific programs for two of the games, Breakthrough and Connect4, confirming our intuition as shown in Table 2.

Second, we note that the nps counts are substantially lower for the MC-based search than the MM-based one. This is expected, and explained by the generation of all legal moves taking substantially longer time than playing a single move and updating the game state accordingly. In MM-based search, where all generated legal moves are explored at interior nodes, there are many state updates for each legal moves generation, whereas in MC, where only a single (random) move is explored, there is only one state update per legal

Table 1: Running speed in nodes per second (nps) of the fastest GDL reasoner for each game/algorithm: FLUXPLAYER was the fastest for all entries except the asterisk-marked ones, where CADIAPLAYER was the fastest.

Game	MM (nps)	MC (nps)	Ratio
Amazons	4,877	454*	20.6
Breakthrough	3,783	2,638	1.4
Chinese checkers 1p	6,249	4,538	1.4
Chinese checkers 2p	4,340	3,193	1.4
Chinese checkers 3p	3,195	2,464	1.3
Chinese checkers 4p	2,018	1,511*	1.3
Chinese checkers 6p	1,581	1,500*	1.1
Chinese checkers 6p-sim	2,400	1,023	2.3
Connect4	1,780	933*	1.9
Othello-2007	747*	258*	2.9
Skirmish	3,327	1,391	2.4
Tictactoe	14,471	11,988	1.2

Table 2: Running speed in nodes per second (nps) of the fastest GDL reasoner vs. game-specific reasoner.

Game	Exp.	Specific (nps)	GDL (nps)	Ratio
Breakthrough	MM	5,543,666	3,783	1,465
Connect4	MM	5,072,668	1,780	2,850
Breakthrough	MC	709,230	2,638	269
Connect4	MC	2,530,426	933	2,712

moves generation. We see from column three in Table 1 that this ratio may differ considerably from one game to the next. Two factors contribute to this: the average branching factor of the game tree (the higher the branching factor the higher the ratio), and the relative time complexity of legal moves generations compared to state updates. The speed difference ratio is particularly profound in the game Amazons, which is not surprising given the game's high branching factor.

The performance of Prolog based reasoners can be improved by optimizing the Prolog clauses that are generated from the GDL rules. To our knowledge, FLUXPLAYER is the only player currently doing this. The optimizations regarding game rules in FLUXPLAYER consist mainly of precomputing and tabling of static predicates (those predicates that do not depend on the current state of the game) and removing unnecessary fluents from the state representation (e.g., some versions of *Chinese checkers* contain cells on the board that no piece can be moved to). As we said, these optimization were turned off for the experiments. If switched on, these improvements result in up to two-fold speedup for some games. However, this does not change the huge performance gap between game-specific programs and general game players.

4.3 Relative Performance

Next we turn our attention to the relative performance of the reasoners. Figure 3 depicts the result for MM- and MC-search. The current trend in GGP is to use MC-based agents, making the latter graph more indicative of tournament play.

The first thing to notice is that the Prolog reasoners, FLUXPLAYER and CADIAPLAYER, are by far the most efficient. FLUXPLAYER's MM-search is the fastest in all games but one (Othello-2007), where CADIAPLAYER is a clear winner. For the MC-search the two players continue to be in a class of their own, but now perform more comparably: FLUXPLAYER being fastest in seven out of the twelve games but CADIAPLAYER in the remaining five. This shift in the agents' performance between MM and MC search can be explained by CADIAPLAYER's state updates being on average somewhat slower than that of FLUXPLAYER's. The effect of this is less profound in the MC search, because there is only a single state update per legal moves generation. The unlike approaches the two agents use for representing game states (as explained in subsections 3.1–3.2) apparently have their strengths and weaknesses. FLUXPLAYER's method seems to work better on average; however, as seen from the *Chinese checkers* multi-player results, the method used in CADIAPLAYER gains ground with increased number of players.

The second thing of interest is the huge relative performance swing in individual games, for example, the MC search of FLUXPLAYER is almost three times as fast as CADIAPLAYER's in *Skirmish*, but close to ten times slower in *Othello*. The main explanation for this is again the unlike approaches the agents use for representing game states.

Finally, it is evident that the remaining GDL reasoners are non-competitive to FLUXPLAYER and CADIAPLAYER. This is not too surprising for JAVAPROVER, C++REASONER, and GGPBASEPROVER, which are relatively immature reasoners in comparison to state-of-the-art Prolog interpreters. However, the poor performance of JAVAECLIPSE is somewhat surprising as it uses the same Prolog engine as FLUXPLAYER although through a different interface. Apparently, the inter-process communication application-programming interface used to communicate with ECLiPSe Prolog from a non-Prolog host programming language imposes excess overhead. It should be noted, that this overhead is to some extend operation system specific and may be influenced by settings of the loopback network adapter. However, we did not test this.

4.4 Robustness

To check the robustness of the reasoners we compared the node counts reported for the fixed-depth minimax searches.

As is to be excepted, the most established systems seem most robust. All three Prolog-based systems, FLUXPLAYER, CADIAPLAYER and JAVAECLIPSE, played without errors. However, CADIAPLAYER reports higher node counts for *Chinese checkers* and *Othello* because it does not remove duplicates from the legal moves.

JAVAPROVER turned out to be the most stable one of the custom-made GDL interpreters. It always crashed on *Othello* before completing the game, but produced the correct node counts without errors on all the other games.

GGPBASEPROVER also consistently crashed on *Othello* as well as occasionally on *Amazons*, but produced correct results for all the other games.

C++REASONER never crashed, but had occasional errors on all games except for *Amazons* and *Tictactoe*.

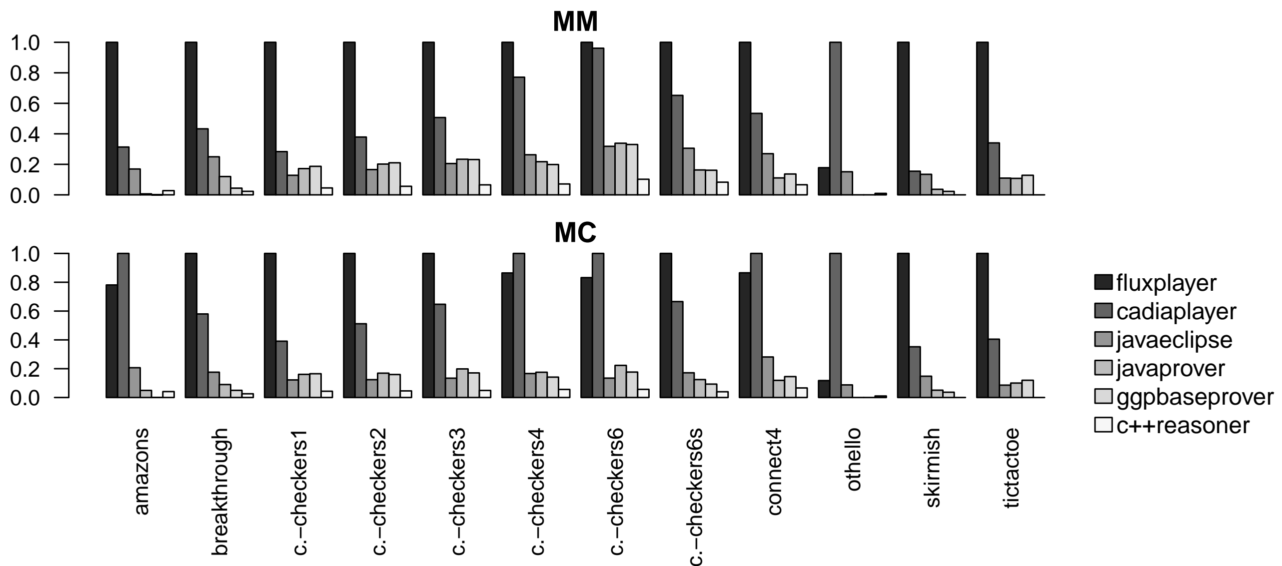


Figure 3: Running speed in nodes per second relative to the fastest reasoner for each of the games.

5 Conclusions and Future Work

Several conclusions can be drawn from this work. We list the main ones and discuss some implications they may have:

- *The GDL reasoners are at least two to three orders of magnitude slower than their game-specific counterparts.*

  Although one cannot expect generalized approaches to be quite as efficient as game-specific ones, this huge difference is nonetheless somewhat worrisome. For example, the slow state-space reasoning does exclude potentially interesting search and learning techniques from being effectively applied in GGP — this is particularly noticeable when using statistically based approaches, because there are simply not enough samples generated for getting meaningful statistics. Furthermore, in GGP it is important to validate ones research ideas on a wide range of games (and play hundreds of matches for each game to get statistical significance). The slow reasoning typically requires longer thinking times to be used per move than would otherwise be possible. This again can lead to excessively long turnarounds times for validating promising research ideas.

- *Of the GDL reasoners tested,* FLUXPLAYER *and* CADIAPLAYER *are by far the most efficient, both using Prolog engines as their backends.*

  FLUXPLAYER is overall the fastest GDL reasoner, however, it has the drawback that the given level of performance is only realized when the host program is written in Prolog. The ECLiPSe Prolog engine it uses as a backend does not provide an efficient API for a different host programming language. CADIAPLAYER is comparable in performance to FLUXPLAYER for MC search. Furthermore, the YAP Prolog backend used by CADIAPLAYER allows for a convenient and efficient API access for C/C++ programs.

- *The relative performance of the GDL reasoners can be quite game dependent.*

  This is particularly visible with FLUXPLAYER and CADIAPLAYER. Neither of the two reasoner dominates the other, and their relative performance is highly dependent on the game at hand, ranging from one being three times slower to ten times faster. This suggests that there may still be substantial scope for improvements for GDL reasoner by detecting properties that affect their performance adversely and use appropriate representations.

- *The tested non-Prolog based reasoners are non-competitive efficiency wise.*

  As mentioned before, this is somewhat understandable as they are relatively immature in contrast to state-of-the-art Prolog systems. However, they still have at least two advantages over the Prolog-based reasoners: 1) They are easier to integrate into GGP projects written in the same host programming language (i.e., Java or C++) and may thus in some cases be a better choice; 2) They can be be used by GGP programs using thread-based parallelization, which is problematic for the Prolog-based GDL reasoners as the underlying Prolog systems are non-reentrant (parallelization in GGP agents such as FLUXPLAYER and CADIAPLAYER is thus on the process level).

- *Robustness is a problem with some of the less mature publicly available GDL reasoners.*

As for future work a more comprehensive performance comparison of existing GDL reasoners would be valuable. A similar study focusing on GDL-II [Thielscher, 2010] reasoners would also be of interest. The results also suggests that it might be a good idea to combine the relative strengths of the FLUXPLAYER and CADIAPLAYER reasoners, for example, by altering the state updates in CADIAPLAYER to more in line with that of FLUXPLAYER. Also, a concentrated effort

into building a well-documented state-of-the-art non-Prolog based GDL reasoner for use by the GGP community at large would be quite useful. Such an undertaking would have potentials for combining the best of both worlds: a robust and efficient GDL reasoner that is both easily integrated into GGP projects written in a popular imperative programming language and that permits thread-based parallelism.

References

[Apt and van Emden, 1982] Kristoff R. Apt and Maarten H. van Emden. Contributions to the theory of logic programming. *Journal of the ACM*, 29(3):841–862, 1982.

[Björnsson and Finnsson, 2009] Yngvi Björnsson and Hilmar Finnsson. Cadiaplayer: A simulation-based general game player. *IEEE Trans. Computational Intelligence and AI in Games*, 1(1):4–15, 2009.

[Bryant, 1985] Randal E. Bryant. Symbolic manipulation of boolean functions using a graphical representation. In *Proc. of the 22nd ACM/IEEE Design Automation Conference (DAC '85)*, pages 688–694, Los Alamitos, Ca., USA, June 1985. IEEE Computer Society Press.

[Costa *et al.*, 2012] Vítor Santos Costa, Ricardo Rocha, and Luís Damas. The YAP prolog system. *TPLP*, 12(1-2):5–34, 2012.

[ecl, 2013] The ECLiPSe constraint programming system. http://www.eclipseclp.org/, 2013.

[Edelkamp and Kissmann, 2008] Stefan Edelkamp and Peter Kissmann. Symbolic classification of general two-player games. In *German Conference on Artificial Intelligence (KI)*, pages 185–192, 2008.

[Finnsson and Björnsson, 2008] Hilmar Finnsson and Yngvi Björnsson. Simulation-based approach to general game playing. In *Proc. of the 23rd AAAI Conference on Artificial Intelligence, AAAI*, pages 259–264. AAAI Press, 2008.

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

[Günther *et al.*, 2013] Martin Günther, Stephan Schiffel, and Sam Schreiber. GameController/GGPServer project. http://ggpserver.sourceforge.net/, 2013.

[Halderman *et al.*, 2006] Nick Halderman, Michael Tung, Justin Flatt, and Andrew Willis-Woodward. Javaprover. http://games.stanford.edu/resources/reference/java/java.html, 2006.

[Haufe *et al.*, 2011] Sebastian Haufe, Daniel Michulke, Stephan Schiffel, and Michael Thielscher. Knowledge-based general game playing. *KI*, 25(1):25–33, 2011.

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

[joc, 2007] Jocular reference player. http://games.stanford.edu/resources/reference/jocular/jocular.html, 2007.

[Kaiser and Stafiniak, 2011] Łukasz Kaiser and Łukasz Stafiniak. Translating the game description language to Toss. In *Proc. of the IJCAI-11 Workshop on General Game Playing (GIGA'11)*, 2011.

[Kaiser and Stafiniak, 2013] Łukasz Kaiser and Łukasz Stafiniak. Toss. http://toss.sourceforge.net/, 2013.

[Kissmann and Edelkamp, 2011] Peter Kissmann and Stefan Edelkamp. Gamer, a general game playing agent. *KI*, 25(1):49–52, 2011.

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

[Möller *et al.*, 2011] Maximilian Möller, Marius Schneider, Martin Wegner, and Torsten Schaub. Centurio, a general game player: Parallel, java- and ASP-based. *KI*, 25(1):17–24, 2011.

[Saffidine and Cazenave, 2011] Abdallah Saffidine and Tristan Cazenave. A forward chaining based game description language compiler. In *Proc. of the IJCAI-11 Workshop on General Game Playing (GIGA'11)*, 2011.

[Schaeffer and van den Herik, 2002] J. Schaeffer and H. J. van den Herik. *Chips challenging champions: Games, computers and artificial intelligence*. Elsevier, 2002.

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

[Schiffel, 2013] Stephan Schiffel. GGPServer. http://ggpserver.general-game-playing.de/, 2013.

[Schkufza *et al.*, 2008] Eric Schkufza, Nathaniel Love, and Michael R. Genesereth. Propositional automata and cell automata: Representational frameworks for discrete dynamic systems. In *Proc. of the 21st Australasian Joint Conference on Artificial Intelligence*, pages 56–66, 2008.

[Schreiber, 2013] Sam Schreiber. The general game playing base package. http://code.google.com/p/ggp-base/, 2013.

[Schultz *et al.*, 2008] Norbert Schultz, Norbert Manthey, and David Müller. C++-reasoner. http://www.general-game-playing.de/downloads.html, 2008.

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

[Warren, 1983] David H. D. Warren. An abstract Prolog instruction set. Technical Note 309, SRI International, Menlo Park, CA, October 1983.

[Waugh, 2009] Kevin Waugh. Faster state manipulation in general games using generated code. In *Proc. of the IJCAI-09 Workshop on General Game Playing (GIGA'09)*, 2009.

Online Adjustment of Tree Search for GGP

Jean Méhat & Jean-Noël Vittaut
Université de Paris 8 Vincennes–Saint Denis
France
jm@ai.univ-paris8.fr jnv@ai.univ-paris8.fr

Abstract

We present an adaptive method that enables a General Game Player using Monte-Carlo Tree Search to adapt its use of RAVE to the game it plays. This adaptation is done with a comparison of the UCT and RAVE prediction for moves, that are based on previous playout results. We show that it leads to results that are equivalent to those obtained with a hand tuned choice of RAVE usage and better than a fit-for-all fixed choice on simple ad hoc synthetic games. This is well adapted to the domain of General Game Playing where the player cannot be tuned for the characteristics of the game it will play before the beginning of a match.

1 Introduction

It this introduction, we present the General Game Playing (GGP) and the Monte-Carlo Tree Search with UCT and RAVE used by the General Game player we use for this study.

1.1 General Game Playing

Since its definition by the Logic Group of the Stanford University in 2005 [Genesereth *et al.*, 2005], GGP has given rise to research in the field of computer playing programs that are able to play a large class of different games without modification. It is an important step to create a unified framework able to cover the common aspects of game playing.

In GGP, games are described with the General Description Language (GDL); it allows the description of any finite deterministic game with complete information [Genesereth, 2006]. GDL is based on first order logic with negation as failure; most notably it does not include arithmetic that has to be defined as needed in the game description. It is supplemented with a few keywords (see table 1). Based on the *Knowledge Interchange Format* (KIF), GDL has a syntax reminiscent of Lisp and is semantically very similar to Datalog.

An extension of GDL, named GDL2, allows the description of games with incomplete information [Thielscher, 2010]. It adds only a keyword *(sees p x)* to describe the percepts of each player. The player name *random* is also reserved for a source of non-determinism. These extensions to GDL have been proved to be sufficient to make it universal [Thielscher, 2011].

(role p)	p is a player
(legal p m)	move m is legal for player p
(does p m)	player p played the move m
terminal	the match is finished
(goal p n)	player p got n points ($0 \leq p \leq 100$)
(init x)	x is true in the initial position
(true x)	x is true in the current position
(next x)	x will be true after the current move

Table 1: GDL keywords used to describe a game in the context of first order logic.

Every year since 2005, there is an international GGP competition hosted by the AAAI or IJCAI conferences where different teams pit their players against each other on new games designed by the organizers.

1.2 Monte-Carlo Tree Search

Since 2008, most of the players participating in the GGP competition use some kind of Monte-Carlo Tree Search (MCTS). The base of MCTS is to combine a stochastic sampling of the search space and the buildup of a tree of game positions linked by possible moves.

An exploration is made of four phases: selection of a tree leaf, tree growth, playout to the end of the game and update of the tree nodes.

The *selection* of a tree leaf is done with a descent in the tree. During this descent, previous sampling results are used to select the parts of the tree where exploration is promising. When this descent reaches a node with unplayed moves, a move is selected and a new leaf is built and added to the tree, and a *playout* is performed: successive moves are selected according to a given policy and played until a terminal situation is reached. The result, as described by the game rules, is used to update the nodes and/or edges forming the path built during the descent in the tree.

After the pioneer work of Brugmann [Brügmann, 1993], MCTS methods have been used with great success in the game of Go where they allow programs to

reach the level of the best human players on small boards [Coulom, 2007]. They are now applied to many fields of Artificial Intelligence with many variants [Browne *et al.*, 2012]. An attractive characteristic of MCTS is that it does not rely on a heuristic evaluation of a game situation. In GGP there is no general method known to build a reliable heuristic.

Upper Confidence bound applied to Trees
During the descent phase, one has to make a compromise between *exploration*: the selection of less visited branches and *exploitation*: accumulating visits in parts of the tree where previous samplings gave good results. This dilemma is frequently solved using *Upper Confidence bound applied to Trees* (UCT) [Kocsis and Szepesvári, 2006].

With UCT, the move selected during the descent in the tree is one that maximizes

$$\mu_i + C \times \sqrt{log(t)/s_i}$$

where μ_i is the mean result of the playouts starting with the move, t is the total number of playouts played in the current node and s_i is the number of playouts starting with this move. The constant C, named the *UCT constant* is used to adjust the level of exploration of the algorithm: high values favor exploration and low values favor exploitation.

Rapid Action Value Estimates
When using bare UCT, the first move selections in a given node of the tree have to be made according to a statically encoded heuristic (not possible in GGP) or at random, as long as there is not enough playout results to guide the selection. To alleviate this inconvenience, most Go playing programs use some variant of *All Moves As First* (AMAF): the back-propagation of the playout results takes into account the move played *and the subsequent moves*.

The currently most common variant of AMAF is *Rapid Action Value Estimates* (RAVE): a node contains a table associating legal moves with the results of all the playouts where these moves were selected during the playout. This table is used to obtain a RAVE estimation that is combined with the UCT estimation in a way that depends on the number of playouts starting with this move: the move choice is principally based on the RAVE estimation when there are few playouts; the importance of the UCT estimation grows with the number of playouts starting with this move [Gelly and Silver, 2007].

2 Implementation of RAVE in our General Game Player

We detail here the precise implementation of RAVE in our General Game Player that we used for the experiments.

Each edge in the built part of the game tree contains the mean result of the explorations that went through this edge. Each node contains a RAVE table associating each legal move with the mean result of all the playouts that went through this node where this move was played later on.

In the back-propagation phase, a *RAVE estimation* is updated in each node with the mean playout results for each legal move that was selected during the playout.

In the tree selection part, a *move score* is computed as follows:

- if there was a playout starting with this move, its score is the mean of the results of the playouts that started with this move; if no playout has been played starting with this move, it receives a *default mean score* that we fixed to the maximum possible result after informal tests. This ensures that unexplored moves are preferred over sub-optimal explored moves.
- when the move has been used at the beginning of a playout, the *mean score* is replaced by a *UCT score* u using the upper confidence bound computed with the usual UCT formula. This ensures that a move giving always good results in a few playouts will receive a better score than a never explored move. As the number of experiments on this move grows, the upper confidence bound on its mean value will decrease and other moves will be explored. This property is particularly desirable in GGP where the number of playouts can remain small with the usual time settings, due to GDL interpretation time.
- if the move has been used later in some playouts, a *RAVE score* r is computed as the mean of these playout results; a *RAVE influence factor* α is computed as $\alpha = \sqrt{k/3t + k}$ and the move final score is $(1 - \alpha)u + \alpha r$; k is the *RAVE equivalence constant* balancing the weight of UCT and RAVE. RAVE will influence the selection more when there are few playouts while UCT will have the greatest influence when the number of playouts grows.

Finally, the move is selected pseudo-randomly between those having the maximum score.

3 Online adjustments of RAVE usage

Finsson et al. show in the context of General Game Playing that RAVE can bring an advantage on some games (Checkers, Othello) while it can be detrimental for some others (Skirmish) [Finnsson and Björnsson, 2010].

What we are interested in is whether it is possible to dynamically adapt RAVE usage to the characteristics of the game. The player has no knowledge of the characteristics of the game and the static analysis of its properties based on its description is difficult.

We first study the degradation of the results obtained when only some choices of edges in the playouts, selected at random, are made using RAVE: this way there are always some choices the RAVE usage of which is optimal for the characteristics of the game at hand. We show in the experiments section that as the usage of RAVE augments, the results are modified in a nearly linear way. So mixing randomly playouts using RAVE and playouts

not using it does not give good results, since what is gained on games where RAVE is beneficial is lost on games where it is detrimental.

One would like to use online learning on the information gathered in the first playouts to deduce some properties of the game and use Rave only when it is profitable. When the tree is built and playout results are used in the back-propagation phase to update moves characteristics in the nodes, data is accumulated on RAVE and UCT estimations. This data can be used to adjust RAVE usage: increasing RAVE usage when RAVE estimation is better or nearly identical to UCT estimation gives overall results that show only a slight degradation to those obtained when RAVE usage is specifically adapted to the game at hand.

4 Games designed for RAVE

In this section, we present three games specifically designed to present characteristics where RAVE gives a significant advantage or disadvantage. They are tweaked versions of *Sum Of Switches* games (SOS).

There are at least two kinds of situations where RAVE is known to hinder the results of explorations. First when a move is good if played as first move but bad if played later; as the move played later leads to bad results, its exploration as a first move is not encouraged by RAVE; it occurs usually in Go in the context of *semeai* or *tsumego* problems where the first move is crucial and has to be played first to be of some value.

RAVE is detrimental in a second kind of situations: when moves are good when played later on but bad if played as first moves, RAVE evaluation is so good that it favors their exploration. It leads to bad choices of the part of the game tree to explore. When the number of explorations is used for the selection of the move to play it actually can lead to the choice of a bad move. This typically occurs in Go with *ko* threats: a *ko* threat is a good move when played at the right time but is silly if played before the beginning of the *ko* fight or when the *ko* can be taken back.

We use synthetic games to specifically embody these situations: *Blind Cashing Checks* that was also studied in [Tom and Müller, 2010] under the generic name *Sum Of Switches* and another tweaked version *Cashing Stale-dated Checks* with characteristics of the first kind that makes RAVE detrimental. We also present another tweaked version of *Cashing Checks* that we call *Cashing Post Dated Checks* that belongs to the second kind that is not used in this study.

4.1 Sums of switches: *Cashing Checks*

Berlekamp and al. present the family of games named *Cashing Checks* [Berlekamp *et al.*, 1982, p. 120]. The material is a set of bearer checks for certain amounts; a player move consists in taking one of the checks for his own. At the end of the game, the sum of each player checks are summed up and the winner is the player that holds the largest amount.

This game is a *Sum Of Switches* (SOS); in the context of Combinatorial Game Theory it is a sum of sub-games the values of which are either $+n$ or $-n$ depending on the player who takes the check; these are *switches*, noted $\pm n$.

The best strategy is naturally to take the check for the largest amount that remains on the board. In a game starting with k checks with amounts n_i, the first player will score $\sum_{1<2i<k} n_{2i}$ and the second $\sum_{0<2i+1<k} n_{2i+1}$ when $n_i \geq n_{i+1}$.

4.2 *Blind Cashing Checks*

The game becomes more interesting when the players are *innumerate*, i.e. are not able to read or compare numbers: they have to select the checks without knowledge of the amount that is written on it. At the end of the game, an arbiter sums the amounts gained by each player and announces the winner. This final result is the only clue the players get on the value of the checks. We call this game *Blind Cashing Checks*. This game was already used to study properties of RAVE [Tom and Müller, 2010].

RAVE works well for a MCTS player at *Blind Cashing Checks*: the player who takes the checks with the largest amounts wins this playout, without consideration of the turn where she took the check, so RAVE will promote the choice of these checks in the first turns, leading to the winning strategy.

The difficulty of the game can be adjusted by varying either k the number of turns or N the number of checks as long as $N > k$. As Tom et al., we use checks referring to the first N non-zero positive integers and compensate the first player advantage by setting a *komi* of $k/2$. If both players play optimally the sums of the amounts written on checks held by both players (plus *komi* for the second player) are equal and the game is declared a draw.

4.3 *Cashing Stale-Dated Checks*

We tweak the game of *Cashing Checks* to have a game where RAVE will be detrimental: on each check we add a limit of validity under the form of a turn of the match; if a player takes a check before this turn, she cashes the amount written on the check; if she takes it after this turn, the date is stale and it gives no point at all.

The date of each check is the turn where it is taken when both players play the optimal strategy at *Blind Cashing Checks*. The check with the biggest value amounts to nothing after the first move, the second one after the second turn and so on. This modification of the game allows to modelize the situations where RAVE is a disadvantage because a move is good if played as first move but bad if played later.

The limit of validity of the checks introduces another winning strategy: a winning move for a player is either to take the valid check with the largest amount *or* to take the check with the next amount, as the check with the largest amount will give nothing after this move. This way, the first player can force the second player to take

a check that amounts to nothing at the last turn if the number of checks and the number of turns are equal. To avoid this issue we use at least one more check than there are turns in a match (i.e. $N > k$).

4.4 *Cashing Post Dated Checks*

Another variation is *Cashing Post Dated Checks*. In this game, each check is dated with a turn; it amounts to nothing if taken before this turn and for the sum written on it if taken on this turn or any subsequent turn.

We set the date for each check according to this amount: a check for a sum of $N - k + i$ is valid only on turn i and subsequent turns. The optimal strategy for both players is then to take the checks in the reverse order, starting with the one that has the least amount and finishing with the one with the biggest amount. A reverse *komi* compensates the advantage of the second player.

This allows to modelize the situation when a move is good if played at the right time but bad if played sooner. We did not use this game in the experiments as this does not seem to be a crucial issue in the current common General Game Playing settings.

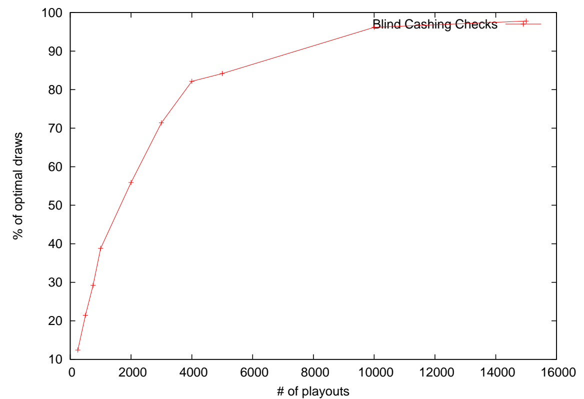


Figure 1: The percentage of matches with optimal draws at *Blind Cashing Checks* as a function of the number of playouts per move.

5 Experimental settings

For the experiments we have used a GDL description of *Blind Cashing Check* using twenty checks and ten turns with a *komi* of five points. For *Cashing Stale-dated Checks* the number of turns was also set to ten but the number of checks was limited to twelve to give comparable results.

Both players were instances of our General Game Player Ary [Méhat and Cazenave, 2010] in its usual setting: UCT with an exploration constant of 0.4 (actually 40 since the reward of a player can vary between 0 and 100) and transposition tables.

For the experiments, two players with the same parameters played together 500 complete matches and the percentage of draws with optimal moves by both players (*optimal draw*) was counted; a larger percentage indicates that the players played well, so the value of the parameter is well suited to the task at hand.

To fix the number of playouts used by the players to select a move, we studied the results obtained when this number varies. The results are presented in figure 1: the results are better as the number of playouts grows but not linearly. Later on, we set the number of playouts to 2000 for the experiments, where the number of optimal draws was over 50% in order to leave some space for improvements when RAVE is used.

With the previous settings, the value of the RAVE equivalence constant was made to vary for the games *Blind Cashing Checks* and *Cashing Stale-dated Checks* (see figure 2). As the value of the equivalence constant becomes greater, the level of play gets better at *Blind Cashing Checks* and gets worse at *Cashing Stale-dated Checks*. Here also the influence on the results is more obvious for the small values of the constant.

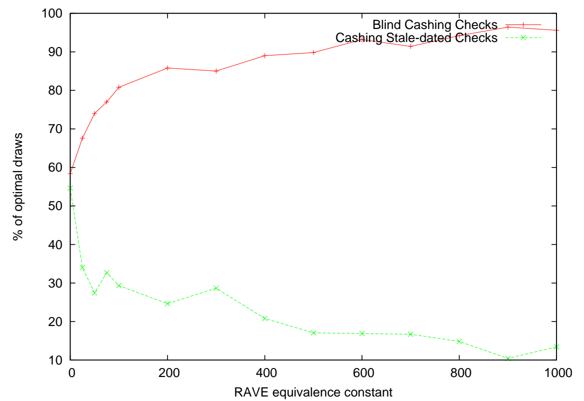


Figure 2: The percentage of matches with optimal moves of both players at *Blind Cashing Checks* and *Cashing Stale-dated Checks* as a function of the RAVE equivalence constant.

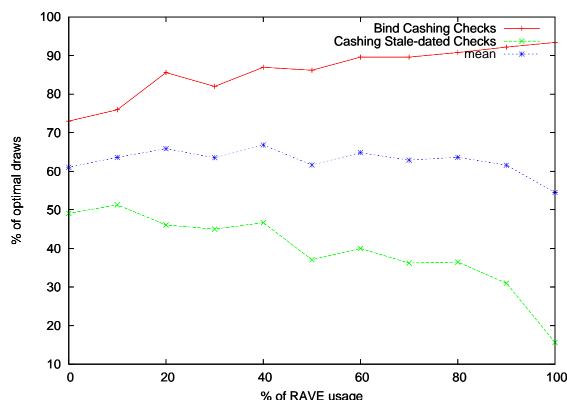


Figure 3: The percentage of draw games with optimal play of both players at *Cashing with Stale-dated Checks* and their mean as a function of the RAVE usage.

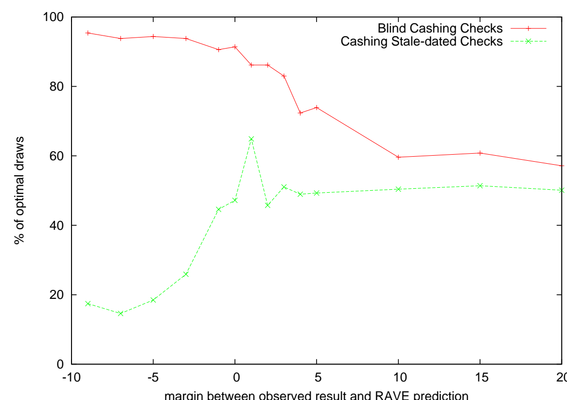


Figure 4: The percentage of optimal draws at *Blind Cashing Checks* and *Cashing Stale-dated Checks* as a function of the margin used to consider RAVE move prediction to be too optimistic.

Given these results, we choose to set the RAVE equivalence constant to 700 for the following experiments, as this value appears in the floor where small modifications of the constant do not modify significantly the results.

6 Using RAVE only for some choices

We first explore what happens when a player uses RAVE for some choices during the descent and does not use it for other choices. With the RAVE equivalence constant set to 700, we vary the percentage of children selections in the descent phase of UCT where RAVE is used.

As it descends in the built tree, a (pseudo-)random number is used to decide if the next edge will be chosen using only UCT or if RAVE is to be used as described in section 2. The results give a measure of the importance of using RAVE or not using it systematically on every choice.

The results are summarized in figure 3: as expected, the level of play, reflected by the number of draws, diminishes for *Blind Cashing Checks* as RAVE is used more often, while it gets better for *Cashing Stale-dated Checks*.

7 Comparing observed results and RAVE predictions

As the tree is built, the results of the playouts are accumulated in edges and nodes to be used as a basis for the predictions of UCT and RAVE. We propose to compare observed results with the RAVE predictions to get an evaluation of when to use RAVE. This evaluation will not be precise, but as shown by the previous experiment, one can expect to get a result that is better than a *fit for all* setting and that is proportional to the precision of the evaluation.

As a measure of the precision, we sum at the root node the number of playouts for the moves where RAVE gave an estimation that was higher than the observed mean result and the number of moves where it is lower, with an error margin. When the number of playouts starting with moves where RAVE was optimistic is greater than the number of playouts starting with moves where it is pessimistic, RAVE is used for the choices of children of the next descent in the built tree.

The results are presented in figure 4. With a negative margin the level of play at *Blind Cashing Checks* diminishes slightly while it gets much better for *Cashing Stale-dated Checks* as the margin tends to zero. With a positive margin, the level of play at *Cashing Stale-dated Checks* stays about the same when the margin augments, while it decreases at *Blind Cashing Checks*.

When the margin is set to 0, the percentage of optimal draws is 91.40 % at *Blind Cashing Checks* and 47.20 % at *Cashing Stale-dated Checks*. This figures are to be compared with those obtained with the best setting for a game: 93.39 % at *Blind Cashing Checks* when always using RAVE and 49.10 % at *Cashing Stale-dated Checks* when never using RAVE: the player adapts successfully its use of RAVE to the game at hand.

8 Conclusion and perspectives

We have shown that it is possible to use the comparison of the mean results observed during playouts with the results predicted by RAVE to adapt the use of RAVE to the characteristics of the game. This way, one gets an overall result that is equivalent to what can be obtained with a usage of RAVE tuned before the match beginning.

The results presented here were obtained on synthetic games designed specifically to emphasize characteristics that make RAVE beneficial or detrimental. The observations have to be extended to the third synthetic game, to real games whose characteristics regarding RAVE are less bold and in realistic playing situations where the number of playouts can be much smaller due to the slowness of GDL interpretation.

There are many other ways to measure the correlation between observed playout results and RAVE predictions that have to be explored. We intend to investigate if this

correlation can be used to adapt the value of the *RAVE* equivalence constant.

The method used here would not work in another synthetic game built by summing the two games used, for example alternating a move in *Blind Cashing Checks* and a move in *Cashing Stale-dated Checks* because the comparison between observed results and RAVE prediction was always calculated on the root node. It would be possible to observe this correlation at every node when it stores enough playout results.

More generally, the method presented here uses information that is already present in the tree built by the MCTS to determine characteristics of the game it plays. It could also be interesting outside of GGP for games where characteristics regarding RAVE vary depending on the position.

References

[Berlekamp *et al.*, 1982] Elwyn R Berlekamp, John Horton Conway, and Richard K Guy. *Winning ways for your mathematical plays. Volume 1.* Academic Press, 1982.

[Browne *et al.*, 2012] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *Computational Intelligence and AI in Games, IEEE Transactions on*, 4(1):1–43, 2012.

[Brügmann, 1993] Bernd Brügmann. Monte carlo go, 1993.

[Coulom, 2007] Rémi Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *Computers and games*, pages 72–83. Springer, 2007.

[Finnsson and Björnsson, 2010] Hilmar Finnsson and Yngvi Björnsson. Learning simulation control in general game-playing agents. In *Proc. 24th AAAI Conf. Artif. Intell., Atlanta, Georgia*, pages 954–959, 2010.

[Gelly and Silver, 2007] Sylvain Gelly and David Silver. Combining online and offline knowledge in uct. In *Proceedings of the 24th international conference on Machine learning*, pages 273–280. ACM, 2007.

[Genesereth *et al.*, 2005] Michael Genesereth, Nathaniel Love, and Barney Pell. General game playing: Overview of the aaai competition. *AI magazine*, 26(2):62, 2005.

[Genesereth, 2006] Michael Genesereth. General game playing: Game description language specification, 2006.

[Kocsis and Szepesvári, 2006] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. *Machine Learning: ECML 2006*, pages 282–293, 2006.

[Méhat and Cazenave, 2010] Jean Méhat and Tristan Cazenave. Ary, a general game playing program. In *13th board game studies colloquium*, pages 168–170, 2010.

[Thielscher, 2010] Michael Thielscher. A general game description language for incomplete information games. In *Proceedings of the Twenty-fourth National Conference on Artificial Intelligence (AAAI 2010)*, pages 994–999, 2010.

[Thielscher, 2011] Michael Thielscher. The general game playing description language is universal. In *Proceedings of the Twenty-Second international joint conference on Artificial Intelligence-Volume Volume Two*, pages 1107–1112. AAAI Press, 2011.

[Tom and Müller, 2010] David Tom and Martin Müller. A study of uct and its enhancements in an artificial game. *Advances in Computer Games*, pages 55–64, 2010.

Annex: *Blind Cashing Checks* GDL Description

```
(nstep 10)    ; ten steps
(maxvalue 20) ; twenty checks
(komi 5)      ; komi is 5 points

;; two players
(role left)
(role right)

;; legal moves
(<= (legal ?p ?v)
    (true (control ?p))
    (value ?v)
    (not (true (played ?v ?s))))
(<= (legal ?p noop)
    (true (notcontrol ?p)))

;; alternate play
(init (control left))
(init (notcontrol right))
(<= (next (control ?p)) (true (notcontrol ?p)))
(<= (next (notcontrol ?p)) (true (control ?p)))

;; turns
(init (step 0))
(<= (next (step ?n+1))
    (true (step ?n))
    (+ 1 ?n ?n+1))
(<= terminal (nstep ?nstep) (true (step ?nstep)))

;; maintain sum for each player
(init (sum left 0))
(init (sum right 0))

(<= (next (sum ?p ?n+m)) ; play valid check
    (role ?p)
    (true (sum ?p ?n))
    (does ?p ?m)
    (+ ?m ?n ?n+m))

(<= (next (sum ?p ?n))
    (role ?p)
    (true (sum ?p ?n))
    (does ?p noop))

;; do not take the same check twice (with transpo)
(<= (next (played ?v ?s)) (true (played ?v ?s)))
(<= (next (played ?v somestep)) (does ?p ?v))

;; goal
(<= l>r
    (true (sum left ?l))
    (true (sum right ?r))
    (komi ?k)
    (+ ?k ?r ?r+k)
    (gt ?l ?r+k))
(<= r>l
    (true (sum left ?l))
    (true (sum right ?r))
    (komi ?k)
    (+ ?k ?r ?r+k)
    (gt ?r+k ?l))

(<= (goal left 100) l>r)
(<= (goal left 0) r>l)

(<= (goal right 0) l>r)
(<= (goal right 100) r>l)

(<= (goal ?p 50)
    (role ?p)
    (not l>r)
    (not r>l))

; values of the checks
(<= (value ?n)
    (gt ?n 0)
    (maxvalue ?max)
    (not (gt ?n ?max)))

;; arithmetic: addition, comparison
(<= (+ 0 ?x ?x))
(<= (+ ?a ?b ?a+b)
    (++ ?a-1 ?a)
    (++ ?b ?b+1)
    (+ ?a-1 ?b+1 ?a+b))
(<= (gt ?a ?b) (++ ?b ?a))
(<= (gt ?a ?b) (++ ?a-1 ?a) (gt ?a-1 ?b))

;; integers
(++ 0 1) (++ 1 2) ...
```

Stratified Logic Program Updates for General Game Playing

David Spies
dspies@ualberta.ca
University of Alberta
Canada

Abstract

General Game Playing Agents often play far more poorly than their game-specific counterparts due to the overhead of repeatedly querying an evolving logic program. A natural alternative approach is to instead maintain a grounded logic program and update it as the game state changes. This paper presents a simple algorithm for updating a stratified logic program to reflect changes to the game state and shows for Connect-4 that using logic program updates as opposed to recomputing from scratch at each visited game-state constitutes a big-O improvement in the running time of checking for 4-in-a-row.

1 Introduction

GDL (Game Description Language[1]) is a minimalistic declarative language for describing games. Many games have large and complex descriptions in GDL. For this reason, General Game Playing Agents often play far more poorly than their game-specific counterparts due to the overhead of constantly querying an evolving logic program. Although GDL requires that game descriptions be stratified, and there exist linear-time algorithms for querying stratified logic programs, the games are often complex enough to make even linear algorithms prohibitively slow. One natural alternative approach is to instead maintain a grounded logic program and update the program as the game state changes. This paper presents a simple algorithm for updating a stratified logic program to reflect changes to the game state and shows for Connect-4 that using logic program updates as opposed to recomputing from scratch at each visited game-state constitutes a big-O improvement in the running time of checking for 4-in-a-row.

Section 2 provides some definitions are provided for talking about a grounded logic program. It is assumed that the provided program is already grounded. The approach described in this paper only applies to grounded programs so a game whose grounding is too large to fit in memory cannot be played using the techniques described in this paper. Section 3 defines the concept of a *numeric model* of a logic program which is crucial to understanding the algorithm outlined in this paper. Section 4 provides the pseudocode for an approach to updating a stratified logic program and proves its correctness and running time. Section 5 shows, as an example, why this technique is expected to exhibit such impressive results for Connect 4. Section 6 describes some optimizations that were used to speed up performance in an actual implementation of this technique. Section 7 shows some experimental results for a couple common games (including Connect 4).

[1] As defined in [Love *et al.*, 2008]

2 Some Definitions

Stratified Program

In [Apt *et al.*, 1988], a stratified logic program is defined as:

> A program is *stratified* if there is a partition $P = P_1 \dot\cup \dots \dot\cup P_n$
> such that the following two conditions hold for $i = 1, \dots, n$:
>
> 1. If a relation symbol occurs positively [as a positive literal] within a clause P_i, then its definition is contained within $\bigcup_{j \le i} P_j$
> 2. If a relation symbol occurs negatively [as a negative literal] within a clause P_i, then its definition is contained within $\bigcup_{j < i} P_j$
>
> P_1 can be empty
>
> We say that P is *stratified* by $P_1 \dot\cup \dots \dot\cup P_n$ and each P_i is called a *stratum* of P

Given a rule
$r = [h \Longleftarrow \{a_1 \dots a_m\}, \neg \{b_1 \dots b_n\}]$
we can say $h = \mathbf{h}(r)$ to mean h is the head of rule r.
$\mathbf{b}_+(r) = \{a_1 \dots a_m\}$
and
$\mathbf{b}_-(r) = \{b_1 \dots b_n\}$
denote the sets of atoms contained in the body of r as positive and negative literals respectively and their union is:

$\mathbf{b}(r) = \mathbf{b}_+(r) \bigcup \mathbf{b}_-(r)$

If $h = \mathbf{h}(r)$, and $\mathbf{b}_+(r) \subset M$ and $\mathbf{b}_-(r) \bigcap M = \emptyset$, then by r we know that h must be true and we can say r **supports** h in M, written

$M(r) \vdash h$

Dependency Graph

[Apt *et al.*, 1988] also defines the "dependency graph" of a logic program P as:

> The directed graph representing the relation *refers to* between the relation symbols of P. Formally, p *refers to* q in P iff there is a clause C in P where p is the relation symbol in the head of C and q is the relation symbol of a literal in the body of C.

Because this paper is only interested in grounded programs, we will use dependency graph to mean the dependency graph between the atoms of a program as if each atom is its own relation symbol of arity 0.

Atom Types

In a GDL program, the atoms can be divided into two classes. First, there are the *true* and *does* atoms hereafter referred to as "**base**" atoms. These do not appear in the head of any rule and are instead determined by the game state and players' moves respectively. All the remaining atoms are "**view**" atoms. These atoms' values are determined by a stratified logic program which describes the game. This program must be solved in order to ascertain a player's legal moves, to tell whether the game is in a terminal position, to compute child positions, and to determine the winner of the game.

Game State

A **game state** S is a subset of the base atoms in a game G which are taken to be "true" in this game state.

For the purposes of this paper, a game state includes assignments to *does* atoms as well as *true*.

Game State Program

Given a game description G as a set of rules and a game state S, we define the **game state program** $P_G(S)$ as $G \cup S$ (ie each atom $a \in S$ is a fact in $P_G(S)$).

Canonical Stratification

In Lemma 1 of [Apt *et al.*, 1988], they show by construction that a program is stratified if the dependency graph for that program contains no negative edges as part of a cycle.

This is done by taking the strongly connected components $P_1 \ldots P_n$ of the dependency graph of P and ordering them topologically. They then go on to show that this partitioning $P_1 \dot\cup \ldots \dot\cup P_n$ constitutes a stratification of P.

For a given game G, let $G_1 \ldots G_n$ be the stratification of G obtained in this way. We will refer to this as the **canonical stratification** of G. Furthermore, given a game-state program $P_G(S)$ for any game-state S, it is easy to see that $P_1 = S$ and $P_2 \ldots P_{n+1} = G_1 \ldots G_n$ constitutes a stratification of $P_G(S)$ which we can call the canonical stratification of $P_G(S)$.

Because $P_2 \ldots P_{n+1}$ is always the same regardless of the game state, we can assign to each rule $r \in G$ a **level**

$l(r) = k | r \in P_k$

In other words, the level of r is the stratum in which r occurs in the canonical stratification of $P_G(S)$ for any state S.

In addition to giving each rule a level, it will also be useful to assign a level to each atom.

The **level** of any view atom a is the maximum level over all rules which have a as their head:

$l(a) = \max_{a=\mathbf{h}(r)} l(r)$

The level of any base atom is just 1

$l(b) = 1$

Supported Model

A **supported model** M of a game state program $P_G(S)$ is a set of atoms M such that $S \subset M$ and for each view atom $h \in G$, there is a rule r that supports h in M. (ie $\exists r\, M(r) \vdash h$)

Stable Model

A **stable model** or **minimal model** of S is a supported model M that is minimal in the sense that no subset of M is a supported model of S.

As [Apt *et al.*, 1988] show, the unique minimal model M_P of a program P stratified by $P = P_1 \dot\cup \ldots \dot\cup P_n$ is

$M_P = M_n$

where

$M_1 = T_{P_1} \uparrow \omega(\emptyset)$

and

$\forall k,\, M_k = T_{P_k} \uparrow \omega(M_{k-1})$.

Here T_P is the immediate consequence operator meaning for some set of facts M,

$T_P(M) = \{h | \exists r \in P : M(r) \vdash h\}$

and $T_P \uparrow \omega$ is defined by

- $T_P \uparrow 0(M) = M$
- $T_P \uparrow (n+1)(M) = T_P(T_P \uparrow n(M)) \cup T_P \uparrow n(M)$
- $T_P \uparrow \omega(M) = \bigcup_{n=0}^{\infty} T_P \uparrow n(M)$

Since P is finite and we're only interested in grounded programs, $T_P \uparrow \omega(M)$ can be obtained by repeatedly adding all consequences of $P \cup M$ to M until M converges.

3 An Ordering on Rules and Atoms

3.1 Finding Levels

We start by finding the level (as defined in the above section) of each rule and each atom.

Since the level of any rule or atom does not depend on the game state, this can be done in preprocessing. We simply build the dependency graph of G and then topologically sort the rules. Then we identify the strongly connect components and index them according to the order of first appearance in the topologically sorted list.

```
function assignLevels(rules)
        rules = topoSort(rules)
        SCCs = identifySCCs(rules)
        i=0
        for rule in rules {
                if rule.scc.level==None {
                        rule.scc.level = i
                        i = i + 1
                }
                rule.level = rule.scc.level
        }
endfunction
```

3.2 Numeric Model

Of a Positive Program

Given a positive program P with stable solution $M_P = T_P \uparrow \omega(\emptyset)$, we define a numeric model of P as follows. For each atom a

$$v(a) = \begin{cases} \min\{n \geq 1 | a \in T_P \uparrow n(\emptyset)\} & a \in M_P \\ 0 & a \notin M_P \end{cases}$$

In other words, the value of any atom which is true in M_P is the number of iterations of applying T_P (starting from the empty set) it takes to discover $a \in M_P$

For each rule r with head $h = \mathbf{h}(r)$, we define the value of r to be $v(r) = \begin{cases} \max\{v(a) | a \in \mathbf{b}(r)\} & M_P(r) \vdash h \\ 0 & M_P(r) \nvdash h \end{cases}$

In other words, the value of any rule which is "activated" in M_P is the max value over all the atoms in its body while the value of any "deactivated" rule is 0.

An assignment of integer values to each atom and each rule in P constitue a **numeric model** of P. Intuitively, a numeric model tells how many steps it takes to prove that an atom a belongs in the minimal model M_P.

Of a Stratified Program

A numeric model of a stratified program P assigns values to each rule and each atom according to the numeric model of each of the semi-positive strata $P_1 \ldots P_n$ of P. An atom a is said to belong to P_k if its level $l(a) = k$.

In other words, for an atom a with $l(a) = k$ and a minimal model $\bar{M}_{k-1}$ of $\bar{P}_{k-1} = P_1 \cup \ldots \cup P_{k-1}$, the value of a is

$$v(a) = \begin{cases} \min\left\{n \geq 1 | a \in T_{P_k} \uparrow n\left(\bar{M}_{k-1}\right)\right\} & a \in M_P \\ 0 & a \notin M_P \end{cases}.$$

This definition of the value of an atom a within some stratum P_k of a stratified program is almost identical to the definition for a positive program, except that instead of starting from the empty set, we start from the minimal model $\bar{M}_{k-1}$ of all the strata $P_1 \cup \ldots \cup P_{k-1}$ prior to P_k.

Theorem 1

For any activated rule r with head $h = \mathbf{h}(r)$, $l(r) < l(h)$ implies $v(h) = 1$.

Proof

Since $l(r) < k$, we know that $\mathbf{b}_+(r) \subseteq \bar{M}_{k-1}$

Since r is activated, $\mathbf{b}_-(r) \cap M_P = \emptyset$ and so by $M_{k-1} \subset M_P$, this shows that $\mathbf{b}_-(r) \cap M_{k-1} = \emptyset$

So from this we can see that $a \in T_{P_k}(M_{k-1}) \subseteq T_{P_k} \uparrow 1(M_{k-1})$

□

The value of a rule r with head $h = \mathbf{h}(r)$ is defined almost exactly as for a positive program, except that we're only concerned with body atoms with the same level as r.

$$v(r) = \begin{cases} \max\{v(a) | a \in \mathbf{b}_+(r) \wedge l(a) = l(r)\} & M_P(r) \vdash h \\ 0 & M_P(r) \nvdash h \end{cases}$$

(where $\max(\emptyset) = 1$)

Theorem 2

Let $v_k(r) = \begin{cases} v(r) & k = l(r) \\ 0 & k \neq l(r) \end{cases}$

In a stratified program P with minimal model M_P, the value of any atom $a \in M_P$ is

$v(a) = 1 + \min\left\{v_{l(a)}(r) | M(r) \vdash a\right\}$

In other words $v(a)$ is one more than the minimum value over all potential supporting rules with the same level as a or just 1 if there is a supporting rule from a lower level.

Proof:

As was shown in Theorem 1, if an atom a has a true supporting rule r such that $l(r) < l(a)$, then $v(a) = 1$.

Let us therefore assume the only supporting rules have the same level as a, ie $l(r) = l(a) \forall M(r) \vdash a$.

Let $M = \bar{M}_{k-1}$. For an atom a with $l(a) = k$, if $v(a) = n$, then we know that $a \in T_{P_k} \uparrow n(M)$ and furthermore by the minimality of v we know that $a \notin T_{P_k} \uparrow (n-1)\left(\bar{M}_{k-1}\right)$.

Since $T_{P_k} \uparrow n(M) = T_{P_k}(T_{P_k} \uparrow (n-1)(M)) \cup T_{P_k} \uparrow (n-1)(M)$ it must be the case that $a \in T_{P_k}(T_{P_k} \uparrow (n-1)(M))$

This means there exists a rule $r \in P_k$ for which $T_{P_k} \uparrow (n-1)(M)(r) \vdash a$, but there does not exist a rule r' for which $T_{P_k} \uparrow (n-2)(M)(r') \vdash a$. Then it follows that r has level $n-1$ and there is no rule supporting a whose level is $< n-1$. Then $n-1 = \min\{v(r) | M(r) \vdash a\}$. Since we assumed $l(a) = l(r) \forall M(r) \vdash a$, we can therefore say $v(a) = n = 1 + \min\left\{v_{l(a)}(r) | M(r) \vdash a\right\}$

□

3.3 An Ordering on Rules and Atoms

Between levels and values, we now have a partial ordering on all the rules and atoms in a stratified program.

Let us say that the atoms and rules in a program are ordered lexicographically first by level and then by value:

$a \preceq b$ iff $l(a) < l(b)$ or $l(a) = l(b) \wedge v(a) \leq v(b)$

If we sort all the atoms and rules according to this ordering (where atoms come before rules of the same level and value), this gives a possible order in which a stable model of P can be constructed.

4 The Algorithm

The algorithm I propose below maintains and updates a numeric model as changes are propogated up from the base atoms. This guarantees that the maintained model is always stable.

Note that although the pairs inserted into the priority queue have both a rule and a value, the value is only used for ordering the priority queue. When the rule is polled from the queue, it's value is recomputed and may not be the same as the value that was inserted into the queue.

```
function propogate(changedBaseAtoms) {
  for atom in changedBaseAtoms
    for rule in atom.inBody
      pqueue.add((rule=rule, newVal=0))
  while pqueue.hasNext()
    propogateRule(pqueue.poll())
}

function compare(p1, p2) {
  //p1 and p2 are (rule, newVal) pairs
  if p1.rule.level != p2.rule.level
    return compareNum(p1.rule.level,
        p2.rule.level)
  else
    return compareNum(p1.newVal,
        p2.newVal)
}

pqueue = priorityqueue(
    comparefunction = compare)

function propogateAtom(atom) {
  newval = 0
  for rule in atom.headof {
    if rule.value > 0 {
      if rule.level == atom.level
        nv = rule.value + 1
      else
        nv = 1
      if newval == 0 or nv < newval
        newval = nv
    }
  }
  if (atom.value > 0
      and newval > atom.value) {
    for support in atom.headof {
      if support.value > 0 {
        pqueue.add(support, support.value)
        support.value = 0
      }
    }
    newval = 0
  }
  if newval != atom.value {
    atom.value = newval
    nbval = min(newval, 1)
    for rule in atom.inPosBody {
      if rule.level == atom.level
        pqueue.add((rule = rule,
            newval = newval))
      else
        pqueue.add((rule = rule,
            newval = nbval))
    }
    for rule in atom.inNegBody
      pqueue.add((rule = rule,
          newval = 1 - nbval))
  }
}

function propogateRule(rule) {
  newval = 1
  for atom in rule.posBody {
    if atom.value == 0 {
      newval = 0
    } else if (atom.value > newval
        and newval > 0
        and atom.level == rule.level) {
      newval = atom.value
    }
  }
  for atom in rule.negBody {
    if atom.value > 0
      newval = 0
  }
  if newval != rule.value {
    rule.value = newval
    propogateAtom(rule.head)
  }
}
```

Theorem 3
If the propogate method terminates, the resulting values form a stable numeric model:

Proof
This can be seen as the result of a couple observations:

- Every time any value changes, the result is propogated to all the atoms which might possibly be affected by that change.
- Every time the propogateRule method is called, when the method returns, the rule has the correct value locally (ie assuming each of the supporting atoms have the correct value).
- With one exception (to be addressed), every time the propogateAtom method returns, the atom has the correct value locally (ie assuming each of the supporting rules have the correct value).

The one exceptional case occurs when a currently non-zero atom would increase in value. Instead the atom and all the rules supporting the increase all have their values set to zero and the rules are thrown back into the queue. Because the rules are thrown back onto the queue, we know that they will all necessarily have their values recomputed.

The atom will also be recomputed unless all supporting rules have new value zero in which case it's already correct.

This means that for every atom which might potentially change value, propogateAtom is called, and the

last time propogateAtom is called on any atom, it will have the correct value.

□

Although the above proof shows that the algorithm can terminate only with the correct value, it makes no guarantees as to the running time or even that the algorithm will ever terminate. To prove this we need the following theorem.

Theorem 4

Calling propogate cannot cause any atom to change value more than twice.

Proof:

Inductively, this follows from the following three properties:

1. For any atom or rule q whose current value $v(q) > 0$ is already positive, after calling propogate, the new value will be $v'(q) \leq v(q)$.
2. When any atom or rule q changes value, the new value it takes on is always either 0 or the actual correct value $v(q)$.
3. When propogating a (rule, newval) pair p_1, for any (rule, newval) pair p_2 that gets added to the queue, $p_1 \preceq p_2$.

1. is clearly true for atoms because the code includes a conditional statement saying "If the value of this atom would increase, instead set it and all its supporting rules to zero and add them back to the queue"

To see that 1. holds for rules, note that if $v = \max(a_1 \ldots a_n)$, then if some change to/removal of one a_k increases v, it must be that a_k increases, but since atom values do not directly increase, this cannot happen.

To see that 2. and 3. are true for some atom a, apply the inductive hypothesis that 2. and 3. hold for all rules r for whom $r \prec a$.

Then it follows that any rule supporting $M(r) \vdash a$ must have its correct value already. And since we are guaranteed by 3. to hit a in order by $\prec$, all supporting rules for whom $v(r) < v(a)$ must already have their correct values. All other supporting rules will have value 0 or else be unchanged from their starting value if that value is $> v(a)$. In either case, the minimum of the nonzero supporting rules will be $v(a) - 1$ so a will take on its correct value $v(a)$.

It only remains to show that the special case where a would increase, but instead all its supporting rules are set to 0 occurs only when the pair on the queue is of the form $(r, 0)$. This will prove that 3 always holds.

To see this, keep in mind that because of 1., no rule can increase in value, so the way in which an atom increases in value is that all its minimal supporting rules are set to false and the atom falls back on a larger-valued supporting rule. When this happens, the propogated value is the one coming from the minimal supporting rules which are no longer true so that value must be 0.

□

5 Example: Connect 4 Win-Checking

Consider the game Connect 4. Any GGP agent must query the board at every game state to determine whether the state contains a winning 4-in-a-row connection or not. Typically, such a connection would be GDL-encoded to look something like:

```
(dir N 1 0) (dir E 0 1)
(dir NE 1 1) (dir SE -1 1)

(<= (inarow 1 ?player ?row ?col ?dir)
 (true (cell ?row ?col ?player))
 (role ?player)
 (dir ?dir ?dr ?dc))

(<= (inarow ?n ?player ?row ?col ?dir)
 (true (cell ?row ?col ?player))
 (inarow ?nlast ?player ?r1 ?c1 ?dir)
 (dir ?dir ?dr ?dc)
 (eq ?row (sum ?r1 ?dr))
 (eq ?col (sum ?c1 ?dc))
 (lessoreq ?n 4)
 (eq ?n (sum ?nlast 1)))

(<= (winner ?player)
 (inarow 4 ?player ?row ?col ?dir))
```

Now consider how much work is involved in determining the truth of the grounded atom (winner x) where the players are x and o.

For each row ?row, for each column ?col for each direction ?dir, we must check the value of (inarow 1 x ?row ?col ?dir).

Now, for all cells where this atom is true, we must check neighboring cells to determine the truth of (inarow 2 x ?row ?col ?dir).

For a connect-4 board whose size is WxH, the big-O running time of this check is $O(WH)$.

But if we use incremental propogation, after each move is made, only one cell has changed value, so only that one cell needs to be checked for a 4-in-a-row. This operation is always constant time $O(1)$ regardless of the board size.

6 Optimizations

In this section, we provide some optimizations that can be used to speed up the performance of the algorithm:

6.1 Rule Level $+\frac{1}{2}$

Whenever $l(r) < l(\mathbf{h}(r))$, add $\frac{1}{2}$ to the level $l(r)$. This way, the algorithm doesn't bother computing the value of r until after all it's body atoms have the correct value so there's no need to update r more than once.

6.2 Primary Supporting Rule Tracker

For each true atom with only supporting rules on the same level, keep a pointer to one supporting rule whose value is $v(r) = v(a) - 1$. Now whenever another rule besides r changes value, if its new value is 0 or $\geq v(a) - 1$, we know that the value $v(a)$ will not change as a

result, so we don't need to recompute $v(a)$. If r' has a new value which is $< v(a)$ but still > 0, then we know without having to look at all potential supporting rules that a's new value will be $v_{\text{new}}(a) = v(r) + 1$. This is analogous to DPLL's *two watched literals* approach to Boolean Constraint Propogation as presented in section 2 of [Moskewicz *et al.*, 2001]. The analogy is made almost exact by considering the LP-implied expression

$a \to [r_1] \vee \ldots \vee [r_n]$

where $\{r_1 \ldots r_n\}$ is the set of rules that have a as their head and $[r]$ indicates an implied atom with a single supporting rule $[r] \leftarrow [\mathbf{b}_+(r), \neg\mathbf{b}_-(r)]$.

This expression can be rewritten in disjunctive normal form as:

$\neg a \vee [r_1] \vee \ldots \vee [r_n]$

Now our two watched literals are $\neg a$ and r_k where r_k is the chosen supporting rule. So long as r_k does not change value, we need not worry about a changing value.

6.3 Priority Queue Optimizations

In practice the bottleneck of this algorithm is priority queue operations. For this reason, it is beneficial to use an application-specific priority queue. The following two optimizations speed up the program quite a bit:

No Duplicates

In addition to maintining a priority queue with all (rule, newval) pairs. We also keep a hash set. Before adding a pair to the queue, we look it up in the hash set to see if it's already been added. If so, we don't bother.

Bucket Queue

The priority queue is maintained as a collection of buckets (one for each level in the program). Each bucket itself contains its own priority queue. In this way, the usual $O(\log n)$ insertion time for a heap queue is reduced to $O(\log k)$ where k is the number of pairs in the queue at the level to be inserted.

It's even useful to keep a special bucket at each level for pairs with value 0 or 1 and add only pairs with higher values to the priority queue. In this way only the atoms which are part of head cycles suffer from the penalty of inserting elements into a heap. For tight games such as Othello and Connect 4 (but not Hex), this removes the need for a priority queue entirely.

6.4 Supporting Atom Count Tracker

To reduce the number of rules that must be added to the priority queue, each rule r can keep a count c_r of the number of body literals which are satisfied. This count is updated not when the rule's propogate method is called, but rather immediately whenever any body atom changes value. Since every atom must be satisfied in order to activate the rule, we do not need to bother adding the rule to our priority queue unless every literal is satisfied after the change or was satisfied before the change (this can be checked in constant time by seeing whether $c_r = |\mathbf{b}(r)|$).

Furthermore, as with the "supporting rule tracker", for activated rules we can track the body atom whose value

	Early Game	Midgame	End game
Tic Tac Toe	48.3×10^3	-*	-*
Connect 4 (7x6)	29.3×10^3	33.7×10^3	37.3×10^3
Hex (7x7)	7.19×10^3	9.48×10^3	12.7×10^3
Y (36 cells)	10.3×10^3	13.1×10^3	17.4×10^3
Othello (8x8)	0.808×10^3	1.13×10^3	1.94×10^3

Table 1: Average Playouts / 5 sec on some standard games
* The agent solves the game too quickly to get significant meaningful statistics

is maximal. If any other atom changes value (positive to positive) then it cannot affect the value of this rule (since positive changes can only decrease as was proved in Theorem 4) so we only need to know when our supporting atom changes value.

6.5 Lazy As-Needed Propogation

The different things we might wish to query about our program are:

- What are my legal moves (legal ?player ?move)
- What is the next game state (next ?fact)
- Is the game over (terminal)
- What's the score (goal ?player ?v)

Starting with these four possible queries, during preprocessing we can backtrack over the dependency graph to determine what use each atom might potentially have.

For instance, in Connect 4, the atom

```
(inarow 2 x 4 4 N)
```

is only of interest in determining the score of the game. If we already know the game state is not terminal, then we may not need to query this information.

For this reason, we can maintain four separate priority queues (one for each of the different potential queries). Then when each rule is propogated, it is only added to the priority queues corresponding to that rule's uses.

When we query for one of these four things, we first propogate the rules only on the corresponding priority queue and ignore the other queues. The overhead of inserting each change into up to four queues as opposed to one appears to be not as serious as of propogating values that are of no immediate interest.

7 Results

Table 1 lists the performance of a simple Java MCTS player. For Tic Tac Toe and Connect 4, the performance is significantly (in the early game almost an order of magnitude) faster than in [Waugh, 2009].

8 Future Work

Although this paper dealt exclusively with atoms and with grounded programs, the concept of levels and strata extend naturally to predicate logic. There may be merit

in finding ways to construct the true atoms and find their values on the fly and assign levels based on the corresponding predicate to each atom. This could be particularly useful for programs where a predicate has a large arity, but only a small subset of the grounded instances of that predicate are ever true simultaneously. It would be interesting to optimize this approach using generated C++ code as in [Waugh, 2009]. Additionally, it would be useful to have an upper bound on the ratio $\frac{\#\text{ atom values changed in numeric model}}{\#\text{ atom values changed in stable model}}$. Of course for tight programs, this is just 1, but many games, such as Hex, Y, and Go, require loopy programs. In these cases it would be interesting to know exactly how much "extra" work is necessary to maintain that information. Perhaps propogation would run in faster amortized time if we loosen the restriction on values which atoms and rules can take on to say $v(a)$ is greater than the value of some supporting rule and $v(r)$ is greater than or equal to the value of all same-level supporting atoms.

9 Conclusion

By recognizing which aspects of the game change with respect to each move, we can eliminate much of the repeated work done for querying game states. Incorporating techniques from logic program updates is a promising path for closing the performance gap between general game playing agents and their game-specific counterparts.

Acknowledgments

Thank you to the University of Alberta GAMES Group. Also special thanks to Jia You and Ryan Hayward for all their help with this work.

References

[Apt *et al.*, 1988] Krzysztof R Apt, Howard A Blair, and Adrian Walker. Towards a theory of declarative knowledge. *Foundations of deductive databases and logic programming*, pages 89–148, 1988.

[Love *et al.*, 2008] Nathaniel Love, Timothy Hinrichs, David Haley, Eric Schkufza, and Michael Genesereth. General game playing: Game description language specification, 2008.

[Moskewicz *et al.*, 2001] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.

[Waugh, 2009] Kevin Waugh. Faster state manipulation in general games using generated code. *Proceedings of the 1st general intelligence in game-playing agents (GIGA)*, 2009.