

Yngvi Björnsson
Nathan Sturtevant
Michael Thielscher (Eds.)

The IJCAI-11 Workshop on General Game Playing

General Intelligence in Game-Playing Agents, GIGA'11

Barcelona, Spain, July 2011
Proceedings

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,
- reasoning,
- heuristic search,
- automated planning,
- computational game-theory,
- multi-agent systems,
- machine learning,
- game design,
- applications

These are the proceedings of GIGA'11, the second ever workshop on General Intelligence in Game-Playing Agents following the inaugural GIGA Workshop at IJCAI'09 in Pasadena (USA). 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 12 papers that have been selected for presentation at this workshop. This is exactly the same number as at the previous GIGA'09 Workshop, indicating the continuing interest in General Game Playing as an AI Grand Challenge Problem. All submissions were reviewed by a distinguished international program committee. The accepted papers cover a multitude of topics such as the automatic analysis and transformation of game descriptions, simulation-based methods, learning techniques, heuristics, and extensions of the general game description language.

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!

July 2011

Yngvi Björnsson
Nathan Sturtevant
Michael Thielscher

Organization

Workshop Chairs

Yngvi Björnsson, Reykjavík University, Iceland

Nathan Sturtevant, University of Denver, USA

Michael Thielscher, The University of New South Wales, Australia

Program Committee

Yngvi	Björnsson	Reykjavík University
Tristan	Cazenave	LAMSADE - Université Paris-Dauphine
Stefan	Edelkamp	University of Bremen
Gregory	Kuhlmann	21st Century Technologies
Torsten	Schaub	University of Potsdam
Nathan	Sturtevant	University of Denver
Michael	Thielscher	The University of New South Wales
Mark	Winands	Maastricht University
Michael	Wooldridge	University of Liverpool

Additional Reviewers

Hilmar	Finnsson
Peter	Kissmann
Hootan	Nakhost
Abdallah	Saffidine
Marius	Schneider
Richard	Valenzano

Table of Contents

Distance Features for General Game Playing	7
<i>Daniel Michulke, Stephan Schiffel</i>	
On the Complexity of BDDs for State Space Search: A Case Study in Connect Four	15
<i>Stefan Edelkamp, Peter Kissmann</i>	
Game-Tree Properties and MCTS Performance	23
<i>Hilmar Finnsson, Yngvi Björnsson</i>	
Neural Networks for High-Resolution State Evaluation in General Game Playing	31
<i>Daniel Michulke</i>	
Tree parallelization of Ary on a cluster	39
<i>Jean Méhat, Tristan Cazenave</i>	
MCTS: Improved Action Selection Techniques for Deterministic Games	45
<i>Stefan Freyr Gudmundsson, Yngvi Björnsson</i>	
On the Comparative Expressiveness of Epistemic Models and GDL-II	53
<i>Ji Ruan, Michael Thielscher</i>	
Heuristic Interpretation of Predicate Logic Expressions in General Game Playing	61
<i>Daniel Michulke</i>	
A Forward Chaining Based Game Description Language Compiler	69
<i>Abdallah Saffidine, Tristan Cazenave</i>	
Extending the General Game Playing Framework to Other Languages	77
<i>Xinxin Sheng, David Thente</i>	
First-Order Logic with Counting for General Game Playing	85
<i>Lukasz Kaiser, Łukasz Stafiniak</i>	
Translating the Game Description Language to Toss	91
<i>Lukasz Kaiser, Łukasz Stafiniak</i>	

Distance Features for General Game Playing

Daniel Michulke

Department of Computer Science
Dresden University of Technology
daniel.michulke@mailbox.tu-dresden.de

Stephan Schiffel

School of Computer Science
Reykjavík University
stephans@ru.is

Abstract

General Game Playing (GGP) is concerned with the development of programs that are able to play previously unknown games well. The main problem such a player is faced with is to come up with a good heuristic evaluation function automatically. Part of these heuristics are distance measures used to estimate, e.g., the distance of a pawn towards the promotion rank. However, current distance heuristics in GGP are based on too specific detection patterns as well as expensive internal simulations, they are limited to the scope of totally ordered domains and/or they apply a uniform Manhattan distance heuristics regardless of the move pattern of the object involved.

In this paper we describe a method to automatically construct distance measures by analyzing the game rules. The presented method is an improvement to all previously presented distance estimation methods, because it is not limited to specific structures, such as, game boards. Furthermore, the constructed distance measures are admissible.

We demonstrate how to use the distance measures in an evaluation function of a general game player and show the effectiveness of our approach by comparing with a state-of-the-art player.

1 Introduction

While in classical game playing, human experts encode their knowledge into features and parameters of evaluation functions (e.g., weights), the goal of General Game Playing is to develop programs that are able to autonomously derive a good evaluation function for a game given only the rules of the game. Because the games are unknown beforehand, the main problem lies in the detection and construction of useful features and heuristics for guiding search in the match.

One class of such features are distance features used in a variety of GGP agents (e.g., (Kuhlmann, Dresner, and Stone 2006; Schiffel and Thielscher 2007; Clune 2007; Kaiser 2007)). The way of detecting and constructing features in current game playing systems, however, suffers from a variety of disadvantages:

- Distance features require a prior recognition of board-like game elements. Current approaches formulate hypotheses about which element of the game rules describes a board and then either check these hypotheses in internal simulations of the game (e.g., (Kuhlmann, Dresner, and Stone

2006; Schiffel and Thielscher 2007; Kaiser 2007)) or try to prove them (Schiffel and Thielscher 2009a). Both approaches are expensive and can only detect boards if their description follows a certain syntactic pattern.

- Distance features are limited to board-like structures, that is, n -dimensional structures with totally ordered coordinates. Distances over general graphs are not considered.
- Distances are calculated using a predefined metric on the boards. Consequently, distance values obtained do not depend on the type of piece involved. For example, using a predefined metric the distance of a rook, king and pawn from $a2$ to $c2$ would appear equal while a human would identify the distance as 1, 2 and ∞ (unreachable), respectively.

In this paper we will present a more general approach for the construction of distance features for general games. The underlying idea is to analyze the rules of game in order to find dependencies between the fluents of the game, i.e., between the atomic properties of the game states. Based on these dependencies, we define a distance function that computes an admissible estimate for the number of steps required to make a certain fluent true. This distance function can be used as a feature in search heuristics of GGP agents. In contrast to previous approaches, our approach does not depend on syntactic patterns and involves no internal simulation or detection of any predefined game elements. Moreover, it is not limited to board-like structures but can be used for every fluent of a game.

The remainder of this paper is structured as follows: In the next section we give an introduction to the Game Description Language (GDL), which is used to describe general games. Furthermore, we briefly present the methods currently applied for distance feature detection and distance estimation in the field of General Game Playing. In Section 3 we introduce the theoretical basis for this work, so called fluent graphs, and show how to use them to derive distances from states to fluents. We proceed in Section 4 by showing how fluent graphs can be constructed from a game description and demonstrate their application in Section 5. Finally, we conduct experiments in Section 6 to show the benefit and generality of our approach and discuss and summarize the results in Section 9.

2 Preliminaries

The language used for describing the rules of general games is the Game Description Language (Love et al. 2008) (GDL). GDL is an extension of Datalog with functions, equality, some syntactical restrictions to preserve finiteness, and some predefined keywords.

The following is a partial encoding of a Tic-Tac-Toe game in GDL. In this paper we use Prolog syntax where words starting with upper-case letters stand for variables and the remaining words are constants.

```

1 role(xplayer). role(oplayer).
2
3 init(cell(1,1,b)). init(cell(1,2,b)).
4 init(cell(1,3,b)). ...
5 init(cell(1,3,b)). init(control(xplayer)).
6
7 legal(P, mark(X, Y)) :-
8   true(control(P)), true(cell(X, Y, b)).
9 legal(P, noop) :-
10  role(P), not true(control(P)).
11
12 next(cell(X,Y,x)) :- does(xplayer,mark(X,Y)).
13 next(cell(X,Y,o)) :- does(oplayer,mark(X,Y)).
14 next(cell(X,Y,C)) :-
15   true(cell(X,Y,C)), distinct(C, b).
16 next(cell(X,Y,b)) :- true(cell(X,Y,b)),
17   does(P, mark(M, N)),
18   (distinct(X, M) ; distinct(Y, N)).
19
20 goal(xplayer, 100) :- line(x).
21 ...
22 terminal :- line(x) ; line(o) ; not open.
23
24 line(P) :- true(cell(X, 1, P)),
25   true(cell(X, 2, P)), true(cell(X, 3, P)).
26 ...
27 open :- true(cell(X, Y, b)).

```

The first line declares the roles of the game. The unary predicate **init** defines the properties that are true in the initial state. Lines 7-10 define the legal moves of the game with the help of the keyword **legal**. For example, **mark**(X,Y) is a legal move for role P if **control**(P) is true in the current state (i.e., it's P's turn) and the cell X,Y is blank (**cell**(X,Y,b)). The rules for predicate **next** define the properties that hold in the successor state, e.g., **cell**(M,N,x) holds if xplayer marked the cell M,N and **cell**(M,N,b) does not change if some cell different from M,N was marked¹. Lines 20 to 22 define the rewards of the players and the condition for terminal states. The rules for both contain auxiliary predicates **line**(P) and **open** which encode the concept of a line-of-three and the existence of a blank cell, respectively.

We will refer to the arguments of the GDL keywords **init**, **true** and **next** as fluents. In the above example, there are two different types of fluents, **control**(X) with $X \in \{xplayer, oplayer\}$ and **cell**(X, Y, Z) with $X, Y \in \{1, 2, 3\}$ and $Z \in \{b, x, o\}$.

¹The special predicate **distinct**(X, Y) holds if the terms X and Y are syntactically different.

In (Schiffel and Thielscher 2009b), we defined a formal semantics of a game described in GDL as a state transition system:

Definition 1 (Game). Let Σ be a set of ground terms and 2^Σ denote the set of finite subsets of Σ . A game over this set of ground terms Σ is a state transition system $\Gamma = (R, s_0, T, l, u, g)$ over sets of states $\mathcal{S} \subseteq 2^\Sigma$ and actions $\mathcal{A} \subseteq \Sigma$ with

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

This formal semantics is based on a set of ground terms Σ . This set is the set of all ground terms over the signature of the game description. Hence, fluents, actions and roles of the game are ground terms in Σ . States are finite sets of fluents, i.e., finite subsets of Σ . The connection between a game description D and the game Γ it describes is established using the standard model of the logic program D . For example, the update function u is defined as

$$u(A, s) = \{f \in \Sigma : D \cup s^{\text{true}} \cup A^{\text{does}} \models \text{next}(f)\}$$

where s^{true} and A^{does} are suitable encodings of the state s and the joint action A of all players as a logic program. Thus, the successor state $u(A, s)$ is the set of all ground terms (fluents) f such that **next**(f) is entailed by the game description D together with the state s and the joint move A . For a complete definition for all components of the game Γ we refer to (Schiffel and Thielscher 2009b).

3 Fluent Graphs

Our goal is to obtain knowledge on how fluents evolve over time. We start by building a *fluent graph* that contains all the fluents of a game as nodes. Then we add directed edges (f_i, f) if at least one of the predecessor fluents f_i must hold in the current state for the fluent f to hold in the successor state. Figure 1 shows a partial fluent graph for Tic-Tac-Toe that relates the fluents **cell**(3,1,Z) for $Z \in \{b, x, o\}$.

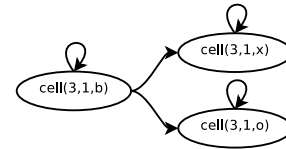


Figure 1: Partial fluent graph for Tic-Tac-Toe.

For cell (3,1) to be blank it had to be blank before. For a cell to contain an x (or an o) in the successor state there are two possible preconditions. Either, it contained an x (or o) before or it was blank.

Using this graph, we can conclude that, e.g., a transition from **cell**(3,1,b) to **cell**(3,1,x) is possible

within one step while a transition from $\text{cell}(3,1,o)$ to $\text{cell}(3,1,x)$ is impossible.

To build on this information, we formally define a fluent graph as follows:

Definition 2 (Fluent Graph). Let Γ be a game over ground terms Σ . A graph $G = (V, E)$ is called a fluent graph for Γ iff

- $V = \Sigma \cup \{\emptyset\}$ and
- for all fluents $f \in \Sigma$, two valid states s and s'

$$(s' \text{ is a successor of } s) \wedge f' \in s' \quad (1)$$

$$\Rightarrow (\exists f)(f, f') \in E \wedge (f \in s \cup \{\emptyset\})$$

In this definition we add an additional node $\emptyset$ to the graph and allow $\emptyset$ to occur as the source of edges. The reason is that there can be fluents in the game that do not have any preconditions, for example the fluent g with the following next rule: $\text{next}(g) :- \text{distinct}(a, b)$. On the other hand, there might be fluents that cannot occur in any state, because the body of the corresponding next rule is unsatisfiable, for example: $\text{next}(h) :- \text{distinct}(a, a)$. We distinguish between fluents that have no precondition (such as g) and fluents that are unreachable (such as h) by connecting the former to the node $\emptyset$ while unreachable fluents have no edge in the fluent graph.

Note that the definition covers only some of the necessary preconditions for fluents, therefore fluent graphs are not unique as Figure 2 shows. We will address this problem later.

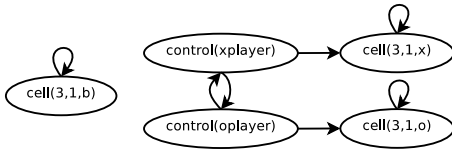


Figure 2: Alternative partial fluent graph for Tic-Tac-Toe.

We can now define a distance function $\Delta(s, f')$ between the current state s and a state in which fluent f' holds as follows:

Definition 3 (Distance Function). Let $\Delta_G(f, f')$ be the length of the shortest path from node f to node f' in the fluent graph G or ∞ if there is no such path. Then

$$\Delta(s, f') = \min_{f \in s \cup \{\emptyset\}} \Delta_G(f, f')$$

That means, we compute the distance $\Delta(s, f')$ as the shortest path in the fluent graph from any fluent in s to f' .

Intuitively, each edge (f, f') in the fluent graph corresponds to a state transition of the game from a state in which f holds to a state in which f' holds. Thus, the length of a path from f to f' in the fluent graph corresponds to the number of steps in the game between a state containing f to a state containing f' . Of course, the fluent graph is an abstraction of the actual game: many preconditions for the state transitions are ignored. As a consequence, the distance $\Delta(s, f')$ that we compute in this way is a lower bound on

the actual number of steps it takes to go from s to a state in which f' holds. Therefore the distance $\Delta(s, f')$ is an admissible heuristic for reaching f' from a state s .

Theorem 1 (Admissible Distance). Let

- $\Gamma = (R, s_0, T, l, u, g)$ be a game with ground terms Σ and states $\mathcal{S}$,
- $s_1 \in \mathcal{S}$ be a state of Γ ,
- $f \in \Sigma$ be a fluent of Γ , and
- $G = (V, E)$ be a fluent graph for Γ .

Furthermore, let $s_1 \mapsto s_2 \mapsto \dots \mapsto s_{m+1}$ denote a legal sequence of states of Γ , that is, for all i with $0 < i \leq m$ there is a joint action A_i , such that:

$$s_{i+1} = u(A_i, s_i) \wedge (\forall r \in R) l(r, A_i(r), s_i)$$

If $\Delta(s_1, f) = n$, then there is no legal sequence of states $s_1 \mapsto \dots \mapsto s_{m+1}$ with $f \in s_{m+1}$ and $m < n$.

Proof. We prove the theorem by contradiction. Assume that $\Delta(s_1, f) = n$ and there is a legal sequence of states $s_1 \mapsto \dots \mapsto s_{m+1}$ with $f \in s_{m+1}$ and $m < n$. By Definition 2, for every two consecutive states s_i, s_{i+1} of the sequence $s_1 \mapsto \dots \mapsto s_{m+1}$ and for every $f_{i+1} \in s_{i+1}$ there is an edge $(f_i, f_{i+1}) \in E$ such that $f_i \in s_i$ or $f_i = \emptyset$. Therefore, there is a path $f_j, \dots, f_m, f_{m+1}$ in G with $1 \leq j \leq m$ and the following properties:

- $f_i \in s_i$ for all $i = j, \dots, m+1$,
- $f_{m+1} = f$, and
- either $f_j \in s_1$ (e.g., if $j = 1$) or $f_j = \emptyset$.

Thus, the path $f_j, \dots, f_m, f_{m+1}$ has a length of at most m . Consequently, $\Delta(s_1, f) \leq m$ because $f_j \in s_1 \cup \{\emptyset\}$ and $f_{m+1} = f$. However, $\Delta(s_1, f) \leq m$ together with $m < n$ contradicts $\Delta(s_1, f) = n$. $\square$

4 Constructing Fluent Graphs from Rules

We propose an algorithm to construct a fluent graph based on the rules of the game. The transitions of a state s to its successor state s' are encoded fluent-wise via the **next** rules. Consequently, for each $f' \in s'$ there must be at least one rule with the head $\text{next}(f')$. All fluents occurring in the body of these rules are possible sources for an edge to f' in the fluent graph.

For each ground fluent f' of the game:

1. Construct a ground disjunctive normal form ϕ of $\text{next}(f')$, i.e., a formula ϕ such that $\text{next}(f') \supset \phi$.
2. For every disjunct ψ in ϕ :
 - Pick one literal $\text{true}(f)$ from ψ or set $f = \emptyset$ if there is none.
 - Add the edge (f, f') to the fluent graph.

Note, that we only select one literal from each disjunct in ϕ . Since, the distance function $\Delta(s, f')$ obtained from the fluent graph is admissible, the goal is to construct a fluent graph that increases the lengths of the shortest paths between the fluents as much as possible. Therefore, the fluent graph should contain as few edges as possible. In general the complete fluent graph (i.e., the graph where every fluent is connected to every other fluent) is the least informative because the maximal distance obtained from this graph is 1.

The algorithm outline still leaves some open issues:

1. How do we construct a ground formula ϕ that is the disjunctive normal form of $\text{next}(f')$?
2. Which literal $\text{true}(f)$ do we select if there is more than one? Or, in other words, which precondition f' of f do we select?

We will discuss both issues in the following sections.

4.1 Constructing a DNF of $\text{next}(f')$

A formula ϕ in DNF is a set of formulas $\{\psi_1, \dots, \psi_n\}$ connected by disjunctions such that each formula ψ_i is a set of literals connected by conjunctions. We propose the algorithm in Figure 4.1 to construct ϕ such that $\text{next}(f') \supset \phi$.

Algorithm 1 Constructing a formula ϕ in DNF with $\text{next}(f') \supset \phi$.

Input: game description D , ground fluent f'

Output: ϕ , such that $\text{next}(f') \supset \phi$

```

1:  $\phi := \text{next}(f')$ 
2:  $\text{finished} := \text{false}$ 
3: while  $\neg \text{finished}$  do
4:   Replace every positive occurrence of  $\text{does}(r, a)$  in  $\phi$ 
     with  $\text{legal}(r, a)$ .
5:   Select a positive literal  $l$  from  $\phi$  such that  $l \neq \text{true}(t)$ ,  $l \neq \text{distinct}(t_1, t_2)$  and  $l$  is not a re-
     cursively defined predicate.
6:   if there is no such literal then
7:      $\text{finished} := \text{true}$ 
8:   else
9:      $\hat{l} := \bigvee_{h: -b \in D, l\sigma = h\sigma} b\sigma$ 
10:     $\phi := \phi \{l/\hat{l}\}$ 
11:   end if
12: end while
13: Transform  $\phi$  into disjunctive normal form, i.e.,  $\phi = \psi_1 \vee \dots \vee \psi_n$  and each formula  $\psi_i$  is a conjunction of literals.
14: for all  $\psi_i$  in  $\phi$  do
15:   Replace  $\psi_i$  in  $\phi$  by a disjunction of all ground in-
     stances of  $\psi_i$ .
16: end for
```

The algorithm starts with $\phi = \text{next}(f')$. Then, it selects a positive literal l in ϕ and unrolls this literal, that is, it replaces l with the bodies of all rules whose head matches l (lines 9, 10). The replacement is repeated until all predicates that are left are either of the form $\text{true}(t)$, $\text{distinct}(t_1, t_2)$ or recursively defined. Recursively defined predicates are not unrolled to ensure termination of the algorithm. Finally, we transform ϕ into disjunctive normal form and replace each disjunct ψ_i of ϕ by a disjunction of all of its ground instances in order to get a ground formula ϕ .

Note that in line 4, we replace every occurrence of **does** with **legal** to also include the preconditions of the actions that are executed in ϕ . As a consequence the resulting formula ϕ is not equivalent to $\text{next}(f')$. However, $\text{next}(f') \supset \phi$, under the assumption that only legal moves can be executed, i.e., $\text{does}(r, a) \supset \text{legal}(r, a)$. This is sufficient for constructing a fluent graph from ϕ .

Note, that we do not select negative literals for unrolling. The algorithm could be easily adapted to also unroll negative literals. However, in the games we encountered so far, doing so does not improve the obtained fluent graphs but complicates the algorithm and increases the size of the created ϕ . Unrolling negative literals will mainly add negative preconditions to ϕ . However, negative preconditions are not used for the fluent graph because a fluent graph only contains positive preconditions of fluents as edges, according to Definition 2.

4.2 Selecting Preconditions for the Fluent Graph

If there are several literals of the form $\text{true}(f)$ in a disjunct ψ of the formula ϕ constructed above, we have to select one of them as source of the edge in the fluent graph. As already mentioned, the distance $\Delta(s, f)$ computed with the help of the fluent graph is a lower bound on the actual number of steps needed. To obtain a good lower bound, that is one that is as large as possible, the paths between nodes in the fluent graph should be as long as possible. Selecting the best fluent graph, i.e., the one which maximizes the distances, is impossible. Which fluent graph is the best one depends on the states we encounter when playing the game, but we do not know these states beforehand. In order to generate a fluent graph that provides good distance estimates, we use several heuristics when we select literals from disjuncts in the DNF of $\text{next}(f')$:

First, we only add new edges if necessary. That means, whenever there is a literal $\text{true}(f)$ in a disjunct ψ such that the edge (f, f') already exists in the fluent graph, we select this literal $\text{true}(f)$. The rationale of this heuristic is that paths in the fluent graph are longer on average if there are fewer connections between the nodes.

Second, we prefer a literal $\text{true}(f)$ over $\text{true}(g)$ if f is more similar to f' than g is to f' , that is $\text{sim}(f, f') > \text{sim}(g, f')$.

We define the similarity $\text{sim}(t, t')$ recursively over ground terms t, t' :

$$\text{sim}(t, t') = \begin{cases} 1 : & t, t' \text{ have arity 0 and } t = t' \\ \sum_i \text{sim}(t_i, t'_i) : & t = f(t_1, \dots, t_n) \text{ and } t' = f(t'_1, \dots, t'_n) \\ 0 : & \text{else} \end{cases}$$

In human made game descriptions, similar fluents typically have strong connections. For example, in Tic-Tac-Toe $\text{cell}(3, 1, x)$ is more related to $\text{cell}(3, 1, b)$ than to $\text{cell}(b, 3, x)$. By using similar fluents when adding new edges to the fluent graph, we have a better chance of finding the same fluent again in a different disjunct of ϕ . Thus we maximize the chance of reusing edges.

5 Applying Distance Features

For using the distance function in our evaluation function, we define the normalized distance $\delta(s, f)$.

$$\delta(s, f) = \frac{\Delta(s, f)}{\Delta_{\max}(f)}$$

The value $\Delta_{max}(f)$ is the longest distance $\Delta_G(g, f)$ from any fluent g to f , i.e.,

$$\Delta_{max}(f) \stackrel{\text{def}}{=} \max_g \Delta_G(g, f)$$

where $\Delta_G(g, f)$ denotes the length of the shortest path from g to f in the fluent graph G .

Thus, $\Delta_{max}(f)$ is the longest possible distance $\Delta(s, f)$ that is not infinite. The normalized distance $\delta(s, f)$ will be infinite if $\Delta(s, f) = \infty$, i.e., there is no path from any fluent in s to f in the fluent graph. In all other cases it holds that $0 \leq \delta(s, f) \leq 1$.

Note, that the construction of the fluent graph and computing the shortest paths between all fluents, i.e., the distance function Δ_G , need only be done once for a game. Thus, while construction of the fluent graph is more expensive for complex games, the cost of computing the distance feature $\delta(s, f)$ (or $\Delta(s, f)$) only depends (linearly) on the size of the state s .

5.1 Using Distance Features in an Evaluation Function

To demonstrate the application of the distance measure presented, we use a simplified version of the evaluation function of Fluxplayer (Schiffel and Thielscher 2007) implemented in Prolog. It takes the ground DNF of the goal rules as first argument, the current state as second argument and returns the fuzzy evaluation of the DNF on that state as a result.

```

1 eval((D1; ..; Dn), S, R) :- !,
2   eval(D1, S, R1), .., eval(Dn, S, Rn),
3   R is sum(R1, .., Rn) - product(R1, .., Rn)
4 eval((C1, .., Cn), S, R) :- !,
5   eval(C1, S, R1), .., eval(Cn, S, Rn),
6   R is product(R1, .., Rn).
7 eval(true(F), S, 0.9) :- occurs(F, S), !.
8 eval(true(F), S, 0.1).
```

Disjunctions are transformed to probabilistic sums, conjunctions to products, and **true** statements are evaluated to values in the interval $[0, 1]$, basically resembling a recursive fuzzy logic evaluation using the product t-norm and the corresponding probabilistic sum t-conorm. The state value increases with each conjunct and disjunct fulfilled.

We compare this basic evaluation to a second function that employs our relative distance measure, encoded as predicate **delta**. We obtain this distance-based evaluation function by substituting line 8 of the previous program by the following four lines:

```

1 eval(true(F), S, R) :- delta(S, F, Distance),
2   Distance <= 1, !,
3   R is 0.8*(1-Distance) + 0.1.
4 eval(true(F), S, 0).
```

Here, we evaluate a fluent that does not occur in the current state to a value in $[0.1, 0.9]$ and, in case the relative distance is infinite, to 0 since this means that the fluent cannot hold anymore.

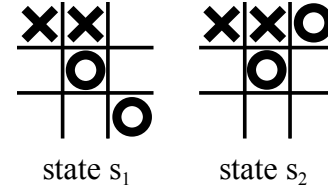


Figure 3: Two states of the Tic-Tac-Toe. The first row is still open in state s_1 but blocked in state s_2 .

5.2 Tic-Tac-Toe

Although on first sight Tic-Tac-Toe contains no relevant distance information, we can still take advantage of our distance function. Consider the two states as shown in Figure 3. In state s_1 the first row consists of two cells marked with an **x** and a blank cell. In state s_2 the first row contains two **x**s and one cell marked with an **o**. State s_1 has a higher state value than s_2 for **xplayer** since in s_1 **xplayer** has a threat of completing a line in contrast to s_2 . The corresponding goal condition for **xplayer** completing the first row is:

```

1 line(x) :- true(cell(1,1,x)),
2   true(cell(2,1,x)), true(cell(3,1,x)).
```

When evaluating the body of this condition using our standard fuzzy evaluation, we see that it cannot distinguish between s_1 and s_2 because both have two markers in place and one missing for completing the line for **xplayer**. Therefore it evaluates both states to $1 * 1 * 0.1 = 0.1$.

However, the distance-based function evaluates **true**(cell(3,1,b)) of s_1 to 0.1 and **true**(cell(3,1,o)) of s_2 to 0. Therefore, it can distinguish between both states, returning $R = 0.1$ for $S = s_1$ and $R = 0$ for $S = s_2$.

5.3 Breakthrough

The second game is Breakthrough, again a two-player game played on a chessboard. Like in chess, the first two ranks contain only white pieces and the last two only black pieces. The pieces of the game are only pawns that move and capture in the same way as pawns in chess, but without the initial double advance. Whoever reaches the opposite side of the board first wins.² Figure 4 shows the initial position for Breakthrough. The arrows indicate the possible moves, a pawn can make.

The goal condition for the player **black** states that black wins if there is a cell with the coordinates $X, 1$ and the content **black**, such that X is an index (a number from 1 to 8 according to the rules of **index**):

```

1 goal(black, 100) :-
2   index(X), true(cellholds(X, 1, black)).
```

Grounding this rule yields

²The complete rules for Breakthrough as well as Tic-Tac-Toe can be found under <http://ggpsserver.general-game-playing.de/>

```

1 goal(black, 100) :-
2   (true(cellholds(1, 1, black);
3   ..;
4   true(cellholds(8, 1, black)))

```

We omitted the `index` predicate since it is true for all 8 ground instances.

The standard evaluation function cannot distinguish any of the states in which the goal is not reached because `true(cellholds(X, 1, black))` is false in all of these states for any instance of `X`.

The distance-based evaluation function is based on the fluent graph depicted in Figure 5.

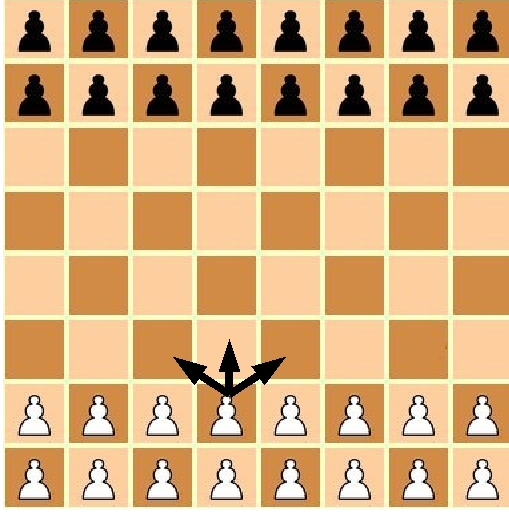


Figure 4: Initial position in Breakthrough and the move options of a pawn.

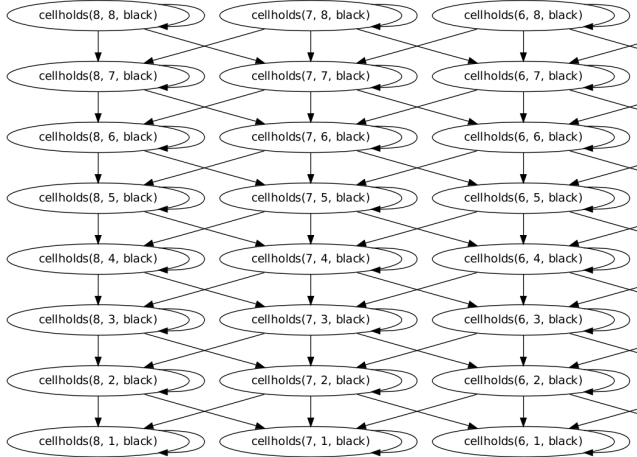


Figure 5: A partial fluent graph for Breakthrough, role black.

Obviously it captures the way pawns move in chess. Therefore evaluations of atoms of the form `true(cellholds(X, Y, black))` have now 9 possible

values (for distances 0 to 7 and ∞) instead of 2 (true and false). Hence, states where black pawns are nearer to one of the cells $(1, 8), \dots, (8, 8)$ are preferred.

Moreover, the fluent graph, and thus the distance function, contains the information that some locations are only reachable from certain other locations. Together with our evaluation function this leads to what could be called “strategic positioning”: states with pawns on the side of the board are worth less than those with pawns in the center. This is due to the fact that a pawn in the center may reach more of the 8 possible destinations than a pawn on the side.

6 Evaluation

For evaluation, we implemented our distance function and equipped the agent system Fluxplayer (Schiffel and Thielscher 2007) with it. We then set up this version of Fluxplayer (“flux_distance”) against its version without the new distance function (“flux_basic”). We used the version of Fluxplayer that came in 4th in the 2010 championship. Since flux_basic is already endowed with a distance heuristic, the evaluation is comparable to a competition setting of two competing heuristics using distance features.

We chose 19 games for comparison in which we conducted 100 matches on average. Figure 6 shows the results.

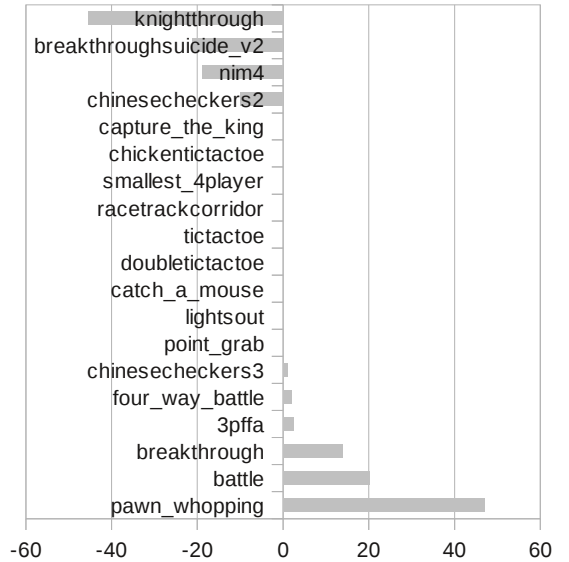


Figure 6: Advantage in Win Rate of flux_distance

The values indicate the difference in win rate, e.g., a value of +10 indicates the flux_distance won 55% of the games against flux_basic winning 45%. Obviously the proposed heuristics produces results comparable to the flux_basic heuristics, with both having advantages in some games. This has several reasons: Most importantly, our proposed heuristic, in the way it is implemented now, is more expensive than the distance estimation used in flux_basic. Therefore the evaluation of a state takes longer and the search tree can

not be explored as deeply as with cheaper heuristics. This accounts for three of the four underperforming games. For example in `nim4`, the `flux_basic` distance estimation provides essentially the same results as our new approach, just much cheaper. In `chinesecheckers2` and `knightthrough`, the new distance function slows down the search more than its better accuracy can compensate.

On the other hand, `flux_distance` performs better in complicated games since our proposed distance evaluation is a better estimate. Interestingly, `flux_distance` therefore loses in `breakthrough_suicide`. The game is exactly the same as `breakthrough`, however, the player to reach the other side of the board first does not win, but loses. The heuristics of both `flux_basic` and `flux_distance` are not good for this game. However, the better distance estimates of `flux_distance` amplify the (bad) heuristics. In this case, `flux_distance` does not capture advanced opponent pawns, because these pawns are near to the side of the board on which the opponent loses. However, these pawns than capture `flux_distance`'s own pawns such that `flux_distance` loses pieces and is forced to advance the few remaining pawns quickly. Thus, the problem in this game is not the distance estimate but the fact that the heuristic is not suitable for the game.

Finally, in some of the games no changes were found since both distance estimates performed equally well. However, rather specific heuristics and analyzation methods of `flux_basic` could be replaced by our new general approach. For example, the original Fluxplayer contains a special method to detect when a fluent is unreachable, while this information is automatically included in our distance estimate.

Apart from the above results, we intended to use more games for evaluation, however, we found that the fluent graph construction takes too much time in some games where the `next` rules are complex. We discuss these issues in Section 8.

7 Related Work

Distance features are part of classical agent programming for games like chess and checkers in order to measure, e.g., the distance of a pawn to the promotion rank. A more general detection mechanism was first employed in Metagamer (Pell 1993) where the features “promote-distance” and “arrival-distance” represented a value indirectly proportional to the distance of a piece to its arrival or promotion square. However, due to the restriction on symmetric chess-like games, boards and their representation were predefined and thus predefined features could be applied as soon as some promotion or arrival condition was found in the game description.

Currently, a number of GGP agent systems apply distance features in different forms. UTexas (Kuhlmann, Dresner, and Stone 2006) identifies order relations syntactically and tries to find 2d-boards with coordinates ordered by these relations. Properties of the content of these cells, such as minimal/maximal x- and y-coordinates or pair-wise Manhattan distances are then assumed as candidate features and may be used in the evaluation function. Fluxplayer (Schiffel and

Thielscher 2007) generalizes the detection mechanism using semantic properties of order relations and extends board recognition to arbitrarily defined n -dimensional boards.

Another approach is pursued by Cluneplayer (Clune 2007) who tries to impose a symbol distance interpretation on expressions found in the game description. Symbol distances, however, are again calculated using Manhattan distances on ordered arguments of board-like fluents, eventually resulting in a similar distance estimate as UTexas and Fluxplayer.

Although not explained in detail, Ogre (Kaiser 2007) also employs two features that measure the distance from the initial position and the distance to a target position. Again, Ogre relies on syntactic detection of order relations and seems to employ a board centered metrics, ignoring the piece type.

All of these approaches rely on the identification of structures in the game (such as game boards), but can not be used for fluents that do not belong to such a structure. Furthermore, they make assumptions about the distances on these structures (usually Manhattan distance) that are not necessarily connected to the game dynamics, e.g., how different pieces move on a board.

In domain independent planning, distance heuristics are used successfully, e.g., in HSP (Bonet and Geffner 2001) and FF (Hoffmann and Nebel 2001). The heuristics $h(s)$ used in these systems is an approximation of the plan length of a solution in a relaxed problem, where negative effects of actions are ignored. This heuristics is known as delete list relaxation. While on first glance this may seem very similar to our approach, several differences exist:

- The underlying languages, GDL for general game playing and PDDL for planning, are different. A translation of GDL to PDDL is expensive in many games (Kissmann and Edelkamp 2010). Thus, directly applying planning systems is not often not feasible.
- The delete list relaxation considers all (positive) preconditions of a fluent, while we only use one precondition. This enables us to precompute the distance between the fluents of a game.
- While goal conditions of most planning problems are simple conjunctions, goals in the general games can be very complex (e.g., checkmate in chess). Additionally, the plan length is usually not a good heuristics, given that only the own actions and not those of the opponents can be controlled. Thus, distance estimates in GGP are usually not used as the only heuristics but only as a feature in a more complex evaluation function. As a consequence, computing distance features must be relatively cheap.
- Computing the plan length of the relaxed planning problem is NP-hard, and even the approximations used in HSP or FF that are not NP-hard require to search the state space of the relaxed problem. On the other hand, computing distance estimates with our solution is relatively cheap. The distances $\Delta_G(f, g)$ between all fluents f and g in the fluent graph can be precomputed once for a game. Then, computing the distance $\Delta(s, f')$ (see Definition 3) is lin-

ear in the size of the state s , i.e., linear in the number of fluents in the state.

8 Future Work

The main problem of the approach is its computational cost for constructing the fluent graph. The most expensive steps of the fluent graph construction are grounding of the DNF formulas ϕ and processing the resulting large formulas to select edges for the fluent graph. For many complex games, these steps cause either out-of-memory or time-out errors. Thus, an important line of future work is to reduce the size of formulas before the grounding step without losing relevant information.

One way to reduce the size of ϕ is a more selective expansion of predicates (line 5) in Algorithm 4.1. Developing heuristics for this selection of predicates is one of the goals for future research.

In addition, we are working on a way to construct fluent graphs from non-ground representations of the preconditions of a fluent to skip the grounding step completely. For example, the partial fluent graph in Figure 1 is identical to the fluent graphs for the other 8 cells of the Tic-Tac-Toe board. The fluent graphs for all 9 cells are obtained from the same rules for `next (cell (X, Y, _))`, just with different instances of the variables X and Y . By not instantiating X and Y , the generated DNF is exponentially smaller while still containing the same information.

The quality of the distance estimates depends mainly on the selection of preconditions. At the moment, the heuristics we use for this selection are intuitive but have no thorough theoretic or empiric foundation. In future, we want to investigate how these heuristics can be improved.

Furthermore, we intend to enhance the approach to use fluent graphs for generalizations of other types of features, such as, piece mobility and strategic positions.

9 Summary

We have presented a general method of deriving distance estimates in General Game Playing. To obtain such a distance estimate, we introduced fluent graphs, proposed an algorithm to construct them from the game rules and demonstrated the transformation from fluent graph distance to a distance feature.

Unlike previous distance estimations, our approach does not rely on syntactic patterns or internal simulations. Moreover, it preserves piece-dependent move patterns and produces an admissible distance heuristic.

We showed on an example how these distance features can be used in a state evaluation function. We gave two examples on how distance estimates can improve the state evaluation and evaluated our distance against Fluxplayer in its most recent version.

Certain shortcomings should be addressed to improve the efficiency of fluent graph construction and the quality of the obtained distance function. Despite these shortcomings, we found that a state evaluation function using the new distance estimates can compete with a state-of-the-art system.

References

- Bonet, B., and Geffner, H. 2001. Planning as heuristic search. *Artificial Intelligence* 129(1–2):5–33.
- Clune, J. 2007. Heuristic evaluation functions for general game playing. In *AAAI*. Vancouver: AAAI Press.
- Hoffmann, J., and Nebel, B. 2001. The FF planning system: Fast plan generation through heuristic search. *JAIR* 14:253–302.
- Kaiser, D. M. 2007. Automatic feature extraction for autonomous general game playing agents. In *Proceedings of the Sixth Intl. Joint Conf. on Autonomous Agents and Multi-agent Systems*.
- Kissmann, P., and Edelkamp, S. 2010. Instantiating general games using prolog or dependency graphs. In Dillmann, R.; Beyerer, J.; Hanebeck, U. D.; and Schultz, T., eds., *KI*, volume 6359 of *Lecture Notes in Computer Science*, 255–262. Springer.
- Kuhlmann, G.; Dresner, K.; and Stone, P. 2006. Automatic Heuristic Construction in a Complete General Game Player. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence*, 1457–62. Boston, Massachusetts, USA: AAAI Press.
- Love, N.; Hinrichs, T.; Haley, D.; Schkufza, E.; and Genereth, M. 2008. General game playing: Game description language specification. Technical Report March 4, Stanford University. The most recent version should be available at <http://games.stanford.edu/>.
- Pell, B. 1993. *Strategy generation and evaluation for meta-game playing*. Ph.D. Dissertation, University of Cambridge.
- Schiffel, S., and Thielscher, M. 2007. Fluxplayer: A successful general game player. In *Proceedings of the National Conference on Artificial Intelligence*, 1191–1196. Vancouver: AAAI Press.
- Schiffel, S., and Thielscher, M. 2009a. Automated theorem proving for general game playing. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*.
- Schiffel, S., and Thielscher, M. 2009b. A multiagent semantics for the game description language. In Filipe, J.; Fred, A.; and Sharp, B., eds., *International Conference on Agents and Artificial Intelligence (ICAART)*. Porto: Springer.

On the Complexity of BDDs for State Space Search: A Case Study in Connect Four

Stefan Edelkamp and Peter Kissmann
TZI Universität Bremen, Germany
{edelkamp, kissmann}@tzi.de

Abstract

Symbolic search using BDDs usually saves huge amounts of memory, while in some domains its savings are moderate at best. It is an open problem to determine if BDDs work well for a certain domain.

Motivated by finding evidences for BDD growths for state space search, in this paper we are concerned with symbolic search in the domain of CONNECT FOUR. We prove that there is a variable ordering for which the set of all possible states – when continuing after a terminal state has been reached – can be represented by polynomial sized BDDs, whereas the termination criterion leads to an exponential number of nodes in the BDD given any variable ordering.

1 Introduction

Binary Decision Diagrams (BDDs) [Bryant, 1986] are a very efficient data structure for storing large sets of states, and we can use them to perform set-based search. For some domains they can save an exponential amount of memory (compared to explicit representation), in others the saving is only a small linear factor, while in most domains its savings are dependent on the variable ordering.

Thus, when trying to work on a domain with BDDs, a question that immediately arises is whether the BDDs will be of any help. For some well-known planning benchmarks such an analysis has already been done. For example, Hung [1997] has proven that BDDs for representing the exact set of reachable states in permutation games such as most SLIDING TILES PUZZLES, especially the well-known $(n^2 - 1)$ -PUZZLE, or the BLOCKSWORLD domain, are of size exponential in the number of variables of the permutation, no matter what variable ordering is chosen. On the other hand, we have shown that, given a good variable ordering, BDDs can save an exponential amount of memory in the domain of GRIPPER [Edelkamp and Kissmann, 2008a].

The game of CONNECT FOUR belongs to the broader class of k -in-a-row games such as TIC-TAC-TOE or GOMOKU. On a $w \times h$ board (w being the width and h the height) the two players (red and yellow) take turns placing their pieces on the board. In contrast to most other games of this class gravity is

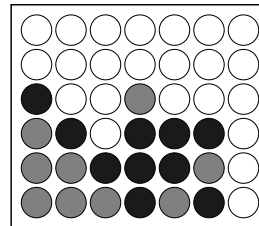


Figure 1: The CONNECT FOUR domain.

important in CONNECT FOUR so that the pieces fall as far to the bottom as possible. Thus, in each state at most w moves are possible (placing the piece in one of the columns, if it is not yet completely filled). The goal is to construct a line of (at least) four pieces of the own color (cf. Figure 1).

CONNECT FOUR on a 7×6 board has been solved independently only a few days apart by Allis [1988] and Allen¹ [2011], but the precise number of reachable states remained unknown – Allis provided only a rough estimate of about $7 \cdot 10^{13}$ states. In an earlier experiment we have determined the exact number with symbolic search using BDDs in about 15 hours [Edelkamp and Kissmann, 2008b] – there are exactly 4,531,985,219,092 states reachable. Later, Tromp came up with the same number using explicit search, which took about 10,000 hours.² Given these runtimes it seems likely that BDDs are good for generating the set of reachable states.

In this paper we will show that BDDs really are good for representing the reachable states if we continue search after finding a terminal state. When stopping at terminal states, things are more difficult. This is because, in contrast to all the previously mentioned problems, the termination criterion in CONNECT FOUR is complex and needs a BDD with an exponential number of nodes to be represented.

In the following we will give introductions to BDDs, the importance of the variable ordering as well as to symbolic search. Then, we will prove that the reachable states of each BFS layer can be represented by a BDD polynomial in the board size, while representing even a reduced termination criterion already results in a BDD of size exponential in the smaller of the board's edges.

¹Reported in rec.games.programmer on October 1st 1988.

²<http://homepages.cwi.nl/~tromp/c4/c4.html>

2 Binary Decision Diagrams

Binary Decision Diagrams (BDDs) [Bryant, 1986] are a memory-efficient data structure used to represent Boolean functions as well as to perform set-based search. In short, a BDD is a directed acyclic graph with one root and two terminal nodes, the 0- and the 1-sink. Each internal node corresponds to a binary variable and has two successors, one (along the low edge) representing that the current variable is false ($\perp$) and the other (along the high edge) representing that it is true ($\top$). For any assignment of the variables derived from a path from the root to the 1-sink the represented function will be evaluated to $\top$.

Bryant proposed to use a fixed variable ordering, for which he also provided two reduction rules (eliminating nodes with the same low and high successors and merging two nodes representing the same variable that share the same low successor as well as the same high successor). These BDDs were called reduced ordered binary decision diagrams (ROBDDs). Whenever we mention BDDs in this paper we actually refer to ROBDDs.

2.1 The Variable Ordering

For many Boolean formulas the corresponding BDD can be of polynomial or even linear size, given a good variable ordering π , although the number of satisfying assignments (i. e., the number of states) might be exponential in n . One such case is the Disjoint Quadratic Form (DQF).

Definition 1 (Disjoint Quadratic Form). *The disjoint quadratic form DQF_n is defined as*

$$DQF_n(x_1, \dots, x_n, y_1, \dots, y_n) := x_1 y_1 \vee x_2 y_2 \vee \dots \vee x_n y_n.$$

Proposition 1 (Linear Size of a BDD for DQF). *Given the variable ordering $\pi = (x_1, y_1, \dots, x_n, y_n)$ the BDD representing the DQF_n contains $2n + 2$ nodes.*

On the other hand, a bad variable ordering might result in a BDD of size exponential in n . For the DQF we can find such an ordering as well.

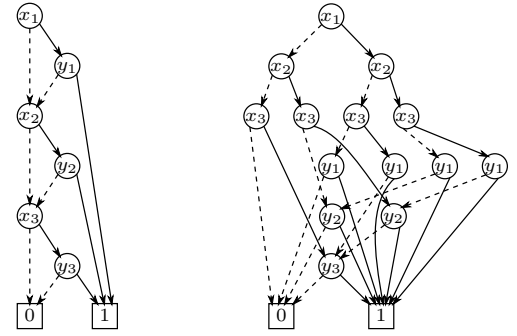
Proposition 2 (Exponential Size of a BDD for DQF). *Given the variable ordering $\pi = (x_1, \dots, x_n, y_1, \dots, y_n)$ the BDD representing the DQF_n contains 2^{n+1} nodes.*

Figure 2 illustrates both cases for the DQF_3 .

Furthermore, there are some functions for which any BDD contains $\Omega(2^n)$ nodes, e. g., integer multiplication, for which Bryant [1986] proved that for any variable ordering there is an output bit for which the BDD contains at least $2^{n/8}$ BDD nodes.

Finally, for some functions any variable ordering results in a polynomial sized BDD. An example for this would be any symmetric function, for which a BDD contains at most $\mathcal{O}(n^2)$ nodes.

Nevertheless, for many functions there are good variable orderings and finding these might be crucial. Yet, finding an ordering that minimizes the number of BDD nodes is co-NP-complete [Bryant, 1986]. Furthermore, the decision whether a variable ordering can be found so that the resulting BDD contains at most s nodes, with s being specified beforehand, is NP-complete [Bollig and Wegener, 1996]. Additionally,



(a) Good variable ordering. (b) Bad variable ordering.

Figure 2: Two possible BDDs representing DQF_3 . Solid lines represent high edges, dashed ones low edges.

Sieling [2002] proved that there is no polynomial time algorithm that can calculate a variable ordering so that the resulting BDD contains at most c times as many nodes as the one with optimal variable ordering for any constant $c > 1$, so that we cannot expect to find good variable orderings using reasonable resources. This shows that using heuristics to calculate a variable ordering [Fujita *et al.*, 1988; Malik *et al.*, 1988; Butler *et al.*, 1991] is a plausible approach.

2.2 Symbolic Search

State space search is an important topic in many areas of artificial intelligence. No matter if we want to find a shortest path from one location to another, some plan transforming a given initial state to another state satisfying a goal condition, or finding good moves for a game – in all these cases search is of high importance.

In explicit search we need to come up with good data structures to represent the states efficiently or algorithms for fast transformation of states. In case of symbolic search [McMillan, 1993] with BDDs the data structure to use is already given. As to efficient algorithms, here we need to handle things in a different manner compared to explicit search. While explicit search is concerned with expansion of single states and calculation of successors of a single state, in symbolic search we handle sets of states. The assignments satisfying a Boolean formula can be seen as sets of states, as well. Similarly, we can represent any state we encounter in a state space search as a Boolean formula with one satisfying assignment. To achieve this, we represent any state as a conjunction of (binary) variables. Thus, a set of states can be seen as the disjunction of such a conjunction of variables, so that we can easily represent it by a BDD.

To perform symbolic search we need two sets of binary variables. One set, S , stores the current states, while the other set, S' , stores the successor states. With these, we can represent a transition relation, which connects the current states with their successors.

For a given set of actions $\mathcal{A}$, any action $a \in \mathcal{A}$ typically consists of a precondition and a number of effects to specify the changes to a state satisfying the precondition and thus the successor state. For each $a = (pre_a, eff_a) \in \mathcal{A}$

Algorithm 1 Symbolic Breadth-First Search (SBFS)

```

1:  $reach_0 \leftarrow \mathcal{I}$ ;
2:  $layer \leftarrow 0$ ;
3: while  $reach_{layer} \neq \perp$  do
4:    $current \leftarrow reach_{layer} \wedge \neg \mathcal{T}$ ;
5:    $reach_{layer+1} \leftarrow image(current)$ ;
6:   for  $0 \leq i \leq layer$  do
7:      $reach_{layer+1} \leftarrow reach_{layer+1} \wedge \neg reach_i$ ;
8:   end for
9:    $layer \leftarrow layer + 1$ ;
10: end while

```

we can generate one transition relation $trans_a(S, S') := pre_a(S) \wedge eff_a(S') \wedge frame_a(S, S')$ with $frame_a$ being the frame for action a , which determines what parts of the current state do not change when applying a . These transition relations can be combined to generate a single (monolithic) transition relation $trans$ by calculating the disjunction of all the smaller transition relations:

$$trans(S, S') := \bigvee_{a \in \mathcal{A}} trans_a(S, S')$$

To calculate the successors of a set of states *current*, the *image* operator is defined as

$$(\exists S.trans(S, S') \wedge current(S)) [S' \rightarrow S].$$

Given a set of states *current*, specified in S , it calculates all the successor states, specified in S' . Afterwards, all variables of S are replaced by those of S' (denoted by $[S' \rightarrow S]$), so that the successor states finally are represented in S and we can continue the search.

The combination of conjunction and existential quantification used in the *image* operator, the so-called relational product, is one of the bottlenecks in symbolic search, as its calculation is NP-complete [McMillan, 1993]. Most BDD packages contain implementations of the relational product which are much more efficient than the manual execution of conjunction followed by the existential quantification.

Given an implementation of the *image* operator, performing symbolic breadth-first search (BFS) is straight-forward (cf. Algorithm 1). For any search problem $\mathcal{P} = \langle \mathcal{V}, \mathcal{A}, \mathcal{I}, \mathcal{T} \rangle$ with $\mathcal{V}$ being a set of (Boolean) variables, $\mathcal{A}$ a set of actions, $\mathcal{I}$ the initial state and $\mathcal{T}$ the set of terminal states, all we need to do is to apply the *image* operator first to $\mathcal{I}$ and afterwards to the last generated successor state set. The search ends when a fix-point is reached (line 3). For this we must remove any terminal states before expanding a set of states (line 4) and all duplicate states that might reside in a newly generated BFS layer (lines 6 – 8).

3 BDD Sizes for CONNECT FOUR

In this section we will analyze the behavior of BDDs in the context of CONNECT FOUR. First, we will prove that the calculation of the set of reachable states can be done using polynomial sized BDDs if we continue the search after reaching terminal states. Afterwards, we will have a closer look at the termination criterion and prove that even for a simpler version of it we already need exponentially many BDD nodes.

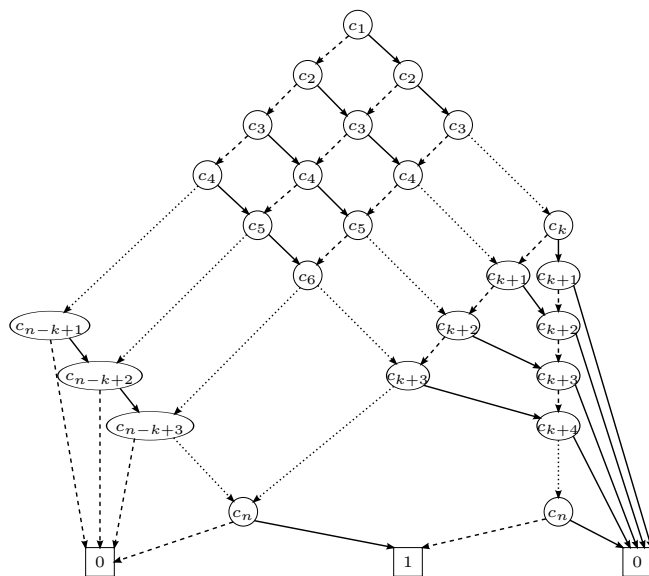


Figure 3: Critical part of the BDD for representing exactly k cells being occupied by pieces of one color. Solid edges correspond to the cell being occupied by a piece of the color in question, dashed ones to the other case.

3.1 Polynomial Upper Bound for Representing the Reachable States

CONNECT FOUR is a game where in each state l pieces are placed on the $n = wh$ positions of the game board, so that its search space is exponential. Nevertheless, we can find a variable ordering, so that the reachable states can be represented by a number of polynomial sized BDDs (one for each BFS layer) if we continue after a terminal state has been reached, i. e., if we omit the removal of terminal states (line 4 of Algorithm 1).

Theorem 3 (Polynomial Bound for BFS Layers without Termination in CONNECT FOUR). *There is a binary state encoding and a variable ordering for which the BDD size for the characteristic function of any BFS layer in CONNECT FOUR is polynomial in $n = wh$, with w being the width and h the height of the board, if we do not cut off the search after reaching a terminal state.*

Proof. We use $2n + 1$ bits to encode a state in CONNECT FOUR, two for each cell (red / yellow / none) and one for the active player. If we organize the encoding in a column-wise manner we need to calculate the conjunction of three polynomial sized BDDs for each layer l .

Two BDDs are similar to the one in Figure 3. The first represents the fact that we have $k = \lfloor l/2 \rfloor$ red pieces on the board, the second one that we have $k = \lfloor l/2 \rfloor$ yellow pieces on the board. As in this game the cells can also be empty we need to replace each node by two, the first one distinguishing between the cases that a cell is empty or not and the second one between the colors. If the cell is empty we insert an edge to a node for the next cell; if it is not we distinguish between the two colors (cf. Figure 4 for red pieces; for yellow pieces, the low and high edges of the lower node are swapped). Due

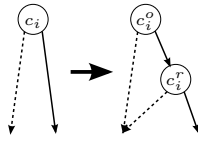


Figure 4: Replacement of a node representing that cell i is occupied by a red piece by two nodes denoting that the cell is occupied (c_i^o) and the occupying piece is red (c_i^r).

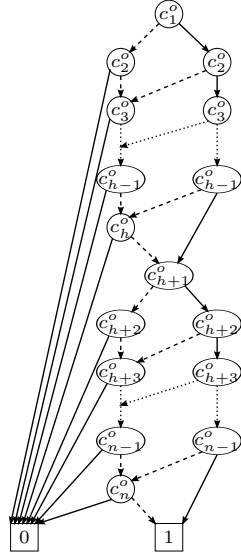


Figure 5: BDD for representing the fact that all pieces are located at the bottom. Each node c_i^o represents the decision for cell i if it is occupied.

to the replacement of each cell by two BDD nodes, each BDD has a size of $\mathcal{O}(2kn) = \mathcal{O}(ln)$.

The third BDD represents the gravity, i.e., the fact that all pieces are located at the bottom of the columns. For this a column-wise variable ordering is important. Starting at the lowest cell of a column, if one is empty all the following cells of this column are necessarily empty as well. On the other hand, if a cell is not empty those above it can still be empty or not. In this BDD it is only relevant if a cell is occupied, so that the actual color of a piece occupying a cell does not matter, which keeps the BDD smaller. Thus, we need at most two BDD nodes for each cell, resulting in a total of $\mathcal{O}(2n) = \mathcal{O}(n)$ BDD nodes (cf. Figure 5).

Calculating the conjunction of two BDDs of sizes s_1 and s_2 results in a new BDD of size $\mathcal{O}(s_1 s_2)$, so that the conjunction of three polynomial BDDs is still polynomial. In this case, the total BDD for each layer l is of size $\mathcal{O}(l^2 n^3)$. As l is at most n , we arrive at a total size of $\mathcal{O}(n^5)$. $\square$

3.2 Exponential Lower Bound for the Termination Criterion

In the domains analyzed before, the termination criterion was always trivial to model as a BDD. E.g., in the $(n^2 - 1)$ -PUZZLE and BLOCKSWORLD only one pattern is correct, while in GRIPPER all balls must end up in room B .

In contrast to these, integrating the terminal states in the set of reachable states (i.e., not continuing search from terminal states) in CONNECT FOUR is a lot more difficult. Here, the termination criterion is fulfilled if one player has succeeded in placing four own pieces in a row (horizontally, diagonally, or vertically), which results in a rather complex Boolean formula.

In the following we will prove that even for a simplified version of this termination criterion, i.e., one that is fulfilled when four adjacent cells in a horizontal or vertical line are occupied, any BDD is of exponential size for a CONNECT FOUR problem encoded in the way proposed in Theorem 3.

To prove this, we can reduce the termination criterion even further.

Definition 2 (Reduced Termination Criterion). *The reduced termination criterion $\mathcal{T}_r$ is the reduction of the full termination criterion to the case where only two horizontally or vertically adjacent cells must be occupied, i.e., $\mathcal{T}_r = \bigvee_{1 \leq i \leq w, 1 \leq j \leq h-1} (x_{i,j} \wedge x_{i,j+1}) \vee \bigvee_{1 \leq i \leq w-1, 1 \leq j \leq h} (x_{i,j} \wedge x_{i+1,j})$ with $x_{i,j}$ specifying that the cell at column i and row j is occupied.*

We can represent the game board by a grid graph $G = (V, E)$, with V representing the wh cells of the board, with edges only between horizontally or vertically adjacent nodes. The reduced termination criterion corresponds to the fact that two nodes connected by an edge are occupied.

Additionally, we make use of the disjoint quadratic form DQF, which we introduced in Definition 1.

Lemma 4 (Exponential Bound for $\mathcal{T}_r$). *For CONNECT FOUR in the cell-wise encoding from Theorem 3 with a board size of $w \times h$ the size of the BDD for representing $\mathcal{T}_r$ is exponential in $\min(w, h)$, independent of the chosen variable ordering.*

Proof. To prove this we show that any variable ordering can be split in two, so that $\Omega(\min(w, h))$ variables are in the one part and connected to $\Omega(\min(w, h))$ variables in the other part. Retaining at most one connection for each variable, we will show that $\Omega(\min(w, h)/7)$ edges still connect the two sets. These represent a DQF with a bad variable ordering, which will give us the exponential bound.

Given any ordering π on the variables we divide it into two sets π_1 and π_2 with π_1 being the first $\lceil wh/2 \rceil$ variables of π and π_2 the remaining $\lfloor wh/2 \rfloor$ variables. These sets correspond to a partitioning of the grid graph G into two sets.

The minimal number of crossing edges between these partitions is in $\Omega(\min(w, h))$. It is achieved when each partition is connected and forms a rectangle of size $(\max(w, h)/2) \cdot \min(w, h)$, so that the cut between the two partitions crosses the grid in the middle but orthogonal to the longer edge and thus crosses $\min(w, h)$ edges.

We might view the two partitions along with the connecting edges as a bipartite graph $G' = (V_1, V_2, E')$, $|E'| = \min(w, h)$, for which we want to find a (not necessarily maximal) matching. To find one, we choose one node from V_1 and a connecting edge along with the corresponding node from V_2 . The other edges starting at one of these nodes are removed (there are at most six of them, as each node can be connected to at most four neighbors, one of which is chosen).

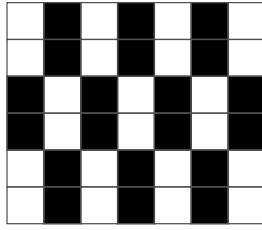


Figure 6: Coloring of the game board in Lemma 5.

This we repeat until we cannot remove any more edges. As we choose one edge in each step and remove at most six, the final matching M contains at least $|E'|/7 = \min(w, h)/7$ edges.

All variables that are not part of the matching are set to $\perp^3$, while the others are retained. Thus, the corresponding formula for $\mathcal{T}_r$ is the DQF over the two sets of variables representing the nodes that are part of the matching. All variables for the one set ($V_1 \cap M$) appear before all variables of the other set ($V_2 \cap M$), so that we have a bad ordering for these variables and the corresponding BDD is of exponential size. Thus, according to Proposition 2 the BDD will contain at least $\Omega(2^{\min(w, h)/7}) = \Omega((2^{1/7})^{\min(w, h)}) = \Omega(1.1^{\min(w, h)})$ nodes. $\square$

This result immediately implies that for square boards the number of BDD nodes is exponential in the square root of the number of cells, i. e., the BDD for representing $\mathcal{T}_r$ contains at least $\Omega(1.1^{\sqrt{n}})$ nodes.

With this result we can show that the case of four adjacent cells in a horizontal or vertical line being occupied results in a BDD of exponential size as well.

Lemma 5 (Exponential Bound for Four Pieces in a Horizontal or Vertical Line). *Any BDD representing the part of $\mathcal{T}$ that states that four cells in a horizontal or vertical line must be occupied is exponential in $\min(w, h)$ for a board of size $w \times h$.*

Proof. We can devise a two-coloring of the game board as shown in Figure 6. Using this, any line of four pieces resides on two white and two black cells. If we replace all variables representing the black cells by $\top$ we can reduce this part of $\mathcal{T}$ to $\mathcal{T}_r$. As this replacement does not increase the BDD size, the BDD is exponential in $\min(w, h)$ as well. $\square$

In a similar fashion we can show that any BDD for the fact that two (or four) diagonally adjacent cells are occupied is exponential for any variable ordering. The combination of all these results illustrates that any BDD representing the termination criterion in CONNECT FOUR is of exponential size, no matter what variable ordering is chosen.

³Replacing a variable by a constant does not increase the number of BDD nodes.

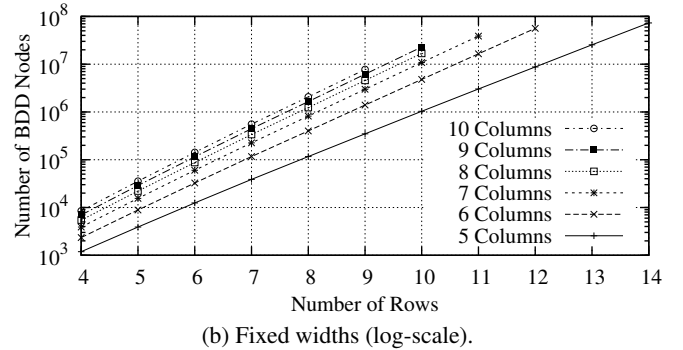
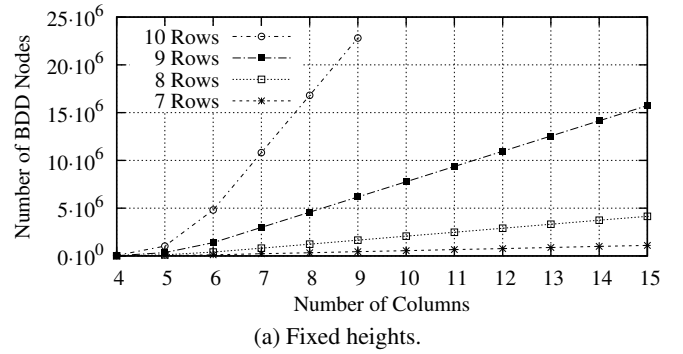


Figure 7: Number of BDD nodes for the termination criterion for one player. The variable ordering is according to the columns.

4 Experimental Evaluation

For a column-wise variable ordering we have evaluated the number of BDD nodes for representing the termination criterion of four pieces of one player being placed in a horizontal or vertical line for several board sizes (cf. Figure 7). For a fixed number of rows (cf. Figure 7a) the growth of the number of BDD nodes is linear in the number of columns, while for a fixed number of columns (cf. Figure 7b) it is exponential in the number of rows. Though it first might seem like a contradiction to our result, it actually shows that it holds for the chosen variable ordering.

In this variable ordering the partitions induced by the split of the variable ordering cut h edges. Thus, the resulting BDD's size is exponential in h . For the case of a fixed height, h does not change, so that the size increases by a constant amount of nodes. On the other hand, when fixing the width, the size changes exponentially in the height, no matter if it is smaller or larger than the width. This happens because for the cases that the height is actually larger than the width this variable ordering is not the best choice – in those cases it should have been in a row-wise manner.

Furthermore, we have evaluated the default instance of 7×6 along with the full termination criterion in two complete searches, one stopping at terminal states and one continuing after them (cf. Figure 8). Again we used a column-wise variable ordering. We can see that the number of states in the first is smaller than in the second case, while the number of BDD nodes needed to represent these states is larger in the

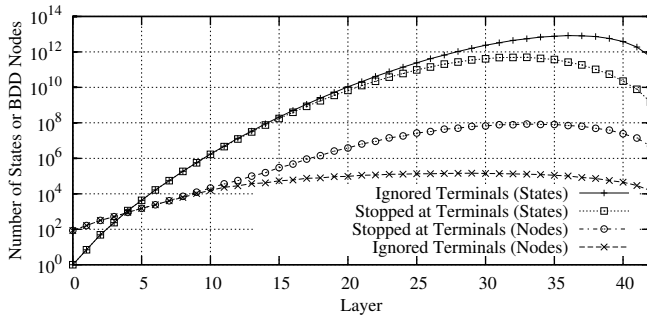


Figure 8: Comparison of the number of states and BDD nodes for different layers with a 7×6 board when ignoring terminal states and when stopping at terminal states. The variable ordering is according to the columns.

Table 1: Classification of some benchmarks with respect to the BDD sizes.

	Polynomial Reachability	Exponential Reachability
Polynomial Termination Criterion	GRIPPER	BLOCKSWORLD SLIDING TILES PUZZLE ($n^2 - 1$)-PUZZLE
Exponential Termination Criterion	CONNECT FOUR TIC-TAC-TOE GOMOKU	

first case. Thus, for this variable ordering we can clearly see that the use of the termination criterion increases the sizes of the BDDs.

5 Discussion

So far we have only investigated one encoding of the game, i. e., the one from the board’s point of view (each cell may be occupied by at most one piece). Another reasonable encoding would be from the pieces’ point of view (each piece may be placed on at most one cell). For this we expect the situation to be similar, i. e., we do not expect the termination criterion to be representable by a polynomial sized BDD. Furthermore, the encoding of each state greatly increases (in our encoding we need $2n$ bits for each state, while in the second encoding we would need $n \lceil \log n \rceil$ bits, as we need to distinguish between the n pieces that can be placed on the board. This might even have a negative impact on the complexity of the reachability analysis.

With the new results established in this paper we can now classify several games with respect to the complexity of BDDs (cf. Table 1).

The lower bound results of this paper are surprisingly general. Though in CONNECT FOUR we only need the case that a number of adjacent cells are occupied by pieces of the same color, we have proven that even for the case of the cells being occupied – no matter by what color – the BDDs are exponential. This might come in handy in the analysis of other domains, as well. Though we have not studied these other games in detail, in the following we provide our expectations for some more complex games that are still of interest in the

game research community.

In CHESS a part of the termination criterion is that all cells adjacent to a king are either occupied by other pieces of the king’s color or guarded by the opponent. For this, Lemma 4 seems applicable, so that we would expect the BDD representation for the checkmate to be exponential.

For GO we do not yet have any feeling concerning the complexity of the BDD representation of the reachable states or the termination criterion. Nevertheless, we expect the transition relation to be rather difficult. For example, finding eyes on the board should be similar to the situation in Lemma 4. Finding a setting with an area completely filled with stones is likely more complex than just two adjacent cells being occupied, so that we expect an exponential sized BDD for this case, as well. Even if the reachable states and the termination criterion can be represented by comparably small BDDs, an exponential sized termination criterion will surely result in a long runtime, as it is part of any image calculation. Thus, it might be necessary to expand our classification matrix (Figure 1) to three dimensions, i. e., the size of the BDDs for the reachable states, for the termination criterion, and for the transition relation.

In CHECKERS the transition relation is likely to be complex, as well. To move a piece, an adjacent cell must either be empty or it must be occupied by an opponent’s piece and the next cell in line must be empty. Similar rules hold for double or even longer jumps. Thus, the lemmas seem to be applicable in this case, as well.

6 Conclusion and Future Work

In this paper we have determined bounds for the BDD sizes for representing the reachable states (when ignoring terminal states) as well as for representing the termination criterion in CONNECT FOUR.

In the domains analyzed before the BDDs were either exponential for any variable ordering or polynomial for at least one variable ordering. In CONNECT FOUR we have identified a domain, for which BDDs can be polynomial, but only if we do not construct the BDD for the termination criterion, which itself is exponential.

The results are reasonably general, so that they might help in the analysis of other games, but this remains as future work, as does an in-depth analysis of other encodings for CONNECT FOUR.

7 Acknowledgments

Thanks to Deutsche Forschungsgesellschaft (DFG) for support in the project ED 74/11-1.

References

- [Allen, 2011] James D. Allen. *The Complete Book of Connect 4: History, Strategy, Puzzles*. Puzzlewright, 2011.
- [Allis, 1988] Louis Victor Allis. A knowledge-based approach of connect-four. Master’s thesis, Vrije Universiteit Amsterdam, October 1988.

- [Bollig and Wegener, 1996] Beate Bollig and Ingo Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45(9):993–1002, 1996.
- [Bryant, 1986] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
- [Butler *et al.*, 1991] Kenneth M. Butler, Don E. Ross, Rohit Kapur, and M. Ray Mercer. Heuristics to compute variable orderings for efficient manipulation of ordered binary decision diagrams. In *DAC*, pages 417–420. ACM Press, 1991.
- [Edelkamp and Kissmann, 2008a] Stefan Edelkamp and Peter Kissmann. Limits and possibilities of bdds in state space search. In Dieter Fox and Carla P. Gomes, editors, *AAAI*, pages 1452–1453. AAAI Press, 2008.
- [Edelkamp and Kissmann, 2008b] Stefan Edelkamp and Peter Kissmann. Symbolic classification of general two-player games. In Andreas Dengel, Karsten Berns, Thomas M. Breuel, Frank Bomarius, and Thomas Roth-Berghofer, editors, *KI*, volume 5243 of *LNCS*, pages 185–192. Springer, 2008.
- [Fujita *et al.*, 1988] Masahiro Fujita, Hisanori Fujisawa, and Nobuaki Kawato. Evaluation and improvements of boolean comparison method based on binary decision diagrams. In *ICCAD*, pages 2–5. IEEE, 1988.
- [Hung, 1997] William Hung. Exploiting symmetry for formal verification. Master’s thesis, University of Texas at Austin, 1997.
- [Malik *et al.*, 1988] Sharad Malik, Albert R. Wang, Robert K. Brayton, and Alberto Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *ICCAD*, pages 6–9. IEEE, 1988.
- [McMillan, 1993] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.
- [Sieling, 2002] Detlef Sieling. The nonapproximability of OBDD minimization. *Information and Computation*, 172(2):103–138, 2002.

Game-Tree Properties and MCTS Performance

Hilmar Finnsson and Yngvi Björnsson

School of Computer Science
Reykjavík University, Iceland
{hif,yngvi}@ru.is

Abstract

In recent years Monte-Carlo Tree Search (MCTS) has become a popular and effective search method in games, surpassing traditional $\alpha\beta$ methods in many domains. The question of why MCTS does better in some domains than in others remains, however, relatively open. In here we identify some general properties that are encountered in game trees and measure how these properties affect the success of MCTS. We do this by running it on custom-made games that allow us to parameterize various game properties in order for trends to be discovered. Our experiments show how MCTS can favor either deep, wide or balanced game trees. They also show how the level of game progression relates to playing strength and how disruptive Optimistic Move can be.

Introduction

Monte-Carlo simulations play out random sequences of actions in order to make an informed decision based on aggregation of the simulations end results. The most appealing aspect of this approach is the absence of heuristic knowledge for evaluating game states. Monte-Carlo Tree Search (MCTS) applies Monte-Carlo simulations to tree-search problems, and has nowadays become a fully matured search method with well defined parts and many extensions. In recent years MCTS has done remarkably well in many domains. The reason for its now diverse application in games is mostly due to its successful application in the game of *Go* (Gelly et al. 2006; Enzenberger and Müller 2009), a game where traditional search methods were ineffective in elevating the state-of-the-art to the level of human experts.

Other triumphs of MCTS include games such as *Amazons* (Lorentz 2008), *Lines-of-Action* (Winands, Björnsson, and Saito 2010), *Chinese Checkers* (Sturtevant 2008), *Spades and Hearts* (Sturtevant 2008) and *Settlers of Catan* (Szita, Chaslot, and Spronck 2009).

MCTS is also applied successfully in *General Game Playing (GGP)* (Finnsson and Björnsson 2008), where it outplays more traditional heuristic-based players in many types of games. However, in other type of games, such as many chess-like variants, the MCTS-based GGP agents are hopeless in comparison to their $\alpha\beta$ -based counterparts.

This raises the questions of which game-tree properties inherit to the game at hand make the game suited for MCTS

or not. Having some idea of the answers to these questions can be helpful in deciding if a problem should be attacked with MCTS, and then using which algorithmic enhancements.

In this paper we identify high level properties that are commonly found in game trees to a varying degree and measure how they affect the performance of MCTS. As a testbed we use simple custom made games that both highlight and allow us to vary the properties of interest. The main contribution of the paper is the insight it gives into how MCTS behaves when faced with different game properties.

The paper is structured as follows. In the next section we give a brief overview of MCTS and go on to describing the game-tree properties investigated. Thereafter we detail the experiments we conducted and discuss the results. Finally related work is acknowledged before concluding.

Monte-Carlo Tree Search

MCTS runs as many Monte-Carlo simulations as possible during its deliberation time in order to form knowledge derived from their end results. This knowledge is collected into a tree-like evaluation function mimicking a manageable part of the game-tree. In this paper we talk about the game tree when referring to the game itself, but when we talk about the MCTS tree we are referring to the tree MCTS builds in memory. When time is up the action at the root which is estimated to yield the highest reward is chosen for play.

MCTS breaks its simulation down into a combination of four well defined strategic phases or steps, each with a distinct functionality that may be extended independent of the other phases. These phases are: *selection*, *playout*, *expansion*, and *back-propagation*.

Selection - When the first simulation is started there is no MCTS tree in memory, but it builds up as the simulations run. The Selection phase lasts while the simulation is still at a game tree node which is a part of the MCTS tree. In this phase informed action selection can be made as we have the information already gathered in the tree. For action selection we use a search policy which has the main focus of balancing exploration/exploitation tradeoff. One can select from multiple methods to implement here, but one of the more popular has been the Upper Confidence Bounds applied to Trees (UCT) algorithm (Kocsis and Szepesvári 2006). UCT selects action a in state s from the set of available actions

$A(s)$ given the formula:

$$a^* = \operatorname{argmax}_{a \in A(s)} \left\{ \operatorname{Avg}(s, a) + 2 * C_p \sqrt{\frac{\ln \operatorname{Cnt}(s)}{\operatorname{Cnt}(s, a)}} \right\}$$

Here $\operatorname{Avg}(s, a)$ gives the average reward observed when simulations have included taking action a in state s . The function Cnt returns the number of times state s has been visited on one hand and how often action a has been selected in state s during simulation on the other hand. In (Kocsis and Szepesvári 2006) the authors show that when the rewards lie in the interval $[0, 1]$ having $C_p = 1/\sqrt{2}$ gives the least regret for the exploration/exploitation tradeoff.

Playout - This phase begins when the simulations exit the MCTS tree and have no longer any gathered information to lean on. A common strategy here is just to play uniformly at random, but methods utilizing heuristics or generalized information from the MCTS tree exist to bias the this section of the simulation towards a more descriptive and believable path. This results in more focused simulations and is done in effort to speed up the convergence towards more accurate values within the MCTS tree.

Expansion - Expansion refers here to the in-memory tree MCTS is building. The common strategy here is just to add the first node encountered in the playout step (Coulom 2006). This way the tree grows most where the selection strategy believes it will encounter its best line of play.

Back-Propagation - This phase controls how the end result of the MC simulations are used to update the knowledge stored in the MCTS tree. This is usually just maintaining the average expected reward in each node that was traversed during the course of the simulation. It is easy to expand on this with things like discounting to discriminate between equal rewards reachable at different distances.

Game-Tree Properties

Following are tree properties we identified as being important for MCTS performance and are general enough to be found in a wide variety of games. It is by no means a complete list.

Tree Depth vs. Branching Factor

The most general and distinct properties of game trees are their depth and their width, so the first property we investigate is the balance between the tree depth and the branching factor. These are properties that can quickly be estimated as simulations are run. With increasing depth the simulations become longer and therefore decrease the number of samples that make up the aggregated values at the root. Also longer simulations are in more danger of resulting in improbable lines of simulation play. Increasing the branching factor results in a wider tree, decreasing the proportion of lines of play tried. Relative to the number of nodes in the trees the depth and width can be varied, allowing us to answer the question if MCTS favors one over the other.

Progression

Some games progress towards a natural termination with every move made while other allow moves that maintain a sta-

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

Optimistic Moves

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

Empirical Evaluation

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

Setup

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

All games have a scoring interval of $[0, 1]$ and MCTS uses $C_p = 1/\sqrt{2}$ with an uniform random playout strategy. The node expansion strategy adds only the first new node encountered to the MCTS tree and neither a discount factor nor other modifiers are used in the back-propagation step. The players only deliberate during their own turn. A custom-made tool is used to create all games and agents. This tool allows games to be setup as FEN strings¹ for boards of any size and by extending the notation one can select from custom predefined piece types. Additional parameters are used to set game options like goals (capture all opponents or reach the back rank of the opponent), artificial termination depth and scoring policy, and whether squares can inflict penalty points.

¹Forsyth-Edwards Notation. <http://en.wikipedia.org/wiki/FEN>

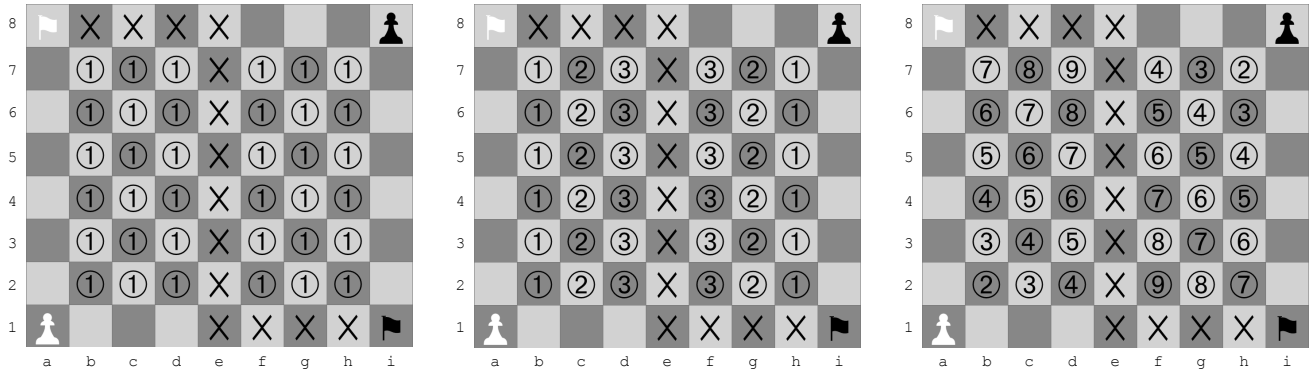


Figure 1: (a)Penalties Game (b)Shock Step Game (c)Punishment Game

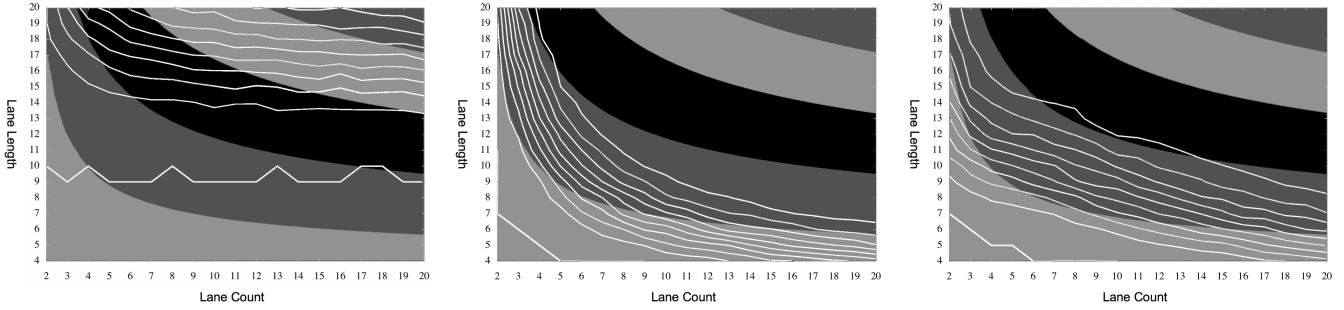


Figure 2: (a)Penalties Results (b)Shock Step Results (c)Punishment results

Tree Depth vs. Branching Factor

The games created for this experiment can be thought of as navigating runners through an obstacle course where the obstacles inflict penalty points. We experimented with three different setups for the penalties as shown in Figure 1. The pawns are the runners, the corresponding colored flags their goal and the big X's walls that the runners cannot go through. The numbered squares indicate the penalty inflicted when stepped on. White and Black each control a single runner that can take one step forward each turn. The board is divided by the walls so the runners will never collide with each other. For every step the runner takes the players can additionally select to make the runner move to any other lane on their side of the wall. For example, on its first move in the setups in Figure 1 White could choose from the moves $a1-a2$, $a1-b2$, $a1-c2$ and $a1-d2$. All but one of the lanes available to each player incur one or more penalty points. The game is set up as a turn taking game but both players must make equal amount of moves and therefore both will have reached the goal before the game terminates. This helps in keeping the size of the tree more constant. The winner is the one that has fewer penalty points upon game termination. The optimal play for White is to always move on lane a , resulting in finishing with no penalty points, while for Black the optimal lane is always the one furthest to the right. This game setup allows the depth of the tree to be tuned by setting the lanes to a different length. The branching factor is tuned through the number of lanes per player. To ensure

that the amount of tree nodes does not collapse with all the transpositions possible in this game, the game engine produces state ids that depend on the path taken to the state it represents. Therefore states that are identical will be perceived as different ones by the MCTS algorithm if reached through different paths. This state id scheme was only used for the experiments in this subsection.

The first game we call Penalties and can be seen in Figure 1 (a). Here all lanes except for the safe one have all steps giving a penalty of one. The second one we call Shock Step and is depicted in Figure 1 (b). Now each non-safe lane has the same amount of penalty in every step but this penalty is equal to the distance from the safe lane. The third one called Punishment is shown in Figure 1 (c). The penalty amount is now as in the Shock Step game except now it gets progressively larger the further the runner has advanced.

We set up races for the three games with all combinations of lanes of length 4 to 20 squares and number of lanes from 2 to 20. We ran 1000 games for each data point. MCTS runs all races as White against an optimal opponent that always selects the move that will traverse the course without any penalties. MCTS was allowed 5000 node expansions per move for all setups. The results from these experiments are shown in Figure 2. The background depicts the trend in how many nodes there are in the game trees related to number of lanes and their length. The borders where the shaded areas meet are node equivalent lines, that is, along each border all points represent the same node count. When moving from

the bottom left corner towards the top right one we are increasing the node count exponentially. The lines, called win lines, overlaid are the data points gathered from running the MCTS experiments. The line closest to the bottom left corner represent the 50% win border (remember the opponent is perfect and a draw is the best MCTS can get). Each border-line after that shows a 5% lower win ratio from the previous one. This means that if MCTS only cares how many nodes there are in the game tree and its shape has no bearing on the outcome, then the win lines should follow the trend of the background plot exactly.

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

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

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

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

Progression

For experimenting with the progression property we created a racing game similar to the one used in the tree depth vs. width experiments. Here, however, the size of the board is kept constant (20 lanes $\times$ 10 rows) and the runners are confined to their original lane by not being allowed to move sideways. Each player, White and Black, has two types of runners, ten in total, initially set up as shown in Figure 3. The former type, named *active* runner and depicted as a pawn, moves one step forward when played whereas the second, named *inactive* runner and depicted by circular arrows, stays on its original square when played. In the context of

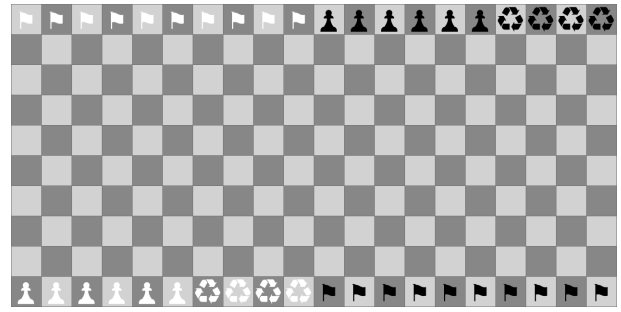


Figure 3: Progression Game

GGP each inactive runner has only a single *noop* move available for play. By changing the ratio between runner types a player has, one can alter the progression property of the game: the more active runners there are, the faster the game progresses (given imperfect play). In the example shown in the figure each player has 6 active and 4 inactive runners. The game terminates with a win once a player's runner reaches a goal square (a square with the same colored flag).

We also impose an upper limit on the number of moves a game can last. A game is terminated artificially and scored as a tie if neither player has reached a goal within the upper limit of moves. By changing the limit one can affect the progression property of the game: the longer a game is allowed to last the more likely it is to end in a naturally resulting goal rather than being depth terminated, thus progressing better. We modify this upper limit of moves in fixed step sizes of 18, which is the minimum number of moves it takes Black to reach a goal (Black can first reach a flag on its 9th move, which is the 18th move of the game as White goes first). A *depth factor* of one thus represents an upper limit of 18 moves, depth factor of two 36 moves, etc.

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

The result is shown in Figure 4, with the winning percentage of MCTS plotted against both the depth factor (left) and percentage of simulations ending naturally (right). Each curve represents a game setup using different number of active runners.² The overall shape of both plots show the same trend, reinforcing that changing the depth factor is a good model for indirectly altering the number of simulations that terminate naturally (which is not easy to change directly in our game setup). When looking at each curve in an isolation we see that as the depth factor increases, so does MCTS's performance initially, but then it starts to decrease again. Increasing the depth factor means longer, and thus fewer, simulations because the number of node expansions per move

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

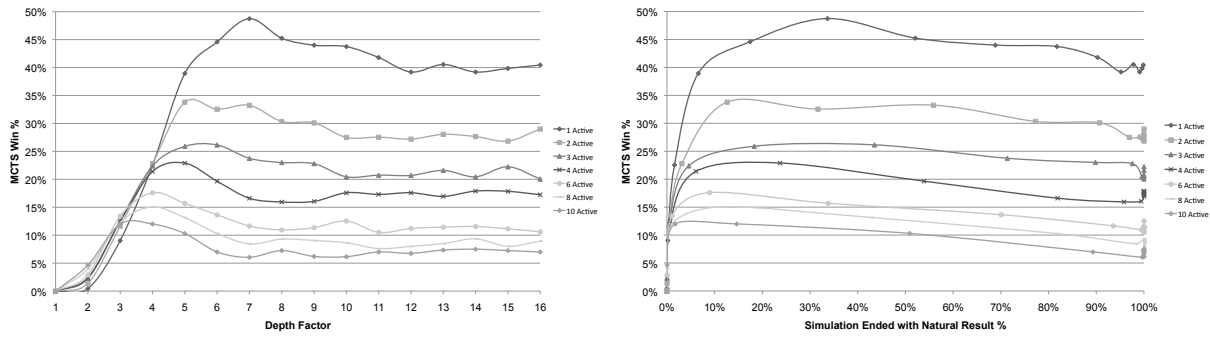


Figure 4: Progression Depth Factor: Fixed Node Expansion Count

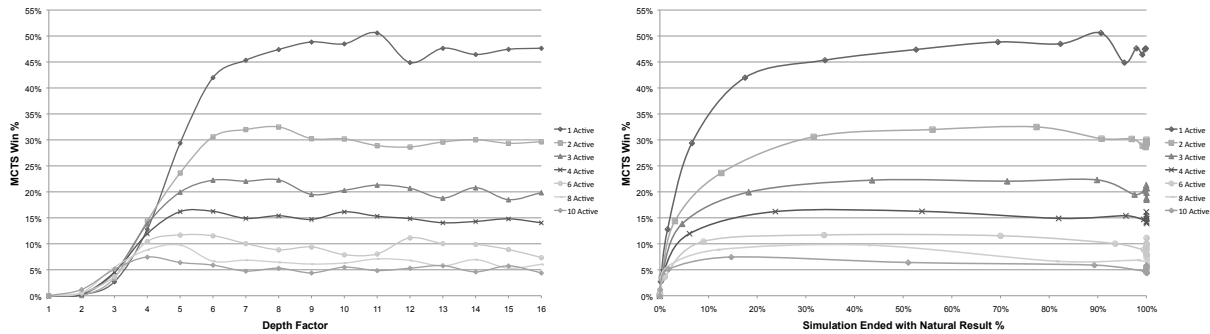


Figure 5: Progression Depth Factor: Fixed Simulation Count

is fixed. The decremental effect can thus be explained by fewer simulations. This is better seen in Figure 5 where the result of identical experiments as in the previous figure is given, except now the number of simulations—as opposed to node expansions—is kept fixed (at 1000).

The above results show that progression is an important property for MCTS, however, what is somewhat surprising is how quickly MCTS’s performance improves as the percentage of simulations ending at true terminal states goes up. In our testbed it already reaches close to peak performance as early as 30%. This shows promise for MCTS even in games where most paths may be non-progressive, as long as a somewhat healthy ratio of the simulations terminate in useful game outcomes. Additionally, in GGP one could take advantage of this in games where many lines end with the step counter reaching the upper limit, by curtailing the simulations even earlier. Although this would result in a somewhat lower ratio of simulations returning useful game outcomes, it would result in more simulations potentially resulting in a better quality tradeoff (as in Figure 4).

We can see the effects of changing the other dimension — number of active runners a player has— by contrasting the different curves in the plots. As the number of active runners increases, so does the percentage of simulations ending in true terminal game outcomes, however, instead of resulting in an improved performance, it decreases sharply. This performance drop is seen clearly in Figure 6 when plotted against the number of active runners (for demonstration, only a single depth factor curve is shown). This behavior,

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

Optimistic Moves

For this experiment we observe how MCTS handles a position in a special variation of Breakthrough which accentuates this property. Breakthrough is a turn-taking game played with pawns that can only move one square at a time either straight or diagonally forward. When moving diagonally they are allowed to capture an opponent pawn should one reside on the square they are moving onto. The player who is first to move a pawn onto the opponent’s back rank wins. The variation and the position we set up is shown in Figure 7. The big X’s are walls that the pawns cannot move onto. There is a clear cut winning strategy for White on

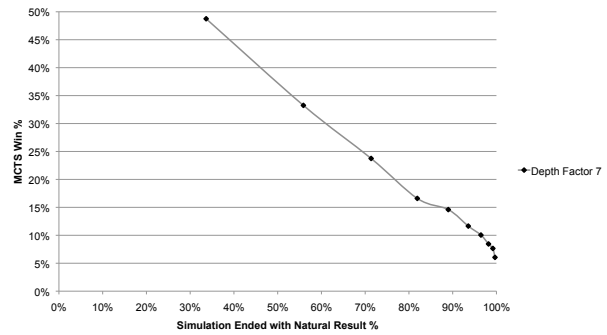
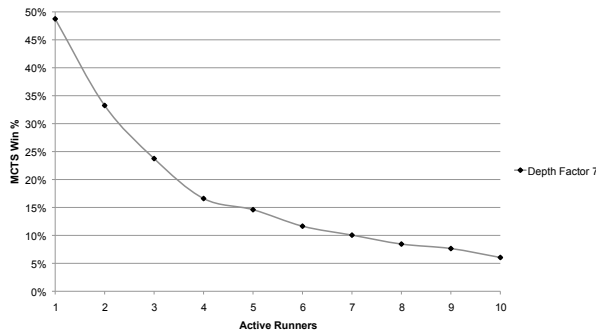


Figure 6: Progression Active Runners: Fixed Node Expansion Count

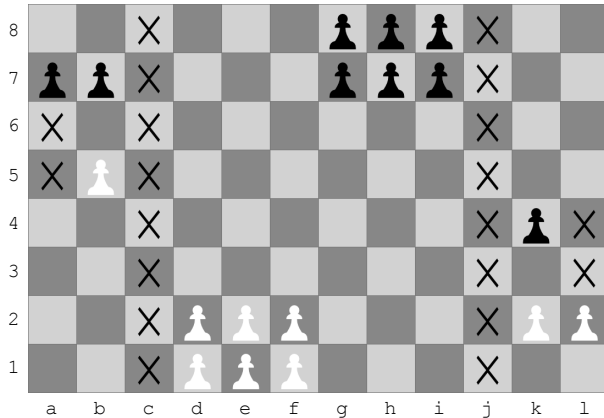


Figure 7: Optimistic Moves Game

the board, namely moving any of the pawns in the midfield on the second rank along the wall to their left. The opponent only has enough moves to intercept with a single pawn which is not enough to prevent losing. This position has also built-in pitfalls presented by an optimistic move, for both White and Black, because of the setups on the *a* and *b* files and *k* and *l* files, respectively. For example, if White moves the *b* pawn forward he threatens to win against all but one Black reply. That is, capturing the pawn on *a7* and then win by stepping on the opponent's back rank. This move is optimistic because naturally the opponent responds right away by capturing the pawn and in addition, the opponent now has a guaranteed win if he keeps moving the capturing pawn forward from now on. Similar setup exists on the *k* file for Black. Still since it is one ply deeper in the tree it should not influence White before he deals with his own optimistic move. Yet it is much closer in the game tree than the actual best moves on the board.

We ran experiments showing what MCTS considered the best move after various amount of node expansions. We combined this with four setups with decreased branching factor. The branching factor was decreased by removing pawns from the middle section. The pawn setups used were the ones shown in Figure 7, one with the all pawns removed from files *f* and *g*, one by additionally removing all pawns

from files *e* and *h* and finally one where the midfield only contained the pawns on *d2* and *i7*. The results are in Table 1 and the row named "Six Pawns" refers to the setup in Figure 7, that is, each player has six pawns in the midfield and so on. The columns then show the three most frequently selected moves after 1000 tries and how often they were selected by MCTS at the end of deliberation. The headers show the expansion counts given for move deliberation.

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

Clearly the bigger branching factors multiply up this problem. The simulations can be put to much better use if this problem could be avoided by pruning these optimistic moves early on. The discovery process of avoiding these moves can be sped up by more greedy simulations or biasing the playouts towards the (seemingly) winning moves when they are first discovered. Two general method of doing so are the MAST (Finnsson and Björnsson 2008) and RAVE (Gelly and Silver 2007) techniques, but much bigger improvements could be made if these moves could be identified when they are first encountered and from then on completely ignored.

Related Work

Comparison between Monte-Carlo and $\alpha\beta$ methods was done in (Clune 2008). There the author conjectures that $\alpha\beta$ methods do best compared to MCTS when: (1)The heuristic evaluation function is both stable and accurate, (2)The game is two-player, (3) The game is turn-taking, (4) The game is

Table 1: Optimistic Moves Results

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

zero-sum and (5) The branching factor is relatively low. Experiments using both real and randomly generated synthetic games are then administered to show that the further you deviate from these settings the better Monte-Carlo does in relation to $\alpha\beta$.

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

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

Conclusions and Future Work

In this paper we tried to gain insight into factors that influence MCTS performance by investigating how three different general game-tree properties influence its strength.

We found that it depends on the game itself whether MCTS prefers deep trees, big branching factor, or a balance between the two. Apparently small nuances in game rules and scoring systems, may alter the preferred game-tree structure. Consequently it is hard to generalize much about MCTS performance based on game tree depth and width. Progression is important to MCTS. However, our results suggests that MCTS may also be applied successfully in slow progressing games, as long as a relatively small percentage of the simulations provide useful outcomes. In GGP games one could potentially take advantage of how low ratio of real outcomes are needed, by curtailing potentially fruitless simulations early, thus increasing simulation throughput. Hints of MCTS having difficulty in committing to a

strategy when faced with many good ones were also discovered. Optimistic Moves are a real problem for MCTS that escalates with an increased branching factor.

For future work we want to come up with methods for MCTS that help it in identifying these properties on the fly and take measures that either exploit or counteract what is discovered. This could be in the form new extensions, pruning techniques or even parameter tuning of known extension. Also more research needs to be done regarding the possible MCTS strategy commitment issues.

Acknowledgments

This research was supported by grants from The Icelandic Centre for Research (RANNÍS).

References

- Clune, J. E. 2008. *Heuristic Evaluation Functions for General Game Playing*. PhD dissertation, University of California, Los Angeles, Department of Computer Science.
- Coulom, R. 2006. Efficient selectivity and backup operators in Monte-Carlo tree search. In *The 5th International Conference on Computers and Games (CG2006)*, 72–83.
- Enzenberger, M., and Müller, M. 2009. 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.
- Finnsson, H., and Björnsson, Y. 2008. Simulation-based approach to general game playing. In Fox, D., and Gomes, C. P., eds., *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, 259–264. AAAI Press.
- Gelly, S., and Silver, D. 2007. Combining online and offline knowledge in UCT. In Ghahramani, Z., ed., *ICML*, volume 227 of *ACM International Conference Proceeding Series*, 273–280. ACM.
- Gelly, S.; Wang, Y.; Munos, R.; and Teytaud, O. 2006. Modification of UCT with patterns in Monte-Carlo Go. Technical Report 6062, INRIA.

- Kocsis, L., and Szepesvári, C. 2006. Bandit based Monte-Carlo planning. In *European Conference on Machine Learning (ECML)*, 282–293.
- Long, J. R.; Sturtevant, N. R.; Buro, M.; and Furtak, T. 2010. Understanding the success of perfect information monte carlo sampling in game tree search. In Fox, M., and Poole, D., eds., *AAAI*. AAAI Press.
- Lorentz, R. J. 2008. Amazons discover monte-carlo. In *Proceedings of the 6th international conference on Computers and Games, CG '08*, 13–24. Berlin, Heidelberg: Springer-Verlag.
- Ramanujan, R.; Sabharwal, A.; and Selman, B. 2010. On adversarial search spaces and sampling-based planning. In *ICAPS'10*, 242–245.
- Sturtevant, N. R. 2008. An analysis of uct in multi-player games. In van den Herik, H. J.; Xu, X.; Ma, Z.; and Winands, M. H. M., eds., *Computers and Games*, volume 5131 of *Lecture Notes in Computer Science*, 37–49. Springer.
- Szita, I.; Chaslot, G.; and Spronck, P. 2009. Monte-carlo tree search in settlers of catan. In van den Herik, H. J., and Spronck, P., eds., *ACG*, volume 6048 of *Lecture Notes in Computer Science*, 21–32. Springer.
- Winands, M. H. M.; Björnsson, Y.; and Saito, J.-T. 2010. Monte carlo tree search in lines of action. *IEEE Trans. Comput. Intellig. and AI in Games* 2(4):239–250.

Neural Networks for High-Resolution State Evaluation in General Game Playing

Daniel Michulke

Department of Computer Science
Dresden University of Technology
daniel.michulke@mailbox.tu-dresden.de

Abstract

$C - IL^2P$ is an algorithm that transforms a propositional domain theory to a neural network that correctly represents the domain theory and is ready-to-use without prior training. Its original intention was to transform explicit symbolic knowledge into a neural network to allow for learning.

The game playing agent system presented in (Michulke and Thielscher 2009) uses the algorithm differently: By transforming the symbolic description of the goal of a game to a neural network it obtains an evaluation function for states of that game. Much like fuzzy logic, the network can be used for graded inference while retaining correctness. But in contrast to fuzzy logic, the network is able to learn and may consequently improve with experience, which is unique among competing agents and arguably an important advantage in a game playing setting. However, since not intended for this use, the transformation algorithm produces networks that cannot correctly represent complex domain theories without losing their ability to distinguish some input vectors that ought to have a different evaluation.

In this paper we introduce a generalization of $C - IL^2P$ that addresses the above issue. It structures the formerly monolithic approach of logic-to-network transformation to allow for lower weights in the network. This increases the output resolution by several orders of magnitude, as experiments demonstrate, while maintaining correctness.

1 Introduction

$C - IL^2P$ (d'Avila Garcez and Gabbay 2002) is an algorithm that transforms a set of propositional rules to a neural network. In this way, it is possible to create a neural network from a propositional domain theory without prior training or search for good initialization parameters. At the same time the neural network correctly represents the domain theory and can be trained using standard algorithms such as backpropagation.

A different application of such a network is as an inference tool that works on real numbers representing truth values. The network input is a vector of truth values and the network evaluates to what extent the domain theory holds on the input. This approach is used in (Michulke and Thielscher 2009) to generate a state evaluation function for a general game playing agent. The propositionalized goal of the game (as stated in a game rules) is considered the domain theory and a game state the input. The output of the network for any

given state corresponds to the output of the goal definition of the game and is therefore useful for distinguishing goal states from non-goal states. Moreover, the output is at same time monotonic, meaning that states that correspond more to the goal of the game produce a higher output. The network thus extrapolates from goal states to non-goal states and is a good evaluation function that can be used to guide search to the most favorable among the states searched.

Additionally, it can also be trained, e.g. using standard backpropagation, which especially in the game playing environment is an important advantage. However, in their paper they also state that $C - IL^2P$ sets tight constraints on weights in order to ensure correctness. A by-product of this is that the network loses its ability to distinguish states from each other under certain circumstances.

For this reason we present a generalization of $C - IL^2P$ that imposes much weaker constraints on the weight settings than its algorithmic predecessor. As a result, much more complex domain theories can be represented by the resulting network while at the same time the output of the network is descriptive (different states produce different outputs) and correct. We demonstrate the benefit of our new approach in the domain of General Game Playing where experiments in more than 160 games show that the ability to correctly distinguish states from each other increases by several orders of magnitude when compared to $C - IL^2P$.

Note that the approach is *not* a fuzzy inference engine in the strict sense. The constructed networks do, however, possess important properties such as correctness and monotonicity and therefore behave similar to fuzzy inference engines.

The remainder of this paper is organized as follows: In the next section we give the necessary background for this paper and show a simple case where a network constructed using $C - IL^2P$ cannot distinguish between different inputs. In section 3 we introduce a structured approach on rule-to-neuron transformation and, based on this, present an algorithmic description of our approach in 4. Finally, we demonstrate that our new approach improves the abilities of neural networks to represent complex domain theories (section 5) and conclude in section 6.

2 Background

2.1 State Evaluation in General Game Playing

In General Game Playing (GGP), the rules of a game are given minutes before the corresponding match starts. These rules are encoded in the Game Description Language (GDL, (Love et al. 2008)) and define the roles (e.g. white and black in Chess), initial state, legal moves, state transitions, terminal states and goals for each role. Agents thus cannot rely on any preprogrammed behaviors but have to autonomously develop a strategy for the game.

A core part of the strategy is the state evaluation function. It maps states to values and indicates how good a state is. In any state, the agent then takes the move that leads to the state with maximum state value. While there are several agents with different approaches on what to use as an evaluation function, the approach of (Schiffel and Thielscher 2007), called Fluxplayer, seems particularly intriguing as it does not exhibit the problems of other approaches such as assuming opponent random behavior, as e.g. in (Finnsson and Björnsson 2008), or time-consuming feature guessing and evaluation mechanisms, as in (Clune 2007).

Instead, it derives an evaluation function directly from the goal conditions of the game. These goal conditions are predicate logic formulas that are transformed by first ground-instantiating them and then evaluating the resulting propositional logic formulas using t-norm fuzzy logics. The approach is successful with Fluxplayer constantly being among the top 4 agents in the annual worldwide championship in GGP.

A similar approach by (Michulke and Thielscher 2009) uses neural networks instead of t-norm fuzzy logics, based on the idea that its ability to learn should prove an important edge against other agents, given that human-level intelligence builds on both, logic and learning. This article deals with the resolution problem outlined in their GGP agent system and shows how the problem's impact can be greatly reduced. While there are other possibilities to address the resolution problem (e.g. using high-precision arithmetic), we know of none that does not imply higher computational cost when used for evaluation. This, however, is prohibitive in a game playing setting where the time needed for state evaluation is critical property of the algorithm.

2.2 Neuro-Symbolic Integration

Neuro-Symbolic Integration (NSI) was originally intended to unify neural networks and symbolic representations of knowledge to combine the capability to learn of neural networks with the logic reasoning facilities of an explicit representation. It is basically concerned with encoding a domain theory into a network, training the network and possibly extracting knowledge afterwards again. This process constitutes a cycle that enables transforming both systems into each other, see (Bader and Hitzler 2005) for an overview.

In the light of our different application of NSI concepts, KBANN (Knowledge-Based Artificial Neural Network, (Towell, Shavlik, and Noordenier 1990)) is the only comparable algorithm. It transforms an "approximately correct" propositional domain theory to a unipolar neural net-

work. Training was then used to correct the errors in the initial theory. When applied as classifier, KBANN learned faster and qualitatively performed better than other learning algorithms. However, the theoretical framework restricted its application to domain theories with only small numbers of rules and antecedents per rule.

$C - IL^2P$ (d'Avila Garcez and Gabbay 2002) addresses this problem using bipolar networks, but the networks still require weights too high to be used for correctly fuzzy-evaluating inputs while at the same time retaining the ability to distinguish among all possible inputs.

2.3 The $C - IL^2P$ algorithm

$C - IL^2P$ represents the propositional values for truth and falsity as intervals of the output o of a bipolar neuron. If a propositional variable is true, the corresponding neuron gives a value of $o \in [A_{min}, 1]$ while an output $o \in [-1, A_{max}]$ is interpreted as falsity. Outputs $o < -1$ or $o > 1$ lie outside of the co-domain of the activation function $h(x) = \frac{2}{1+e^{-x}} - 1$ while outputs in the non-empty interval (A_{max}, A_{min}) are guaranteed to not occur by restricting the weights of connections to the neuron (parameter W , equation (4)). Usually, $-A_{max} = A_{min}$ is set, allowing propositional negation to be encoded as arithmetic negation. For simplicity we describe the algorithm in a slightly modified form. $C - IL^2P$ transforms rules of the form

$$q \Leftarrow \bigotimes_{1 \leq i \leq k} p_i \text{ with } \bigotimes \in \{\wedge, \vee\} \quad (1)$$

where q is an atom and the p_i are positive or negative literals. A rule is represented by $k + 1$ neurons where k neurons represent the literals p_i that have an outgoing connection to the $k + 1$ st neuron representing the head. Negative literals are represented just as positive literals, but with their connection weight negated. In addition to these k connections, a connection to the bias unit (which has constant output 1) is added that works as a threshold θ . The weight of this connection is set depending on the number k of children and the operator $\bigotimes \in \{\wedge, \vee\}$:

$$\theta_{\wedge}(k) = -\theta_{\vee}(k) = \frac{(1 + A_{min}) * (1 - k)}{2} * W \quad (2)$$

The parameter A_{min} determines the truth threshold above which a neural output value represents truth and W the standard weight for any connection within the neural network. Both can be chosen freely, but are subject to the following restrictions:

$$1 > A_{min} > \frac{k_{max} - 1}{k_{max} + 1} \quad (3)$$

$$W \geq 2 * \frac{\ln(1 + A_{min}) - \ln(1 - A_{min})}{k_{max}(A_{min} - 1) + A_{min} + 1} \quad (4)$$

k_{max} is the maximum number of antecedents a rule has.

$C - IL^2P$ for Fuzzy Evaluation Due to the monotonicity of the activation function, an input that corresponds better to the domain theory also produces a higher output of the network. Given that the network at the same time correctly represents the domain theory, it can be used for fuzzy inference.

However, with the steepness $h'(x) = 1 - h(x)^2$ of the activation function approximating zero for high absolute values of x , neurons encoding a conjunction (disjunction) with few (many) fulfilled antecedents may produce the same network output as the following example demonstrates:

Example 1. Consider a neuron z representing a conjunction of the output of 4 preceding neurons $z \Leftarrow \bigwedge_{i=1}^4 y_i$. We assume the maximal number of children of a node to be $k = 4$ and set $A_{min} = -A_{max} = 0.9 > 0.6$ and $W = 4 \geq 3.93$, fulfilling thus equations (3) and (4). The following table shows the neural activation $a_z = W * (\theta_{\wedge}(k) + t)$ and output $o_z = h(a_z)$ of neuron z for t of the 4 y_i representing *true* (having output 1).

true antecedents t	activation a_z	output o_z
0	-27.4	-1.000000
1	-19.4	-1.000000
2	-11.4	-0.999978
3	-3.4	-0.935409
4	4.6	0.980096

While the neuron correctly encodes a conjunction, it is not able to distinguish the cases where none or one antecedent is fulfilled. This is due to the resolution imposed (10^{-6}), but occurs similarly in all finite floating-point representations such as 32-bit computers. This lack of distinction becomes worse when the output values are propagated through the network.

The main reason for this behavior is the weight restriction in equation (4). The higher the absolute weights are, the more likely the activation of a neuron is far away from zero. This is a consequence of the global setting of W and A_{min} that depend on the maximum number of antecedents of a rule k_{max} and implies weights higher than necessary for nodes with $k < k_{max}$ children. Moreover, the minimum and maximum output values of each neuron are fixed to -1 and 1 . Possible lower values are thus not considered for weight determination.

3 Local Neuron Transformation

We reduce these effects in our more general version of $C - IL^2P$. We begin by defining a *standard neuron*.

Definition 1 (Standard Neuron). Let z be an input layer neuron with no predecessors and the output value $o_z \in O \subset [-1, 1]$ OR a neuron with

- a real weight w_z ,
- a real bias $bias_z$,
- and the unbiased input $\hat{i}_z = \sum_{y \in pred(z)} o_y$ where $pred(z) \neq \emptyset$ is the non-empty set of preceding neurons to z ,
- the biased input $i_z = \hat{i}_z + bias_z$,
- the bipolar activation function $h(x) = \frac{2}{1+e^{-x}} - 1$,
- the output $o_z = h(w_z * i_z)$.

Then we call z a standard neuron.

Like in $C - IL^2P$, we represent truth and falsity of a propositional variable by a neuron with an output value in a specified interval. However, instead of using the intervals

$[-1, A_{max}]$ and $[A_{min}, 1]$ with A_{max} and A_{min} as global parameters, we generalize by parameterizing the minimum false and maximum true output value (-1 and 1 respectively) and by setting all parameters *locally*, that is, for each neuron individually. Let a propositional variable q be represented by a neuron z with output o_z . Then we define:

$$q \Leftrightarrow o_z \in [o_z^+, o_z^{++}] \quad \neg q \Leftrightarrow o_z \in [o_z^{--}, o_z^-]$$

We will use the term *limits* to refer to the quadruple $(o_z^{--}, o_z^-, o_z^+, o_z^{++})$ of output parameters of a neuron z .

In order to interpret a neuron output value as a propositional variable, we must ensure that the output intervals for *true* and *false*, are distinct, that is, they do not overlap $o_z^- < o_z^+$. We furthermore constrain propositional truth to positive values and falsity to negative values $o_z^- < 0 < o_z^+$. This will allow us later to define propositional negation as arithmetic negation.

We call a neuron fulfilling this constraint *representative*.

Definition 2 (Representative Neuron). Let z be a standard neuron with the output value o_z .

Then we call z representative if the set of possible output values $O_z = \{o_z\}$ is identical to the set of real values $O_z = [o_z^{--}, o_z^-] \cup [o_z^+, o_z^{++}]$ and $o_z^- < 0 < o_z^+$.

With $o_z^{--} \leq o_z^-$ and $o_z^+ \leq o_z^{++}$ the two intervals of a representative neuron z are not empty: $[o_z^{--}, o_z^-] \neq \emptyset \neq [o_z^+, o_z^{++}]$. Along with the output range $[-1, 1]$ of the activation function the following inequalities hold:

$$-1 \leq o_z^{--} \leq o_z^- < 0 < o_z^+ \leq o_z^{++} \leq 1$$

Note that any neuron is representative regardless of its absolute weight: Since from $o_z^- < 0 < o_z^+$ follows $h(w_z * i_z^-) < 0 < h(w_z * i_z^+)$ and $h(x)$ is an odd function ($h(-x) = -h(x)$), a representative neuron remains representative if we change its current weight w_z to another weight w'_z as long as the sign of the weight is the same $sgn(w_z) = sgn(w'_z)$. This enables us to set the weight freely without confusing truth values.

Our line of argumentation now goes as follows: Consider the rule $q \Leftarrow \bigotimes_{1 \leq i \leq k} p_i$ with $\bigotimes \in \{\wedge, \vee\}$. We assume that

the positive form of all literals p_i has already been translated to the neurons y_i and that these neurons $\{y_i : 0 \leq i \leq k\}$ are representative. Then we create a neuron z that represents q by doing the following:

Output-Input Mapping map the output values o_y of all neurons y to the input value $\hat{i}_z$ such that $\hat{i}_z \in [\hat{i}_z^+, \hat{i}_z^{++}] \Leftrightarrow q$ and $\hat{i}_z \in [\hat{i}_z^{--}, \hat{i}_z^-] \Leftrightarrow \neg q$

Distinct Input increase the weights w_y of all neurons y such that $\hat{i}_z^- < \hat{i}_z^+$

Representative Neuron set the bias $bias_z$ such that z is representative

We obtain a complete representation of the rule without having to impose a constraint on the weight w_z . Instead, we set the weights w_y of the predecessors $y \in pred(z)$. The three steps will be explained subsequently.

Output-Input Mapping The output-input mapping defines how to derive the input limits of a neuron z from a set of output limits of its predecessors $y \in \text{pred}(z)$. These limits determine the input intervals that represent truth and falsity depending on the type of rule (conjunction or disjunction) and the output limits of the preceding neurons.

$$\begin{aligned} \hat{i}_z^{++} &= \sum_{y \in \text{pred}(z)} o_y^{++} & \hat{i}_z^{--} &= \sum_{y \in \text{pred}(z)} o_y^{--} \\ \hat{i}_{z, \text{and}}^+ &= \sum_{y \in \text{pred}(z)} o_y^+ & \hat{i}_{z, \text{and}}^- &= \hat{i}_z^{++} - \min_{y \in \text{pred}(z)} o_y^{++} - o_y^- \\ \hat{i}_{z, \text{or}}^- &= \sum_{y \in \text{pred}(z)} o_y^- & \hat{i}_{z, \text{or}}^+ &= \hat{i}_z^{--} + \min_{y \in \text{pred}(z)} o_y^+ - o_y^{--} \end{aligned} \quad (5)$$

The limits are calculated according to four scenarios: The maximum truth and minimum false values $\hat{i}_z^{++}$ and $\hat{i}_z^{--}$ are simply the sum of the corresponding output values and can be considered the *best-case* values as they are the values with the maximum possible distance to the ambiguous value 0.

The *worst-case* values depend on the operator: Conjunctions represent a worst-case truth if their predecessors represent worst-case truth. However, they become false if all predecessors give even the best-case truth value with the exception of one that gives the maximum false (worst-case false) value. The truth values for disjunctions are calculated similarly.

Finally, negation is represented by arithmetical negation. If z represents a rule where the neuron y is a negative antecedent then substitute y by a neuron y' with the negated output limits:

$$\begin{aligned} o_{y'}^{++} &= -o_y^{--} & o_{y'}^+ &= -o_y^- \\ o_{y'}^- &= -o_y^{++} & o_{y'}^{--} &= -o_y^+ \end{aligned} \quad (6)$$

Note that neither weight nor bias of z are set at this stage but the propositional meaning defined instead.

Since the resulting intervals may overlap, i.e. $\hat{i}_z^- < \hat{i}_z^+$ cannot be guaranteed, we “separate” the two intervals by setting the weights w_y in the following step.

Distinct Input We show that a weight w_y for the predecessors y of a neuron z exists such that the input of z is distinct. We use the parameter $\Delta_z \stackrel{!}{=} \hat{i}_z^+ - \hat{i}_z^-$ to refer to the difference by which the input intervals of z are distinct. For simplicity, we assume $-i^- = i^+$, consequently it holds $-o^- = o^+$. This assumption will be justified in theorem 2.

Theorem 1. Let z be a standard neuron representing a conjunction or disjunction of its predecessors $y \in \text{pred}(z)$ as given in equation (5). Furthermore, let all predecessors be representative standard neurons with $-i^- = i^+$. Then for an arbitrary $\Delta_z < 2$ there exists a weight w_y for all neurons y such that the input of z is distinct by at least Δ_z , i.e. $\hat{i}_z^+ \geq \hat{i}_z^- + \Delta_z$.

$$w_y \geq -\frac{1}{i_{\min}^+} \ln \frac{2 - \Delta_z}{\Delta_z + 2k} \quad (7)$$

$\min$ refers to the neuron that has the minimum worst-case truth value $i_{\min}^+$ of all neurons $y \in \text{pred}(z)$.

Proof for Conjunctive Case (Sketch).

$$\begin{aligned} \hat{i}_z^+ &\stackrel{!}{\geq} \hat{i}_z^- + \Delta_z \\ \sum_{y \in \text{pred}(z)} o_y^+ &\geq \sum_{y \in \text{pred}(z)} o_y^{++} - \min_{y \in \text{pred}(z)} (o_y^{++} - o_y^-) + \Delta_z \\ k * \min_{y \in \text{pred}(z)} o_y^+ &\geq k - \min_{y \in \text{pred}(z)} (1 + o_y^+) + \Delta_z \\ (k+1) * (o_{\min}^+ - 1) &\geq \Delta_z - 2 \\ o_{\min}^+ &\geq \frac{\Delta_z - 2}{k+1} + 1 \\ w_{\min} &\geq -\frac{1}{i_{\min}^+} \ln \frac{2 - \Delta_z}{\Delta_z + 2k} \end{aligned}$$

Since $i_{\min}^+$ is smallest by definition, the weight bounds of any neuron $y \in \text{pred}(z)$ is less or equal to $w_{\min}$. $\square$

The proof is similar for the disjunctive case. The result is the worst-case lower bound for the weights w_y such that the input of z is distinct by at least Δ_z . In extreme cases, this bound corresponds to the weight bound in $C - IL^2P$. However, our algorithm calculates these parameters locally and benefits thus from any situation that does not represent a worst-case.

Besides, the result shows that $2 > \Delta$ is a precondition for all successor neurons, ensuring thereby that Δ is smaller than the maximum possible output difference of a neuron $o^{++} - o^{--} = 2$.

Representative Neurons Now we just set the bias such that z itself is representative.

Theorem 2. Let z be a standard neuron with distinct input. By setting

$$\text{bias}_z = \frac{-\hat{i}_z^- - \hat{i}_z^+}{2} \quad (8)$$

z is representative.

Proof(Sketch). It holds $i_z^- = -i_z^+ \neq 0$. Since the activation function $h(x)$ is odd and strictly monotonically increasing for $w > 0$, the input interval directly translates to a representative output interval. $\square$

Note that by setting the bias in between the limits $\hat{i}_z^-$ and $\hat{i}_z^+$, we obtain $-i^- = i^+$ and $-o^- = o^+$.

4 Transformation Algorithm

A basic algorithm can now be constructed that translates a rule of the form $q \Leftarrow \bigotimes_{1 \leq i \leq k} p_i$ with $\bigotimes \in \{\wedge, \vee\}$ to a

neuron. We consider the p_i already transformed to the neurons y_i . Thus their biased input limits (i_{y_i}) are set while the weights w_{y_i} are not. The algorithm sets these weights w_y for each y and the input limit $\hat{i}_z$ by doing the following:

1. calculate the output limits o_y of each predecessor neuron $y \in \text{pred}(z)$ for a weight w_y

2. calculate the input limits $\hat{i}_z$ (equation (5))
3. if the input limit is distinct then calculate the bias (equation (8)), else increase w_y and return to step 1

The algorithm can be applied to generate a complete network by calling it on a set of rules. When transforming to a neuron z , it sets the $bias_z$ and the input limits i_z together with the weights w_y of its preceding neurons. The transformation works thus simultaneously on two levels of the network and we must pay attention to the input layer and output layer neurons. Since output layer neurons have no successor, they need not guarantee an output distinct by Δ and their weight can thus be set to $w = 1$. On the other hand, input layer neurons need no bias and weight settings as they have well-defined output values.

4.1 Soundness

Since all input layer neurons are defined such that they are representative and we set the bias of each neuron such that it is representative it holds that every neuron is representative. To show that the transformation of a neuron is sound, we show that the output of a neuron z correctly represents a conjunction or disjunction of the variables represented by its predecessor neurons.

Theorem 3. If the neurons y_i represent the variables p_i in the rule $q \leftarrow \bigotimes_{1 \leq i \leq k} p_i$ with $\bigotimes \in \{\wedge, \vee\}$ then we can construct a neuron z such that it represents q .

Proof. Due to the monotonicity of the activation function, we limit the proof to border cases. The proof for the disjunction is omitted as it is similar.

We show that if all predecessor neurons represent *true*, the conjunction neuron z also represents *true*: $\forall y \in pred(z) : o_y \geq o_y^+ \rightarrow o_z \geq o_z^+$. It follows:

$$\sum_y o_y \geq \sum_y o_y^+ = \hat{i}_z^+ \\ \hat{i}_z \geq \hat{i}_z^+$$

As all neurons y are representative, we set a positive weight such that $\hat{i}_z^+ > \hat{i}_z^-$, and from monotonicity of the activation function $h(x)$ follows that $o_z \geq o_z^+$.

If one predecessor neuron y^* represents *false*, the conjunction neuron z also represents *false*: $\exists y^* \in pred(z) : o_{y^*} \leq o_{y^*}^- \rightarrow o_z < o_{y^*}^-$. The maximum unbiased input $\hat{i}_{z,max}$ is then

$$\hat{i}_{z,max} = \sum_{y \in pred(z)} o_y^{++} - o_{y^*}^{++} - o_{y^*}^- \\ \hat{i}_{z,max} \leq \sum_{y \in pred(z)} o_y^{++} - \min_{y \in pred(z)} o_y^{++} - o_y^- = \hat{i}_z^-$$

Again, we set a positive weight w_y such that $\hat{i}_z^+ > \hat{i}_z^-$, and from monotonicity follows that $o_z \leq o_z^-$. $\square$

4.2 Open Problems

When transforming a graph to a network, the weights of a node y are processed by all of its successors and might be increased. If a node z sets the weight w_y of one of his predecessors and later a sibling z' of z increases this weight w_y , the absolute output values of y are increased and might decrease the difference between i_z^- and i_z^+ under certain circumstances.

There is no straightforward way of handling this problem due to the setting of w_y and the input limits i_z for each neuron z . The neuron transformation therefore implicitly ranges over two levels of the neural network. The simplest way to handle the scenario is to recalculate the neuron parameters for all previously transformed parent nodes of y and their successors if the weight w_y is increased. This can be time-consuming in the worst-case. Therefore we propose a heuristics that transforms the rules with the highest number of antecedents first as they are more likely to lead to higher weights. In addition, if the propositional domain theory has a layered structure (e.g. when in Disjunctive Normal Form), it is possible to transform it layer by layer, starting with the lowest. Once a layer is transformed, the weights of the preceding layer need not be recomputed anymore.

A second issue is that, unlike the original $C - IL^2P$, our approach is not directly designed to work on cyclic domain theories such as $a \Leftarrow b, b \Leftarrow a$. Although we consider such a scenario unlikely, cyclic domains do not invalidate our theoretical foundation of a rule-to-neuron transformation. Instead, the top-level algorithm would have to be modified, and one might e.g. opt to start from a $C - IL^2P$ constructed network and gradually reduce weights and see whether neurons remain representative and their input distinct by Δ_z .

In these scenarios, however, we consider it unlikely that resolution and the capability to distinguish between different input vectors play an important role. However, in case they do, simple modifications to the transformation algorithm

5 Experiments

While having proved that our algorithm is correct and as such suitable for translating a set of propositional rules to a neural network, we now demonstrate that it gives a higher output resolution than standard $C - IL^2P$. We do this in the domain of General Game Playing since it is our intended domain of application. Recall that due to the problem outlined in Example 1 different input vectors (in GGP: game states) are mapped to the same value, though one state fulfills more propositions of the domain theory (in GGP: the goal function) than the other.

Therefore we tested the networks generated by standard $C - IL^2P$ and by our generalized version on (by the time of the evaluation) 197 valid game descriptions submitted to the GGP server at Dresden University of Technology¹. 36 of these games have a trivial goal condition without conjunctions or disjunctions, consequently no network can be constructed. We transformed the goal description of the remaining 161 games to a propositional domain theory using

¹<http://euklid.inf.tu-dresden.de:8180/ggpserver>

the same techniques as in (Michulke and Thielscher 2009). We then transformed this domain theory to neural networks and evaluated all states of the game that are reachable from the initial state by one move. Our hypotheses are that 1. the networks constructed with our algorithm evaluate these states to a bigger output interval than $C - IL^2P$ and 2. that our approach can distinguish more of these states from each other. The latter is a direct consequence of the first hypothesis as a higher resolution implies a better treatment of situations as given in example 1.

To avoid superposition of outputs of different networks in the evaluation function, we evaluate a state by using only the output of the network of the first role in the game description and only for the goal condition winning the game. In this way, we keep results comparable for games that have a higher number of roles and goals. Figure 5 depicts our evaluation results. The x-axis represents the number of the game name in alphabetical order. Though game names have no linear relationship, we chose a line chart as different variants of the same game often have an alphabetically similar name.

Bigger Output Interval In the lower part of figure 5 we see the logarithm (base 10) of the size of the output interval of all 161 nontrivial games. We determine the size of the output interval by evaluating all states of depth 1 with our network and subtracting the minimum value from the maximum value obtained on these states. If minimum and maximum are equal, we consider (for visual reasons) the size of the output interval to be the machine precision, that is $2^{-53} \approx 10^{-16}$ for 64 bit floating-point precision. Since the network gives an output in the interval $[-1, 1]$, the logarithm is mostly negative. We can see that the resolution of our version (gray dotted line) is higher in 81 games and equal in 77 games to that of $C - IL^2P$. In three games, the precision is lower which is a side-effect of our lowest-weight approach in networks that consists of exactly one rule. In these cases, however, we could set an arbitrary weight, since our approach does not impose any constraints on the weight of connections to the output neuron.

The 77 games with equal resolution are games where no atomic part of the goal is affected by the first move. An example is the game Checkers where the winner is determined in terms of the number of pieces of both players. As it is impossible to remove any stone in the first move, all states are evaluated to the same value.

The logarithmic output size over all games increases in average from -12.92 to -10.13 , while for all games with an output size bigger than machine precision it increases from -9.91 to -4.38 . Since both values are biased, we roughly expect our algorithm to halve the logarithmic output size, that is, increase the size of the output interval of a neuron to the size of its square root.

Number of Distinct States The upper part of figure 5 shows the number of states that were evaluated to distinct values. The x-axis is aligned with the lower figure and we can see that in 41 of the cases with a higher resolution, also a higher number of states with distinct state value was identified. Since our algorithm is correct, we can conclude that

in these 41 games our evaluation can now distinguish states it could not distinguish before.

6 Summary

We have presented a generalization of $C - IL^2P$ algorithm that correctly represents a set of propositional rules as neural network. It structures the formerly monolithic transformation process of a propositional rule to a neuron into several smaller parts and allows for potentially lower weights. This proves advantageous when using the algorithm to create a neural network that must correctly represent a domain theory and at the same time be able to distinguish between several input vectors. As an example we showed that the evaluation function of a game-playing agent can correctly fuzzy-evaluate and distinguish states for even complex games with an increase in output resolution by several orders of magnitude. The main disadvantage of the algorithm is its higher worst-case run-time requirements. However, for GGP these aspects do not play a significant role due to the layered structure of propositionalized goal conditions. Moreover, the proposed heuristic reduces the probability of a neuron parameter recalculation.

6.1 Future Work

With the approach presented we can derive an evaluation function for General Game Playing based on neural networks. If further enhanced with features, the evaluation function should be able to compete with state-of-the-art systems, given that the typical problems of neural networks (no initialization, low resolution) are addressed. Specifically, the evaluation function should outperform in setups that allow for learning since, unlike other agents, it builds on logic and learning. Currently, however, such a setup is not part of the GGP championship.

Neural networks also implicitly solve the feature weighting problem: Equipped with a feature generation mechanism, features can be inserted in the network and after enough training episodes the connection weights indicate the utility of the feature. A feature with near-zero weights can consequently be removed. Such a feature addition and subtraction mechanism combined with a reasonable evaluation function would also allow for self-play and function evolution.

On the technical level, with parameter Δ fixed to 1, the chance to adapt the local Δ -values is not used for further maximization of the output resolution. An interesting possibility is also to use first-order logic as domain theory as considered in (Bader and Hitzler 2005).

References

- Bader, S., and Hitzler, P. 2005. P.: Dimensions of neural-symbolic integration - a structured survey. In *We Will Show Them: Essays in Honour of Dov Gabbay*, 167–194. College Publications.
- Clune, J. 2007. Heuristic evaluation functions for general game playing. In *AAAI*. Vancouver: AAAI Press.
- d’Avila Garcez, A. B. K. B., and Gabbay, D. 2002. *Neural-Symbolic Learning Systems*. Springer.

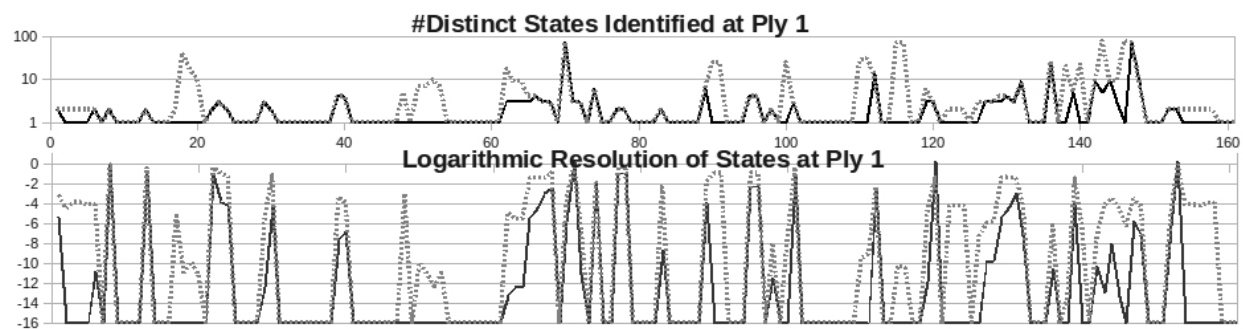


Figure 1: Above: Number of Distinct States Found; Below: Logarithmic Resolution of States

Finnsso, H., and Björnsson, Y. 2008. Simulation-based approach to general game playing. In *AAAI*. AAAI Press.

Love, N.; Hinrichs, T.; Haley, D.; Schkufza, E.; and Gensereth, M. 2008. General game playing: Game description language specification. Technical Report March 4. The most recent version should be available at <http://games.stanford.edu/>.

Michulke, D., and Thielscher, M. 2009. Neural networks for state evaluation in general game playing. In *ECML PKDD '09: Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases*, 95–110. Berlin, Heidelberg: Springer-Verlag.

Schiffel, S., and Thielscher, M. 2007. Fluxplayer: A successful general game player. In *Proceedings of the National Conference on Artificial Intelligence*, 1191–1196. Vancouver: AAAI Press.

Towell, G. G.; Shavlik, J. W.; and Noordenier, M. O. 1990. Refinement of approximate domain theories by knowledge based neural network. volume 2, 861–866.

Tree Parallelization of Ary on a Cluster

Jean Méhat

LIASD, Université Paris 8, Saint-Denis France,
jm@ai.univ-paris8.fr

Tristan Cazenave

LAMSADE, Université Paris-Dauphine, Paris France,
cazenave@lamsade.dauphine.fr

Abstract

We investigate the benefits of Tree Parallelization on a cluster for our General Game Playing program Ary. As the Tree parallelization of Monte-Carlo Tree Search works well when playouts are slow, it is of interest for General Game Playing programs, as the interpretation of game description takes a large proportion of the computing time, when compared with program designed to play specific games. We show that the tree parallelization does provide an advantage, but that it decreases for common games as the number of subplayers grows beyond 10.

Introduction

Monte-Carlo Tree Search is quite successful for General Game Playing (Finnsson and Björnsson 2008; Méhat and Cazenave 2010b) even if other approaches such as the knowledge-based approach also exist (Haufe et al. 2011). An important feature of Monte-Carlo Tree Search is that it improves with more CPU time. Therefore in the time allocated to make a move, it is desirable to develop as much as possible the Monte-Carlo tree in order to gain as much as possible information on the available moves. Parallelizing Monte-Carlo Tree Search is a promising way to make use of more CPU power.

In this paper we investigate the parallelization of our General Game Playing player Ary (Méhat and Cazenave 2010a) on a cluster of machines.

The next section details the parallelization of Monte-Carlo Tree Search. The third section shows how we have applied it to Ary. The fourth section gives experimental results for various games from previous General Game Playing competitions.

Parallelization of Monte-Carlo Tree Search

There are multiple ways to parallelize Monte-Carlo Tree Search (Cazenave and Jouandeau 2007). The most simple one is the *Root Parallelization*. It consists in running separately on different machines or cores the Monte-Carlo Tree Search algorithm developing independently its specific tree, and in collecting at the end of the allocated time the results of the separate searches. Each move at the top of the tree is qualified by combining the results of the independent searches. This way of parallelizing is extremely simple and

works well for some games such as Go (Chaslot, Winands, and van den Herik 2008) or some games from General Game Playing represented in the Game Description Language such as Checkers or Othello (Méhat and Cazenave 2010c).

Another way of parallelizing Monte-Carlo Tree Search is the *Tree parallelization* (Cazenave and Jouandeau 2008; Chaslot, Winands, and van den Herik 2008). It consists in sharing the tree among the various machines or cores. On a multi-core machine there is only one tree in memory and different threads descend the tree and perform playouts in parallel. On a cluster the main machine holds the tree and descends it. After each descent it selects another available machine of the cluster and sends the moves associated to the descent to this machine. The remote machine then plays the moves it has received, starting from the position at hand and continues with a random playout. It then sends back to the main machine the result of the playout and becomes available again.

Tree parallelization of the Fuego Go program using lock-free multi-threaded parallelization has been shown to improve significantly its level of play (Enzenberger and Müller 2009).

Centurio is an UCT based General Game Playing agent. It uses multi-core tree parallelization and cluster based root parallelization (Möller et al. 2011).

Gamer is also an UCT based General Game Playing agent. Experiments with the tree parallelization of Gamer on a multi-core machine brought speedups between 2.03 and 3.95 for four threads (Kissmann and Edelkamp 2011).

Tree parallelization of Ary

Current General Game players using Monte-Carlo Tree Search do not perform many simulations when compared with programs playing specific games. This is due to the way the game description is used for generating legal moves, applying joint moves, determining if a situation is terminal and getting the scores of the players.

In Ary, the game description received from the Game Master in the Game Description Language (GDL) is translated into Prolog and interpreted by a Prolog interpreter. When a node is created in the tree, its legal moves — or the scores of the players in terminal situations — are obtained from the interpreter and stored in the node; they are available for further descents without interaction with the interpreter.

On the other hand, when performing playouts, the interpreter is used at each step to analyze the current situation. The results of this analysis are discarded once they have been used to avoid saturating the memory.

Playouts are slow in General Game Playing, and tree parallelization of Monte-Carlo Tree Search on a cluster gives better speedups when playouts are slow (Cazenave and Jouandeau 2008). It is therefore natural to try Tree Parallelization of our General Game Playing agent Ary on a cluster.

In the cluster, one machine is distinguished as the Player: it interacts with the Game Master and maintains the UCT tree. We name the other the Subplayers; they only perform playouts at the request of the Player. All transmission between the Player and a subplayer are done via standard TCP streams.

At the beginning of a match, the Player transmits to all the subplayers the GDL description of the game received from the Game Master.

Result reception in the Player

Before requesting a playout and before each descent in the UCT tree, the Player scans with a *select* system call its connections with the Subplayers to detect which ones have data available. The available data are playout results: they are read and used to update the UCT tree, and the Subplayers are marked as available for another playout.

Playout request in the Player

Algorithm 1 Main algorithm in the Player.

```

while the available time is not elapsed do
  receive playout results if any
  process received playout results
  node ← root node
  while it is possible to descend the UCT tree do
    select child node
    node ← child
  end while
  expand node
  if node is terminal then
    update tree
  else
    receive playout results if any
    while not available Subplayer do
      process received playout results
      wait for data from any Subplayer
    end while
    send node description to the available Subplayer
    process received playout results
  end if
end while

```

Algorithm 1 presents the main algorithm in the Player, descending in the UCT tree, requesting playouts from the Subplayers and receiving their results.

The Player descends the UCT tree. When it arrives at a leaf of the built tree, it expands it into a new node and if

the node is not terminal, it selects a Subplayer, by scanning their states until finding one marked as available. This scan is done in a fixed order, permitting to establish a preference order between the Subplayers. When all the Subplayers are busy, the Player waits until one has finished the task at hand and reaches the available state.

Once a subplayer is found available, the Player sends to it the situation in the node in GDL. We opted to send the current situation instead of the sequence of moves used from the root node as done usually in Tree Parallelization. It avoids to have to interpret the application of this sequence of moves in the subplayer, which necessitates slow interactions between the Subplayer and its GDL interpreter.

Subplayer loop

Algorithm 2 Subplayer algorithm

```

receive game description
while true do
  get a state description
  play a playout
  send the playout result to the main machine
end while

```

The algorithm 2 resumes the work a Subplayer.

The Subplayers receive a game description in GDL, load it into their GDL interpreters and then enter a loop.

They wait for a description of a situation of the game, play a completely random playout until a terminal situation, and send the results to the Player. In the current setting, the result is only the score of each player in the final situation, but they might send back the sequence of moves played in the playout, at the cost of a slightly slower communication.

Algorithm 3 Send algorithm in the Player.

```

while not available Subplayer and not time elapsed do
  receive playout results if any
end while
if not time elapsed then
  send current node description
  receive playout results if any
  process received playout results
end if

```

Experimental results

We made a single process Ary, using a single thread to descend into the tree and run the playouts, play matches in a variety of games against a version of Ary running on a cluster using between 1 and 16 Subplayers.

The cluster is made a mixture of standard 2 GHz, 2.33 GHz and 3 GHz PC with two gigabytes of central memory running Linux connected via a switched 100 Mbits Ethernet network. Each machine hosted only a single Subplayer or Player to avoid race for memory between the players. For each match, the single Player, the parallel Player and the

game	Number of subplayers				
	1	2	4	8	16
Breakthrough	18	40	58	68	77
Connect 4	26	38	38	42	50
Othello	41	68	67	81	96
Pawn whopping	43	36	51	54	59
Pentago	38	40	55	73	87
Skirmish	73	76	76	78	79

Table 1: The results of the Tree Parallel Player running as second player against a single player, averaged over 100 matches.

game	Number of subplayers				
	1	2	4	8	16
Breakthrough	44	65	60	67	65
Connect 4	28	44	63	66	75
Othello	59	60	72	84	83
Pawn whopping	44	45	43	46	35
Pentago	35	54	68	64	68
Skirmish	71	71	74	76	71

Table 2: The results of the Root Parallel Player running as second player against a single player, from [Méhat and Cazenave, 2010c].

Subplayers were dispatched at (pseudo)-random between available machines.

The matches were run with 8 seconds of initial time and 8 seconds per move. The games tested were *Breakthrough*, *Connect 4*, *Othello*, *Pawn whopping*, *Pentago* and *Skirmish*. The rules used were the ones available on the Dresden Game Server.

For each game, we ran 100 matches with the Tree Parallel Player as second player, except for setting with 16 subplayers where the number of matches was limited to 70 because of time constraints on the use of the cluster.

Results of the matches

The results of each player are presented in table 1. There is only a slight improvement for the games *Skirmish* and *Pawn whopping*, while it is particularly notable for *Breakthrough*, *Pentago* and particularly at *Othello*. The game *Connect 4* is in-between, with results that get better as the number of subplayers augments, but not as much as in *Breakthrough*, *Pentago* and *Othello*.

Comparison with Root Parallelism

These results can be compared with those presented in (Méhat and Cazenave 2010c), where the same games were played using Root Parallelism in the same settings on the same machines, except that the matches were run with a 10 seconds playing time (figure 2). The results used are those

obtained by combining the accumulated values and number of experiments in the root nodes of the trees developed independently in the subplayers, as it is the one that gave the best results for multiplayer games.

Root Parallelism did work for *Breakthrough* and *Skirmish*, and Tree Parallelism also does bring an amelioration for these games. While *Pawn whopping* did not get better with Root Parallelism, it shows some amelioration until eight subplayers with Tree Parallelism.

Secondly, the overall results with Tree Parallelism with 16 subplayers are better than with Root Parallelism in all the games, except *Connect 4*.

The differences between Root Parallelism and Tree Parallelism reside in the sharing of nodes, the choice of branches to explore and the cost of communications. With Root Parallelism, the same node has to be expanded into every Subplayer where it is explored, while with Tree Parallelism, the node is expanded only once. In Root Parallelism, the choice of the branch in the UCT descent phase are only based on the node explored in the the Subplayer, while with Tree Parallelism the results of the playouts of all the Subplayers are taken into account. Finally, Root Parallelism incurs only one interaction per played move, when Tree Parallelism needs an interaction for every playout delegated to a Subplayer.

When there is only one Subplayer, there is only one tree, developed in the Subplayer for Root Parallelism or in the Player for Tree Parallelism. The only distinguishing factor between the two methods is then the communication cost, whose impact should be greater in games with short playouts. It comes as a surprise that Root Parallelism with one Subplayer exhibits significantly better results than Tree Parallelism with one Subplayer for *Breakthrough* and *Othello*, the two games where the playouts are slow. This point needs more investigations.

Use of the subplayer during the game

The benefits obtained from delegating playouts to subplayers vary between phases of the match. At the match goes on, the time of the descent of the tree tends to augment with the depth of the tree, while the time for a playout tends to diminish with the number of moves in the playout. Moreover, when the match is nearly finished, the descent in the UCT tree arrives with a growing frequency to terminal positions where there is no need to run playouts.

This variation has an influence on the benefit brought by using Subplayers. To measure it, we computed the average number of playouts computed by each subplayer at each move in the *one against 16* matches. The following figure shows these numbers for the first, the fourth, the eighth and the sixteenth Subplayer for some of the studied games. As the first available subplayers is solicited when one is needed, it allows to evaluate how useful is each subplayer.

For the game *Skirmish*, the evolution is presented in figure 1. The subplayers are able to compute about 120 playouts at the beginning of the match, and the last subplayer is only used at half of its capacity. As the match advances, the playouts get shorter and their number grow. After the tenth move, the 8th subplayer is less used, until move 27 where its

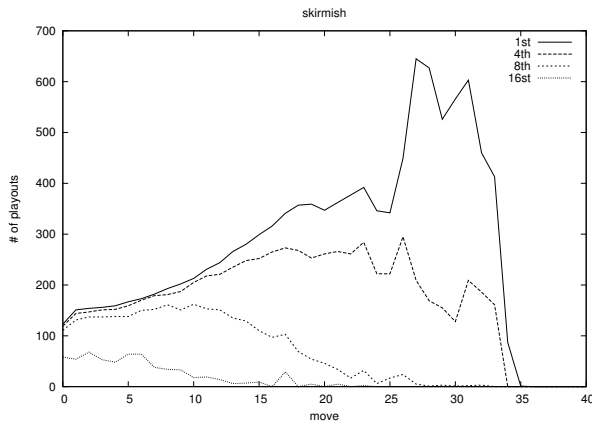


Figure 1: The evolution of the number of playouts for some subplayers in the game of *Skirmish* with 16 Subplayers.

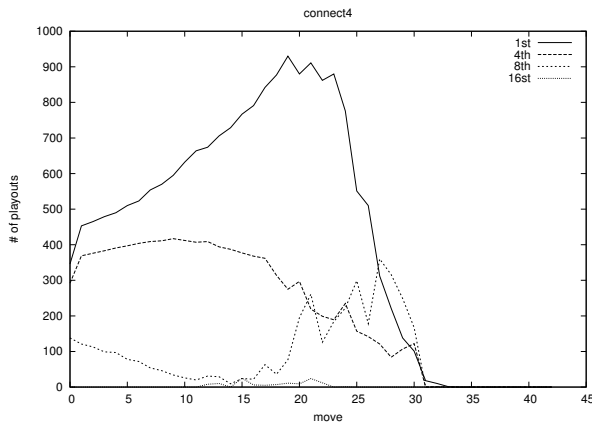


Figure 2: The evolution of the number of playouts for some subplayers in the game of *Connect 4* with 16 Subplayers.

use descends to 0. The curves for *Pawn whopping* are quite similar.

For the game *Connect 4*, presented in figure 2, the 16th subplayer is not solicited during the whole match, and the 8th subplayer is only half busy at the beginning. After move 17, it enters into action. The 4th and 8th subplayer are as busy between moves 20 and 25.

For the game *Pentago*, presented in figure 3, all the subplayers are used at full capacity until move 11 ; then the utility of the 16th subplayer diminishes until getting nearly not used at move 30. The 8th subplayer is used until move 20.

For the game *Breakthrough*, the evolution presented in figure 4 has the same structure, but here the 16th subplayer is kept busy nearly until the end of the game but presents a peak of activity near the end of the game.

The curve for *Othello* appears in figure 5. The interpretation of these rules are pretty slow and the number of playouts at the beginning is around 25 for all the subplayers. The 8th subplayer is kept busy until move 35 and the 4th subplayer nearly until move 50.

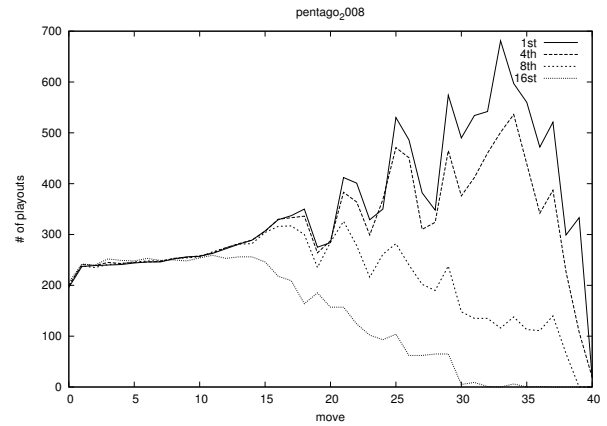


Figure 3: The evolution of the number of playouts for some subplayers in the game of *Pentago* with 16 Subplayers.

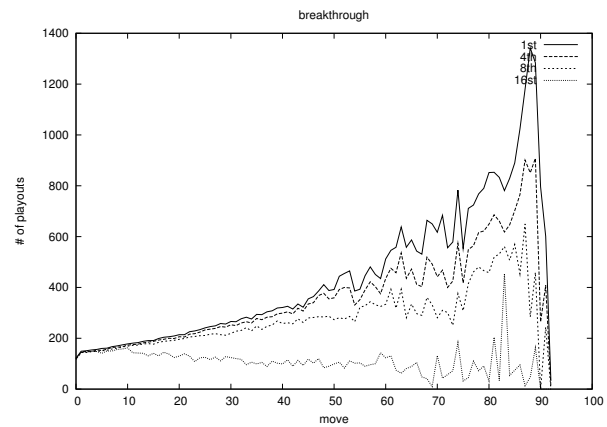


Figure 4: The evolution of the number of playouts for some subplayers in the game of *Breakthrough* with 16 Subplayers.

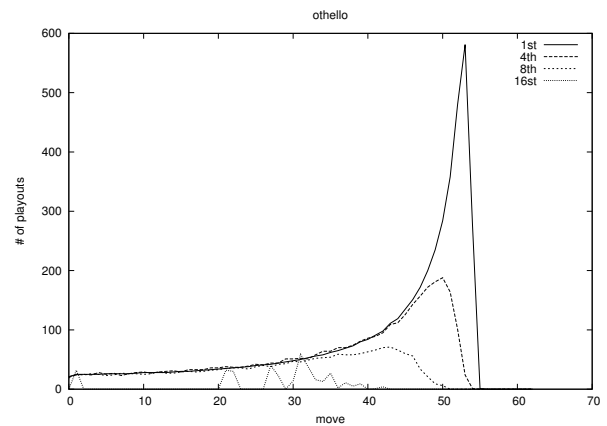


Figure 5: The evolution of the number of playouts for some subplayers in the game of *Othello* with 16 Subplayers.

Conclusion

We have implemented a Tree Parallel version of our General Game Playing agent Ary, and tested it on a variety of games.

We have shown that, in contrast with the Root Parallel version studied in (Méhat and Cazenave 2010c) that worked for some games but not for others, the Tree Parallel version improves the results against a serial player on all considered games, on some games more than others. This improvement is not directly related to the length of the playout, but to the ability of the Player to keep the Subplayers busy at the beginning of a match.

For ordinary games, there is no great benefit to be expected from a number of subplayers over 16.

Acknowledgement

We are grateful to David Elaissi, Nicolas Jouandeau and Stéphane Ténier who gave us access to the machines where the tests were run.

References

- Cazenave, T., and Jouandeau, N. 2007. On the parallelization of UCT. In *CGW*, 93–101.
- Cazenave, T., and Jouandeau, N. 2008. A parallel Monte-Carlo tree search algorithm. In *Computers and Games*, volume 5131 of *Lecture Notes in Computer Science*, 72–80. Springer.
- Chaslot, G.; Winands, M. H. M.; and van den Herik, H. J. 2008. Parallel monte-carlo tree search. In *Computers and Games*, volume 5131 of *Lecture Notes in Computer Science*, 60–71. Springer.
- Enzenberger, M., and Müller, M. 2009. A lock-free multi-threaded monte-carlo tree search algorithm. In *ACG*, volume 6048 of *Lecture Notes in Computer Science*, 14–20. Springer.
- Finnsson, H., and Björnsson, Y. 2008. Simulation-based approach to general game playing. In *AAAI*, 259–264.
- Haufe, S.; Michulke, D.; Schiffel, S.; and Thielscher, M. 2011. Knowledge-based general game playing. *KI* 25(1):25–33.
- Kissmann, P., and Edelkamp, S. 2011. Gamer, a general game playing agent. *KI* 25(1):49–52.
- Méhat, J., and Cazenave, T. 2010a. Ary, a general game playing program. In *Board Games Studies Colloquium*.
- Méhat, J., and Cazenave, T. 2010b. Combining UCT and nested monte-carlo search for single-player general game playing. *IEEE Transactions on Computational Intelligence and AI in Games* 2(4):271–277.
- Méhat, J., and Cazenave, T. 2010c. A parallel general game player. *KI* 25(1):43–47.
- Möller, M.; Schneider, M.; Wegner, M.; and Schaub, T. 2011. Centurio, a general game player: Parallel, java- and asp-based. *KI* 25(1):17–24.

MCTS: Improved Action Selection Techniques for Deterministic Games*

Stefan Freyr Gudmundsson and Yngvi Björnsson

School of Computer Science
Reykjavik University, Iceland
{stefang10,yngvi}@ru.is

Abstract

The simulation-based principles behind *Monte-Carlo Tree Search (MCTS)* have their roots in non-deterministic domains. In game domains, MCTS has nonetheless proved successful in both non-deterministic and deterministic games. This has been achieved by using more or less identical selection mechanisms for choosing actions, thus potentially not fully exploiting the inherent differences of deterministic and non-deterministic games. In this paper we take steps towards better understanding how determinism and discrete game outcomes may influence how action selection is best done in the selection step in MCTS. We use a simple n -arm-bandit test domain to show that action selection can be improved by taking into account whether a game is deterministic and has only few discrete game outcomes possible. We introduce two methods in this context to do so: *moving average* return function and *sufficiency threshold* and evaluate them empirically in the n -arm-bandit test domain, as well as providing preliminary results in a GGP inspired game. Both methods offer significant improvement over the standard UCT action-selection mechanism.

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 1997) 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 has 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. The reason for the success of MCTS in this domain is in part because of the difficulty in automatically generating effective state evaluation heuristics. 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 2011), and MALIGNE (Kirci *et al.* 2011), however, with the notable exception of FLUXPLAYER (Schiffel and Thielscher 2007; Haufe *et al.* 2011), which employs a traditional heuristic-based game-tree search. Currently only finite, deterministic, perfect-information games can be expressed in the GDL language, so the focus of GGP agents has so far been on such games. However, extensions to GDL were recently proposed for augmenting the language to also express non-deterministic, imperfect information games (Thielscher 2010). Subsequently, a relevant question to ask is whether the deterministic vs. non-deterministic nature of a game affects how MCTS is best employed (in GGP). The simulation-

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

based principles behind MCTS have their roots in non-deterministic domains, and although MCTS has been used successfully in both deterministic and non-deterministic game domains, the inherent difference of such types of games has potentially not been fully exploited. For example, actions selection (at non-chance nodes) is currently done the same way in both domains.

In this paper we take steps towards better understanding how determinism and discrete game outcomes can affect the action selection mechanism of MCTS. We show that by taking such properties into consideration action selection can be improved. We introduce two methods for doing so: *moving average* return function and *sufficiency threshold*. Both methods show significant improvement over the standard action-selection mechanism.

The paper is structured as follows. In the next section we give a brief overview of MCTS, followed by a discussion of how determinism and discrete game outcomes influence action selection in MCTS. Next we introduce the two techniques for exploiting the differences and evaluate them empirically. Finally, we conclude and discuss future work.

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 leaf node is reached, where the *expansion* step expands the leaf 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. The four steps are depicted in Figure 1. 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:

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

$N(s)$ 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.

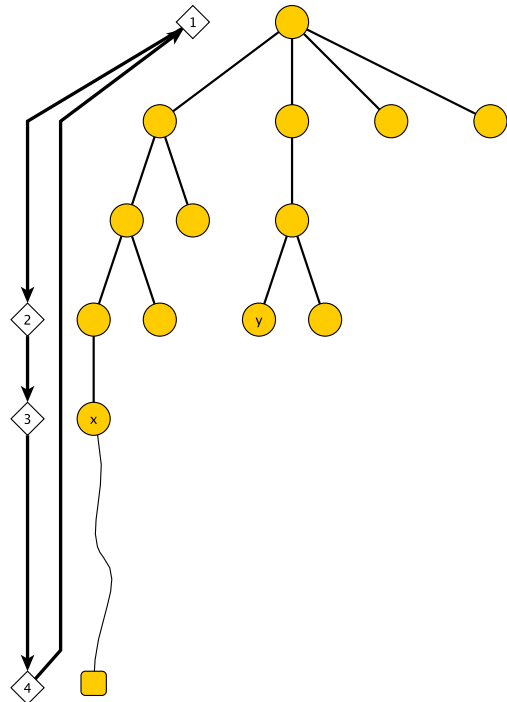


Figure 1: The four steps of MCTS: (1) *selection* step when traversing the tree from the root to a leaf; (2) *expansion* step, where node x is added; (3) *playout* from the expanded node until a terminal position is reached; (4) *back-propagation* step where the game outcome is backed up to the root, updating statistics as appropriate.

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.

Exploiting the Determinism

Assume that after running a fixed number of simulations in the tree in Figure 1 two of the available actions at the root have established themselves as substantially better than the others, say scoring 0.85 and 0.88 respectively. In a non-deterministic game with a substantial chance element, or in a game where the outcome is scored on a fine grained scale (e.g., $[0.00, 1.00]$), 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 only win ($=1$) and loss ($=0$) 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 in 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 legiti-

mate outcome based on enough simulations, spending additional simulations to distinguish between close values is not necessarily wise use of computing resources.

Furthermore, with only win and loss (and draw) outcomes in a deterministic game, once winning lines are found, it may take many simulations for that information to propagate up the tree. This is because the $Q(s, a)$ values are average based. For example, assume that node y in Figure 1 is found to be a win. This implies that the second action at the root is a win (because the second player has no alternative to switch to). Provided that there is already a large number of simulations behind the $Q(s, a)$ value of the second root move, it will take many more simulations to raise the action's value to a win. Although the same is true in games with many possible outcomes, then we are much less likely to see such drastic changes in an evaluation, and if a game is non-deterministic, then the effects of such changes will typically be dampened as they back up the tree because of chance nodes.

Generally speaking, in a non-deterministic game, or a game with a fine grained spread of outcomes, the $Q(s, a)$ values may be considered as reasonable estimators of the true game outcome, whereas in deterministic games with only a few discrete outcomes the value is not directly estimating the true outcome. For example, consider a game like Othello. How do we treat an estimate of, say 0.7, in a given position? We know that the true value of the position is either 0 or 1. Figure 2 shows the nature of the sampling averages when the samples we gather approach infinity. In the infinity we reach the true value (Kocsis and Szepesvári 2006). The figure shows how the estimates approach the true values as we gather more samples. For deterministic games with only two outcomes (DGs) the estimates approach either 0 or 1 but for non-deterministic games (or games with a "continuous" range of outcomes) (NDGs) the true values can take number of values between 0 and 1. If there is no additional information at hand we would expect the true value for an NDG in this situation to follow a β -distribution with mean 0.7. What we propose is that, for a DG, it should be possible to exploit the determinism. In a DG we know that an estimate approaches one of few known values. Our first attempt to exploit this are two methods we call *sufficiency threshold* (ST) and *moving average* return function (MA). We introduce them in the next two sections, respectively.

Sufficiency Threshold

In an NDG we may have two best moves with true values as 0.7 and 0.75 representing a 70% and 75% changes of winning, respectively. Early on in our search their estimates may not differ much and the MCTS agent spends most of its resources, i.e. samples, on deciding which is the better move - occasionally sampling from other moves. This is very important as finding the move with true value as 0.75 increases our winning changes by 5%. In DGs this is not as relevant. Let us take an example of a position in chess where white can capture a rook or a knight. After 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. We argue that at this point it is more important to get more

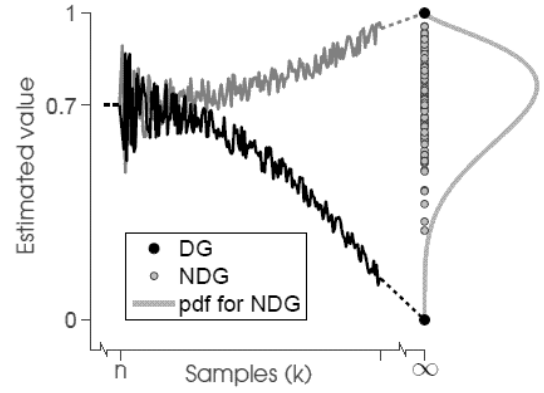


Figure 2: The estimated value is 0.7 after n samples. With infinite samples a DG can only take the values 0 or 1, whereas an NDG can take values spread over the interval $[0, 1]$ following a β -distribution with mean 0.7.

reliable information about one of the moves instead of trying to distinguish between, possibly, 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 $Q(s, a) > \alpha$ we unplug the exploration. To do so we replace C in Equation (1) by $\hat{C}$ as follows:

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

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, the ST agent differs from the UCT agent in temporarily rearranging the order of moves to sample. After such a rearrangement the agents more or less couple back to selecting the same moves to sample from.

Moving Average Return Function

Figure 2 shows how the reward function $Q(s, a)$ either approaches 0 or 1 in a DG with two possible outcomes. From our standpoint we sample more often from moves with high estimates and when we choose the opponents moves we sample more often moves which give us lower estimates. Thus we gradually build up lines of play similar to the minimax approach. Our first samples may give a misleading impression of a move but as we gather more samples the estimate $Q(s, a)$ gets more reliable and moves towards 0 or 1. This is a recipe for thinking about moving averages instead of the customary arithmetic mean used to evaluate $Q(s, a)$. The contribution of old samples in the arithmetic mean can overshadow the increase or decrease in $Q(s, a)$. We want to slowly forget older samples as we gather new ones. We do this using λ such that the update rule for $Q(s, a)$ is

$$Q(s, a) = Q_{old}(s, a) + \lambda(r - Q_{old}(s, a)) \quad (3)$$

where r is the result from the simulation. This method is sometimes called recency-weighted average and results in a weighted average of past rewards and the initial estimate of $Q(s, a)$ (Sutton and Barto 1998). Since the initial estimate plays a major role we do not start using the moving average until we have gathered some minimum amount of samples, M . With fewer samples we use the arithmetic mean and λ becomes

$$\lambda = \begin{cases} 1/N(s, a) & \text{when } N(s, a) \leq M, \\ \lambda_0 & \text{when } N(s, a) > M. \end{cases} \quad (4)$$

where λ_0 is a constant.

Experiments

In the following we first describe the experimental setup. In the experiments that follow we contrast moving average's (MA's) and sufficiency threshold's (ST's) performance to that of UCT. First we give an intuitional example using a single model and show the effect of adding noise to the model. This is followed by experiments representing average performance over a wide range of models. We also look at how increasing the number of actions affects performance. Finally, we provide preliminary results when using MA and ST in an actual game.

Setup

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). With each bandit having its own probability distribution the randomness is inherited. Therefore it is reasonable to simulate an NDG as n one-arm-bandits. Instead of n one-arm-bandits we can equally talk about one n -arm-bandit and we will use that terminology henceforth. To more accurately simulate a DG we alter the n -arm-bandit. We continue to give each bandit a probability distribution, but now we focus on a mean value for each bandit. Playing a bandit means that we choose a number uniformly at random from the interval $[0, 1]$. If the number is below the bandit's mean value the outcome of the play is a win, 1, otherwise it is a loss, 0. Furthermore we let the mean values move towards the possible true values for the DG we are simulating, following the path of a random walk¹. One can argue that the mean values follow the path of a correlated random walk (Goldstein 1951), since paths in the game-tree are often highly correlated. With use of the Central Limit

¹By the path of a random walk we mean a discretized path with constant step size with equal probabilities of taking a step in the positive or negative direction.

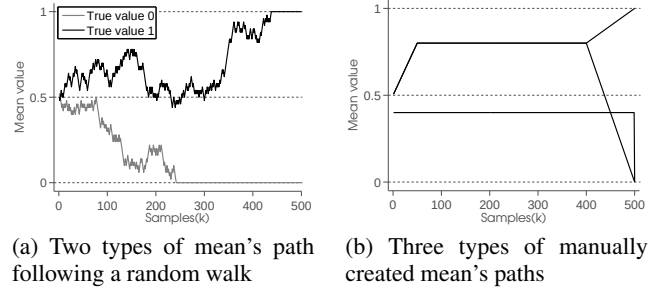


Figure 3: Types of mean's paths

Theorem we can approximate a correlated random walk with a random walk when the observation time, i.e. the number of samples, is long enough. It might be interesting to experiment with a correlated random walk coupled with a limited number of samples. This, however, adds complexity to the model which does not serve our purposes at this time. In our experiments 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 has reached 0 or 1 it has found its true value and it does not change after that. This way we get closer to the true value of a bandit the more samples we can gather from it. Here, we think of the simulation phase purely as information gathering. Instead of trying to maximize our profit from playing a fixed amount of coins we use a fixed amount of coins to gather information. With this information we choose one bandit to gamble all our money on in one bet and the outcome is dictated by the bandit's true value. We think this setup simulates well the process of using a Monte-Carlo sampling-based method in a DG. It is game-invariant and scales well with its parameters. Figure 3a shows possible paths hitting 0 or 1. We let $M_i(k_i)$ be the mean value for bandit i after k_i samples from it. We use the results from each sample to evaluate the expected reward of the bandits. Let $Q_i(k_i)$ be the expected reward for bandit i after k_i samples from it. The total number of samples is $k = \sum_i k_i$. The closer $Q_i(k_i)$ and $M_i(k_i)$ are to each other, the better is our estimate.

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 percentage for each agent of choosing the optimal action after k simulations. 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

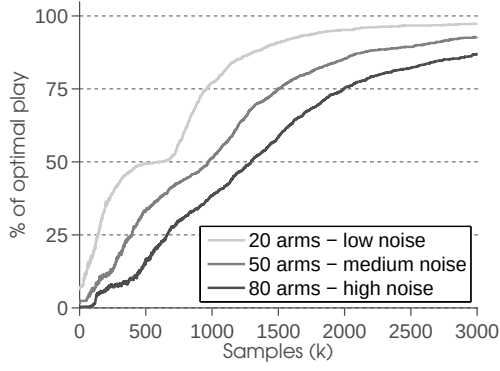


Figure 4: Manual setup using UCT with 1 winner

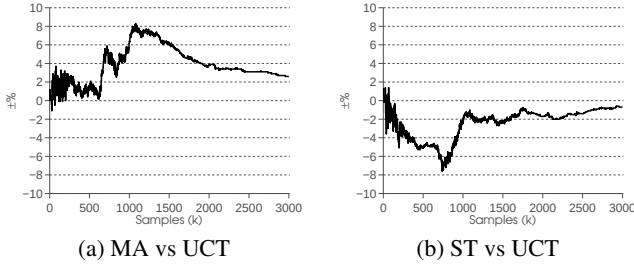


Figure 5: 20 arms - noise level low, 1 winner

game. In all our experiments we compare the performance of ST and MA to UCT.

Intuition Model and Noise Levels

To get an intuition for this setup we start by manually building three types of mean's paths, as seen in Figure 3b. The first and second path are the same up to $k_i = 400$. After that the first path moves towards 1 and the second to 0. The third type of path has a constant mean value of 0.4 up to $k_i = 499$ and is 0 when $k_i = 500$. We build an n -arm-bandit, referred to as a *model*, having one arm of the first type, one arm of the second type and the remaining (varying number of) arms of the third type, representing noise.

In the following experiments we run a single trial of 1000 tasks using the above bandit. We set $C = 0.4$ in UCT (the value used for CADIAPLAYER), $\alpha = 0.6$ in ST, and $M = 30$ and $\lambda = 0.05$ in MA. We did not put much effort into tuning the parameters of ST and MA, so these values do by no mean represent optimal settings. The graph in Figure 4 serves as a baseline and shows the performance of UCT for different number of arms where all but 2 arms are noisy (type 3). More arms thus represent more noise. We can see how the UCT agent distinguishes the two promising arms from the others in the first few steps. The step shape of the low-noise trajectory shows this best. For a while the agent can not distinguish between the two promising arms and thus selects the winners 50% of the time. Thereafter, the agent is able to choose the winner gradually more often as we gather more samples.

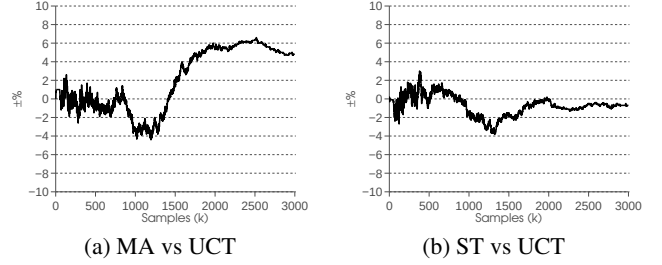


Figure 6: 50 arms - noise level medium, 1 winner

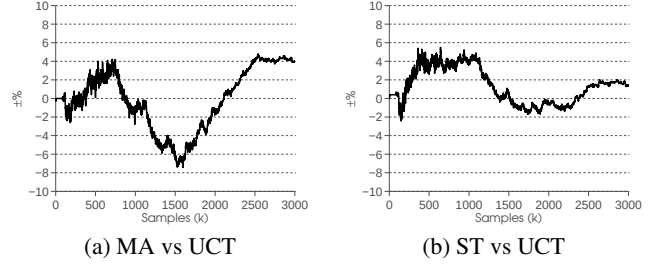


Figure 7: 80 arms - noise level high, 1 winner

The graphs in Figures 5-7 show MA's and ST's performance relative to the UCT baseline (Figure 4), with different levels of noise. The y -axis indicates the difference in how often the optimal action is chosen compared to UCT (e.g., if UCT chooses the correct action 50% of the time, a value of +4 means that MA (or ST) chooses the correct action 54% of the time). Two trends can be read from the graphs. With low-noise the ST agent is doing worse than the UCT agent to begin with as it spends more resources on noisy arms giving false positive results. With increasing noise the ST agent surpasses the UCT agent when the UCT agent spends more resources on exploring all arms instead of proving a promising arm right or wrong. As the noise increases the MA agent needs more samples to surpass the other agents but eventually does so convincingly. In Figures 6a and 7a we notice a zig-zag behavior of the MA agent, scoring better than UCT, then worse and again better. This is a known behavior of moving averages when the step-size, λ_0 , is too large (Sutton and Barto 1998).

Many Models and Actions

The aforementioned experiments provided intuition by using a single contrived model. In here 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. 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 calculate a 95% confidence interval over the models from $s \times t_{49}/\sqrt{50}$ with s as the sampled standard deviation and t_{49} from the Stu-

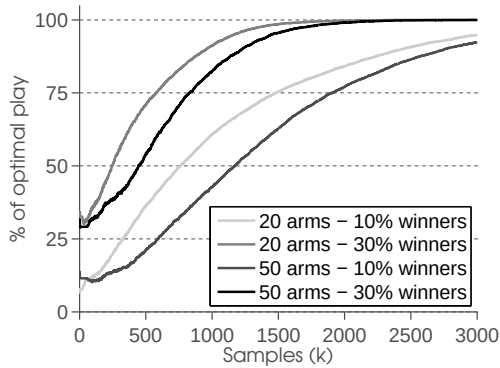


Figure 8: UCT with various number of arms and winners

dent's t -distribution with 49 degrees of freedom.

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 8 shows $\bar{V}(k)$ for UCT, which we use as a benchmark. Figures 9 and 10 show the performance of ST and MA relative to UCT when using 20 and 50 arms, respectively. In Figure 9, first we see how MA performs significantly better with 20 arms and 10% winners. Increasing the arms and winners seems to diminish MA's lead over the other agents. However, we should keep in mind that MA is sensitive to its parameters M and λ_0 . It looks like the MA's parameters are tuned too greedily for more arms and winners. ST does not perform much better than UCT with little noise but with higher percentage of winners it performs significantly better than UCT when few samples have been gathered. With more noise the ST agent is much better than UCT. We changed the parameters to a more passive approach for MA with 50 arms by setting $M = 50$ and $\lambda_0 = \frac{1}{50} = 0.02$. There is no longer a significant difference between MA and UCT with 50 arms.

Breakthrough Game

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. In here we provide preliminary experimental results in a variant of the game Breakthrough², frequently played in different variants in GGP competitions. More in-depth experiments remain as future work. The goal of Breakthrough is to advance a pawn to the end of the board. The pawns move forward, one square at a time, both straight and diagonally and can cap-

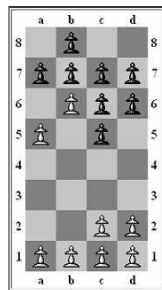


Figure 11: White wins with a5a6

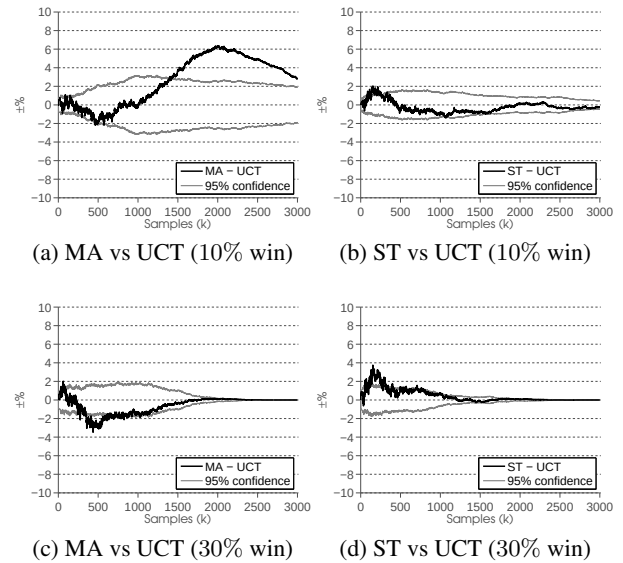


Figure 9: 20 arms

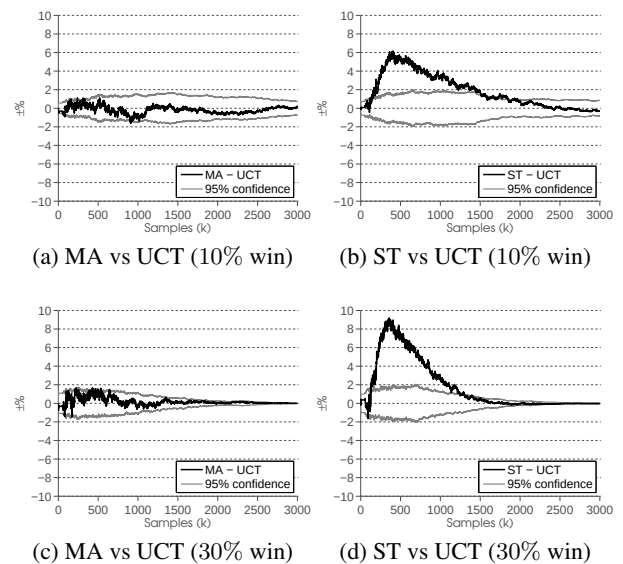


Figure 10: 50 arms

ture opponents pawns with the diagonal moves. The position in Figure 11 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 or b7. From b7 white can move to a8 and win.

²<http://boardgames.about.com/od/free8x8games/a/breakthrough.htm>

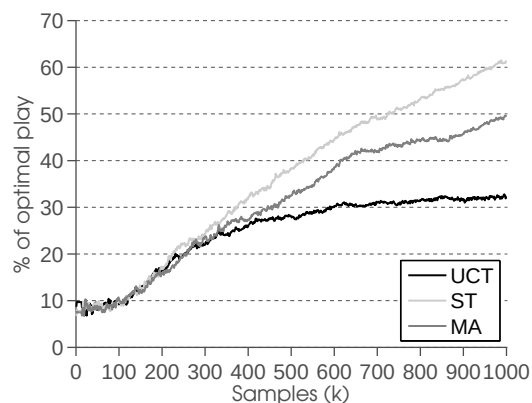


Figure 12: UCT, MA and ST in the position in Figure 11

Table 1: Frequency of a5a6 in 1000 trials

Samples	UCT	ST	MA
500	280	382 (< 1%)	326 (< 5%)
1000	323	611 (< 1%)	497 (< 1%)

Figure 12 shows how the three different methods, UCT, ST and MA, perform in the position in Figure 11. The threshold level for ST is 0.6 and for MA we had $M = 30$ and $\lambda = 0.05$. The exploration constant C was 0.4. Table 1 shows the frequency of the correct move, a5a6, and a measure of the probability that ST and MA are equal to UCT. Both ST and MA perform significantly better than UCT in this position. This position demonstrates clearly the drawbacks of UCT. To begin with, capturing on a7 and c7 are overestimated and playing a6 underestimated. By estimating the expected return with arithmetic mean these over- and underestimates get rectified rather slowly. Also, UCT wastes resources on exploring too much when the estimates are telling us we might have a forced win. We notice that in this real-game setup UCT behaves in a similar way as in the manual setup with low noise. In Figure 12 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.

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).

A solver based variant of MCTS (Winands *et al.* 2008; Cazenave and Saffidine 2010) allows proven values to be propagated correctly in the MCTS game-tree, thus expediting how fast such values back-propagate up the tree. This offers similar benefits as MA does, however, only in the extreme cases of proven values.

We are not aware of any previous work specifically looking into how the selection step in MCTS can take advantage

of the fact that it is playing a deterministic game with only discrete game outcomes.

Conclusions and Future Work

We have shown that the estimates of positions in DGs as a function of the number of samples follow a fundamentally different kind of path than in NDGs. Knowing that a function approaches either 0 or 1 in infinity can give us valuable information. Both our proposals to exploit this behavior, ST and MA, improve the traditional selection strategy in MCTS significantly. We should bear in mind that the MA agent is more sensitive to its parameters, M and λ_0 , than the other agents and we did not put much effort in fine tuning the parameters. We have fixed the exploration factor to $C = 0.4$ for all agents. Different values of C could change the outcome and that needs to be investigated. Apart from that the only parameters we can change are in the ST and MA agents. The results we have for ST and MA can therefore be thought of as lower limits.

We believe the n -arm-bandit setup represents the behavior of simulation-based agents for DGs well. Taking into account that we are not trying to maximize our profit as we gather samples, but using our samples to gather information to have the best probabilities of distinguishing a winning arm from the others. The results show significant improvements over UCT in all setups, for either or both of the agents ST and MA. The experiments are meant to mirror a wide range of possible positions in games with low and medium noise as well as few and many winners. Therefore we are confident that we have shown that our proposed methods, and more importantly our use of the determinism, improve the customary UCT selection method when used in DGs. Of special interest to GGP is the early lead of the ST agent over the others as GGP-players often have very limited time to play their moves.

We have pointed out two major parameters in describing a position of a game, i.e. how many moves are possible, the noise, and how many of them lead to a win. Increasing the noise seems to fit well for ST as well as increasing the number of winners. The MA does not perform as well with increased noise and winners but tuning the parameters in MA could improve it substantially. It is interesting to notice that the ratio of winners is not enough to explain the difference of the agents. It seems like the number of unsuccessful arms, or losers, does also play a part as they contribute to the number of false positive arms to begin with.

The Breakthrough experiment indicates that our methods fit well to GGP domains as ST and MA agents perform significantly better than UCT. Comparing the different methods in real games is the natural next step for future work. Furthermore, it would be interesting to try to characterize a real-game with parameters like noise, winners etc., e.g. the branching factor of the game-tree could be a representation of the noise. For each characterization we could then choose the correct method and parameters for the action selection phase. The methods we introduce here are intentionally simplistic, with the main purpose of demonstrating potentials. More sophisticated methods for handling moving averages and cutoff thresholds exist and will be investigated in future

work in this context. It is interesting to research in more detail the effect of noise, winners and losers on the agents and the parameters, C , α , M and λ_0 , need to be investigated.

References

- 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.
- Michael Buro. How machines have learned to play Othello. *IEEE Intelligent Systems*, 14(6):12–14, November/December 1999. Research Note.
- Murray Campbell, A. Joseph Hoane, Jr., and Feng-Hsiung Hsu. Deep blue. *Artificial Intelligence*, 134(1–2):57–83, 2002.
- Tristan Cazenave and Abdallah Saffidine. Score bounded monte-carlo tree search. In H. Jaap van den Herik, Hiroyuki Iida, and Aske Plaat, editors, *Computers and Games*, volume 6515 of *Lecture Notes in Computer Science*, pages 93–104. Springer, 2010.
- Guillaume Chaslot. *Monte-Carlo Tree Search*. PhD dissertation, Maastricht University, The Netherlands, Department of Knowledge Engineering, 2010.
- 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.
- 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.
- Hilmar Finnsson and Yngvi Björnsson. Cadiaplayer: Search-control techniques. *KI*, 25(1):9–16, 2011.
- 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.
- Michael R. Genesereth, Nathaniel Love, and Barney Pell. General Game Playing: Overview of the AAAI competition. *AI Magazine*, 26(2):62–72, 2005.
- Sidney Goldstein. On diffusion by discontinuous movements and on telegraph equation. *Q. J. Mech. Appl. Math.*, IV:129–156, 1951.
- Sebastian Haufe, Daniel Michulke, Stephan Schiffel, and Michael Thielscher. Knowledge-based general game playing. *KI*, 25(1):25–33, 2011.
- Mesut Kirci, Nathan R. Sturtevant, and Jonathan Schaeffer. A GGP feature learning algorithm. *KI*, 25(1):35–42, 2011.
- Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In *European Conference on Machine Learning (ECML)*, pages 282–293, Berlin / Heidelberg, 2006. Springer.
- 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/>.
- Jean Méhat and Tristan Cazenave. A parallel general game player. *KI*, 25(1):43–47, 2011.
- Jonathan Schaeffer. *One Jump Ahead: Challenging Human Supremacy in Checkers*. Springer-Verlag New York, Inc., 1997.
- Stephan Schiffel and Michael Thielscher. Fluxplayer: A successful general game player. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence (AAAI-07)*, pages 1191–1196. AAAI Press, 2007.
- Richard Sutton and A. Barto. *Reinforcement Learning: An Introduction*. MIT press, Cambridge, MA, USA, 1998.
- Gerald Tesauro. TD-Gammon, a self-teaching backgammon program, achieves master-level play. *Neural Comput.*, 6(2):215–219, 1994.
- Michael Thielscher. A general game description language for incomplete information games. In Maria Fox and David Poole, editors, *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010*. AAAI Press, 2010.
- Mark H. M. Winands, Yngvi Björnsson, and Jahn-Takeshi Saito. Monte-carlo tree search solver. In H. Jaap van den Herik, Xinhe Xu, Zongmin Ma, and Mark H. M. Winands, editors, *Computers and Games, 6th International Conference, CG 2008, Beijing, China, September 29 - October 1, 2008. Proceedings*, volume 5131 of *Lecture Notes in Computer Science*, pages 25–36. Springer, 2008.

On the Comparative Expressiveness of Epistemic Models and GDL-II*

Ji Ruan and Michael Thielscher

School of Computer Science and Engineering
The University of New South Wales, Australia
{jiruan,mit}@cse.unsw.edu.au

Abstract

A *general* game player automatically learns to play arbitrary new games solely by being told their rules. For this purpose games are specified in the game description language GDL, a variant of Datalog with function symbols that uses a few known keywords. A recent extension to GDL allows to describe nondeterministic games with any number of players who may have imperfect, asymmetric information. We analyze the epistemic structure and expressiveness of this language in terms of modal epistemic logic and present two main results: (1) The operational semantics of GDL entails that the situation at any stage of a game can be characterized by a multi-agent epistemic (i.e., S5-) model; (2) GDL is sufficiently expressive to model any situation that can be described by a (finite) multi-agent epistemic model.

Introduction

General game playing aims at building systems that automatically learn to play arbitrary new games solely by being told their rules (Pitrat 1971; Pell 1993). The Game Description Language (GDL) is a special purpose declarative language for defining games (Love *et al.* 2006). GDL is used in the AAAI General Game Playing Competition, where participants are provided with a previously unknown game specified in this language, and are required to dynamically and autonomously determine how best to play this game (Genesereth *et al.* 2005). A recent extension to the original language allows to describe games that include random elements and imperfect information (Thielscher 2010). This opens the door to nondeterministic games like Poker, in which players have imperfect, incomplete, and asymmetric information.

The game description language is a variant of Datalog with function symbols that uses a few known keywords. By applying a standard semantics for logic programs, a game description G can be interpreted by a state transition system. The execution model underlying GDL then determines a *game model* for G , which determines all possible ways in which the game may develop and what information the players acquire as the game proceeds (Love *et al.* 2006; Thielscher 2010). However, an open question has been to

what extent this game model, including its implicit epistemic structure due to imperfect and asymmetric information, satisfies standard properties of epistemic logic, and how expressive it is compared to this logic. The latter is particularly interesting because at first glance GDL seems to be constrained by the fact that all players have perfect knowledge of the game rules and in particular the initial position.

In this paper we analyze the epistemic structure and expressiveness of GDL in terms of epistemic logic. Seminal work in this area is (Hintikka 1962), and since then many philosophers have been interested in further developing the notions of knowledge and belief using a possible world semantics. In the late 1980s these approaches were picked up and further developed by computer scientists, cf. (Halpern and Vardi 1986; Fagin *et al.* 1995). This development 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 (an execution sequence) of the protocol. Apart from computer science, there is much interest in the temporal dynamics of knowledge and belief in areas as diverse as artificial intelligence (Moore 1990), multi-agent systems (Rao and Georgeff 1991), and game theory (Aumann and Brandenburger 1995).

We present, and formally prove, two main results:

1. The game model for any (syntactically valid) GDL game entails that at any round of the game the situation that arises can be characterised by a multi-agent S5-model.
2. Given an arbitrary (finite) epistemic model it is possible to construct a GDL game description which produces the situation described by this model.

This is complemented by an analysis of entailment of epistemic formulas in GDL and a discussion on possibilities to lift the finiteness restrictions in GDL in order to accommodate infinite epistemic models.

The remainder of the paper proceeds as follows. The next section recapitulates both game descriptions and epistemic logic. The third section analyzes the entailment of epistemic formulas in GDL and shows how the situations that arise during a game can always be characterised by a standard epistemic model that entails the exact same formulas. The fourth section discusses the case of infinite games. We conclude with a short discussion of related work.

*This is an extended version of our AAAI'11 paper entitled "The epistemic logic behind the game description language."

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 move ?m
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)	goal value for role ?r is ?v
sees(?r, ?p)	?r perceives ?p in the next position
random	the random player

Table 1: GDL-II keywords: the top eight comprise standard GDL while the last two have been added in GDL-II.²

Preliminaries

Describing Games with GDL

General Game Playing requires a formal language for describing the rules of arbitrary games. A complete game description consists of the names of the players, a specification of the initial position, the legal moves and how they affect the position, and the terminating and winning criteria. The emphasis of the game description language GDL is on high-level, declarative game rules that are easy to understand and maintain. At the same time, GDL has a precise semantics and is fully machine-processable. Moreover, background knowledge is not required—a set of rules is all a player needs to know in order to be able to play a hitherto unknown game.

A variant of Datalog with function symbols, the game description language uses a few known *keywords*; cf. Table 1. GDL is suitable for describing finite, synchronous, and deterministic n -player games with complete information about the game state (Love *et al.* 2006).³ The extended game description language GDL-II (for: *GDL with incomplete/imperfect information*) allows to specify games with randomness and imperfect/incomplete information (Thielscher 2010). Valid game descriptions must satisfy certain syntactic restrictions, which ensure that all deductions “ $\vdash$ ” used in the following definition are finite and decidable; for details we have to refer to (Love *et al.* 2006) for space reasons.

We need two abbreviations: Let $S = \{f_1, \dots, f_k\}$ be a *state*, that is, a finite set of terms (containing the position features that hold in S), then

$$S^{\text{true}} \stackrel{\text{def}}{=} \{\text{true}(f_1), \dots, \text{true}(f_k)\}$$

Furthermore, let $M = \langle m_1, \dots, m_n \rangle$ be a *joint move*, that is, a move (m_i) for each player (r_i) , then

$$M^{\text{does}} \stackrel{\text{def}}{=} \{\text{does}(r_1, m_1), \dots, \text{does}(r_n, m_n)\}$$

Definition 1. (Thielscher 2010) Let G be a valid GDL-II specification whose signature determines the set of ground

²The keywords are accompanied by the auxiliary, pre-defined predicate *distinct* (X, Y) , meaning the syntactic inequality of the two arguments (Love *et al.* 2006).

³Synchronous means that all players move simultaneously. In this setting, turn-taking games are modelled by allowing players only one legal move, without effect, if it is not their turn.

terms Σ . Let 2^Σ be the set of finite subsets of Σ . The semantics of G is the state transition system given by

- $R = \{r \in \Sigma : G \vdash \text{role}(r)\}$ (the roles);
- $s_0 = \{f \in \Sigma : G \vdash \text{init}(f)\}$ (the initial position);
- $t = \{S \in 2^\Sigma : G \cup S^{\text{true}} \vdash \text{terminal}\}$ (the terminal positions);
- $l = \{(r, m, S) : G \cup S^{\text{true}} \vdash \text{legal}(r, m)\}$, for all $r \in R$, $m \in \Sigma$, and $S \in 2^\Sigma$ (the legal moves);
- $u(M, S) = \{f \in \Sigma : G \cup M^{\text{does}} \cup S^{\text{true}} \vdash \text{next}(f)\}$, for all joint moves M and $S \in 2^\Sigma$ (the update function);
- $\mathcal{I} = \{(r, M, S, p) : G \cup M^{\text{does}} \cup S^{\text{true}} \vdash \text{sees}(r, p)\}$, for all $r \in R \setminus \{\text{random}\}$, $M : (R \mapsto \Sigma)$, $S \in 2^\Sigma$, and $p \in \Sigma$ (the information relation, determining the players’ percepts);
- $g = \{(r, v, S) : G \cup S^{\text{true}} \vdash \text{goal}(r, v)\}$, for all $r \in R \setminus \{\text{random}\}$ and $S \in 2^\Sigma$ (the goal relation).

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, and a player *cannot distinguish* two developments if he makes the same moves and perceptions in the two.

Definition 2. (Thielscher 2010) Let $\langle R, s_0, t, l, u, \mathcal{I}, g \rangle$ be the semantics of a GDL-II description G , then a development δ is a sequence

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

such that

- $d \geq 0$
- for all $i \in \{1, \dots, d\}$,
 - M_i is a joint move
 - $s_i = u(M_i, s_{i-1})$.

The length of a development δ , denoted as $\text{len}(\delta)$, is the number of states in δ , and $M(j)$ denotes agent j ’s move in the joint move M .

A player $j \in R \setminus \{\text{random}\}$ cannot distinguish two developments $\delta = \langle s_0, M_1, s_1, \dots \rangle$ and $\delta' = \langle s_0, M'_1, s'_1, \dots \rangle$ (written as $\delta \sim_j \delta'$) iff

- $\text{len}(\delta) = \text{len}(\delta')$
- for all $i \in \{1, \dots, \text{len}(\delta) - 1\}$:
 - $\{p : (j, M_i, s_{i-1}, p) \in \mathcal{I}\} = \{p : (j, M'_i, s'_{i-1}, p) \in \mathcal{I}\}$
 - $M_i(j) = M'_i(j)$.

Modal Epistemic Logic

In order to analyze the epistemic logic behind GDL-II and its semantics, we recapitulate basic notions from standard Modal Epistemic Logic (Fagin *et al.* 1995).

Definition 3. (Language) A basic Modal Epistemic Logic Language for epistemic formulas is given by the following Backus-Naur Form:

$$\phi := P \mid \neg\phi \mid \phi \wedge \psi \mid K_i\phi \mid C_B\phi$$

where P is an atomic proposition, i is an agent, and B is a non-empty set of agents. $\top, \perp, \vee, \rightarrow$ are defined as usual.

```

1 role(generalA). role(generalB). role(random).
2
3 succ(0,1). succ(1,2). ... succ(8,9).
4 time(3am). time(9pm).
5
6 init(round(0)).
7
8 gets_message(?g,?m) <=
9   role(?g), does(?g1,send(?m)),
10  does(random,pass), distinct(?g,?g1),
11  distinct(?g,random).
12
13 gets_new_message(?g) <= gets_message(?g,?m).
14 has_a_message(?g) <= true(message(?g,?m)).
15
16 legal(random,noop) <= true(round(0)).
17 legal(random,pass) <= not true(round(0)).
18 legal(random,stop) <= not true(round(0)).
19 legal(generalA,stime(?t)) <= true(round(0)),
20   time(?t).
21 legal(generalB,noop) <= true(round(0)).
22 legal(generalB,noop) <= true(control(generalB)),
23   not has_a_message(generalB).
24
25 legal(?g,send(ack(?m))) <= true(control(?g)),
26   true(message(?g,?m)).
27 legal(generalA,noop) <= true(control(generalB)).
28 legal(generalB,noop) <= true(control(generalA)).
29
30 sees(?g,?m) <= gets_message(?g,?m).
31
32 next(message(generalA,?t)) <= does(generalA,stime(?t)).
33 next(attack(?t)) <= does(generalA,stime(?t)).
34 next(attack(?t)) <= true(attack(?t)).
35
36 next(message(?g,?m)) <= gets_message(?g,?m).
37 next(message(?g,?m)) <= true(message(?g,?m)),
38   not gets_new_message(?g).
39
40 next(control(generalA)) <= true(round(0)).
41 next(control(generalA)) <= true(control(generalB)).
42 next(control(generalB)) <= true(control(generalA)).
43
44 next(round(?n)) <= true(round(?m)), succ(?m,?n).
45
46 terminal <= true(round(9)).

```

Figure 1: A GDL-II description of the Two Generals' Coordinated Attack Game: G_{ca} .

Intuitively, $K_i\phi$ means agent i knows ϕ , and $C_B\phi$ means that ϕ is common knowledge among the agents in B ; for example, “agent k knows that agent j knows P ” can be expressed as K_iK_jP . To give precise meanings to this language, we need multi-agent epistemic models.

Definition 4. A multi-agent epistemic model E is a structure $\langle W, \{\sim_i: i \in Ag\}, V \rangle$, where W is a set of possible worlds, Ag is a set of agents, each $\sim_i \subseteq W \times W$ is an equivalence relation (called the accessibility relation)⁴ for agent i , and $V: W \mapsto 2^{Atoms}$ is a valuation function that assigns each world a set of atomic propositions (said to be true in that world).

Then the entailment relation is defined as follows.

Definition 5. Given an epistemic model E and an epistemic formula ϕ , the entailment relation $\models$ is defined as follows:

- $E, w \models P$ iff $P \in V(w)$;
- $E, w \models \neg\phi$ iff $E, w \not\models \phi$;
- $E, w \models \phi \wedge \psi$ iff $E, w \models \phi$ and $E, w \models \psi$;
- $E, w \models K_i\phi$ iff for all w' , if $w \sim_i w'$ then $E, w' \models \phi$;
- $E, w \models C_B\phi$ iff for all w' , if $w \sim_B w'$ then $E, w' \models \phi$.

where $\sim_B$ is the transitive and reflexive closure of $\bigcup_{i \in B} \sim_i$.

From GDL-II to Epistemic Models

The section relates the game descriptions in GDL-II to epistemic models so that we can reason about these games using

⁴Note that in general the accessibility relation $\sim_i$ in E does not have to be an equivalence relation, but since epistemic models in this paper are meant to represent the knowledge of agents—rather than belief or other modalities—we restrict $\sim_i$ to be an equivalence relation; hence all epistemic models are S5-models (Fagin et al. 1995).

the modal epistemic logic presented in the previous section.

The choice of S5-models is based on our intention to model the knowledge of players, which is defined via the notion of *indistinguishable worlds*: agent i cannot distinguish two worlds if and only if i observes the same information in these two worlds; in other words, if agent i knows ϕ , then ϕ must be true in all worlds that agent i cannot distinguish from the current world. GDL-II itself only allows us to talk about factual knowledge of agents, e.g., “agent j sees P ” (which is equivalent to saying, “agent j knows P ”). It does not allow us to talk about the higher-order knowledge (the knowledge about the knowledge of agents), as in, “agent i knows that agent j knows P .” The epistemic language of modal logic S5 bridges this gap.

Meanwhile, there is certain information commonly known by all agents. Specifically, in general game playing the game description itself is such common knowledge for all agents. More precisely, not only do all agents *know* the game description they are going to play, but also they *know* that each other player *knows* this, and so on. This is implicit in the execution model for GDL, as the Game Master makes sure that every agent gets the same game description before starting the game. Accordingly, the initial state of a game is a common knowledge as well. This motivates our use of the $C_B\phi$ operator in the epistemic language (cf. Definition 3), which allows us to reason about such knowledge explicitly.

Before we present our results in all technical detail, we introduce a running example adopted from (Fagin et al. 1995).

Example 1. (Coordinated Attack Problem) A valley separates two hills. Two armies, each on its own hill and led by General A and B, respectively, are preparing to attack their common enemy in the valley. The two generals must have their armies attack the valley at the same time in order to

succeed. The only way for the two generals to communicate is by sending messengers through the valley. Unfortunately, there's a chance that any given messenger sent through the valley will be stopped by the enemy, in which case the message is lost but the content is not leaked. The problem is to come up with algorithms that the generals can use, including sending messages and processing received messages, that can allow them to correctly agree upon a time to attack.

It has been proved that such a coordinated attack is impossible (Fagin *et al.* 1995). We use this example to show that GDL-II is flexible enough to specify problems that involve complex epistemic situations and arguments. Specifically, we will use the game semantics to show why a coordinated attack is not possible.

Figure 1 describes a version of this problem in GDL-II. Generals A and B are modelled as two roles, and the enemy is modelled by the standard 'random' role. For the sake of simplicity, general A starts by selecting from just two possible attack times: '3am' or '9pm', and then sends his choice as a message to B. Subsequently, each general takes control in turn, and if he receives a message m then he sends an acknowledgement $ack(m)$ back to the other general, otherwise he does *noop*; simultaneously, the 'random' role always chooses randomly between either *pass*, which allows the message to go to the other general, or *stop*, which intercepts the message. For the sake of simplicity, we assume that the game terminates at round 9 and ignore the specification of goal values.

The semantics of a game description G according to Definition 1 derives a state transition system from a set of rules. We use the operational semantics implicit in Definition 2 to define the special concept of *epistemic game models* for GDL-II.

Definition 6 (GDL-II Epistemic Game Model). *Given an arbitrary GDL-II description G and its semantics $\langle R, s_0, t, l, u, \mathcal{I}, g \rangle$, an epistemic game model of G , denoted by $E(G)$, is a structure $\langle W, Ag, \{\sim_i : i \in Ag\}, V \rangle$ where*

- W is the set of developments of G ;
- Ag is the set of roles $R \setminus \{\text{random}\}$;
- $\sim_i \subseteq W \times W$ is the accessibility relation for agent $i \in Ag$ given by $(\delta, \delta') \in \sim_i$ (also written as $\delta \sim_i \delta'$) iff role i cannot distinguish developments δ and δ' ;
- $V : W \rightarrow 2^\Sigma$ is an interpretation function which associates with each development δ the set of ground terms in Σ that are true in the last state of δ .

In the following, we restrict our attention to finite epistemic models for the following reasons:⁵

- A game description in GDL-II is finite and so is the number of agents.
- At any game state, there are finitely many legal moves for each agent (including the random player).
- There is one initial state, and the outcome of a joint move is unique.

⁵We will discuss the possibility of lifting some of the restrictions to allow infinite models at the end of this paper.

- A game should always terminate.

As an example, from the game description G_{ca} for the Coordinated Attack Problem, we derive a game model $E(G_{ca})$ in two steps (see Figure 2). The first step is to use the game semantics for GDL-II to determine all states that are reachable from the initial state. A joint move is depicted as (a, b, c) , where a, b, c are the moves of, respectively, General A, General B, and 'random'. For instance, there are two possible joint moves at s_0 , $M_1 = (\text{settime}(3am), \text{noop}, \text{noop})$ and $M_2 = (\text{settime}(9pm), \text{noop}, \text{noop})$, which transit s_0 to s_1 and s_2 respectively. From s_1 there are again two possible joint moves which result in s_{11} where B receives A's message, and s_{12} where B receives nothing. Accordingly at state s_{11} , it is legal for B to send an acknowledgement, and s_{11} transits to two possible states s_{111} and s_{112} . This process goes on till a terminal state is reached.

The second step is to collect all the developments and then determine the individual accessibility relations. For example, consider the two developments $\delta_1 = \langle s_0, M_1, s_1 \rangle$ and $\delta_2 = \langle s_0, M_2, s_2 \rangle$. It is easy to check that $\delta_1 \not\sim_A \delta_2$ since General A moves differently in M_1 and M_2 . On the other hand, $\delta_1 \sim_B \delta_2$ since B make the same move in M_1 and M_2 and perceives nothing.

Based on our concept of an epistemic game model for GDL-II, we can define how to interpret formulas in the basic epistemic language over such models in a similar fashion as Definition 5.

Definition 7. *Given an epistemic game model $E(G)$, a development δ , and an epistemic formula ϕ , the entailment relation $\models$ is defined as follows:*

- $E(G), \delta \models P$ iff $P \in V(\text{last}(\delta))$;
- $E(G), \delta \models \neg\phi$ iff $E(G), \delta \not\models \phi$;
- $E(G), \delta \models \phi \wedge \psi$ iff $E(G), \delta \models \phi$ and $E(G), \delta \models \psi$;
- $E(G), \delta \models K_i\phi$ iff for all δ' , if $\delta \sim_i \delta'$ then $E(G), \delta' \models \phi$;
- $E(G), \delta \models C_B\phi$ iff for all δ' , if $\delta \sim_B \delta'$ then $E(G), \delta' \models \phi$.

where $\text{last}(\delta)$ is the last state of development δ , and $\sim_B$ is the transitive and reflexive closure of $\cup_{i \in B} \sim_i$.

Coming back to our running example, a simple and elegant argument can be given now on why a coordinated attack is never possible. First, using the epistemic language (Definition 3) we can express knowledge conditions such as:

- $K_A P$ for "general A knows that P", where P is an atomic expression, e.g., $\text{attack}(3am)$, which means "attack is set to be at 3am";
- $\neg K_B K_A P$ for "general B does not know whether general A knows P";
- $C_{\{A,B\}} P$ for " P is common knowledge for both".

Let $P = \text{attack}(3am)$ and $\delta_1, \delta_{11}, \delta_{111}$ be the left-most developments with length 1, 2, and 3 in Figure 2, then we can verify each of the following: $E(G_{ca}), \delta_1 \models K_A P \wedge \neg K_B K_A P$; $E(G_{ca}), \delta_{11} \models K_B K_A P \wedge \neg K_A K_B K_A P$; and $E(G_{ca}), \delta_{111} \models K_A K_B K_A P \wedge \neg K_B K_A K_B K_A P$. This implies that $E(G_{ca}), \delta \models \neg C_{\{A,B\}} P$ for $\delta = \delta_1, \delta_{11}, \delta_{111}$, that is, the attack time is not common knowledge among A and

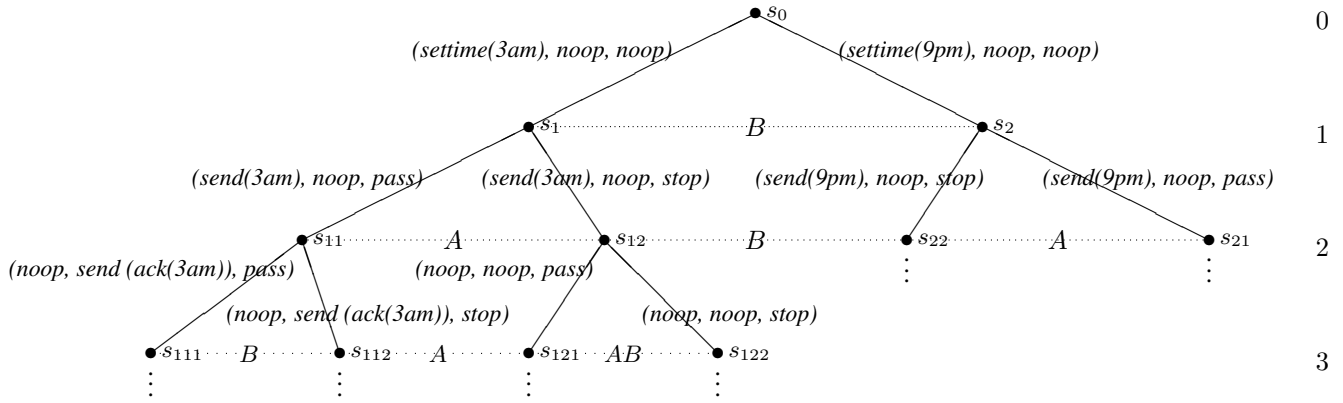


Figure 2: Epistemic game model $E(G_{ca})$ for the Two Generals' Coordinated Attack Game.

B even after the successful delivery of all messages during three rounds. We can generalize this to developments of arbitrary length. Such common knowledge is a precondition of coordinated attack, therefore it is not possible to coordinate such an attack.

In general, it is easy to show that the epistemic game model we constructed for GDL-II is equivalent to the standard concept of models and entailment in Modal Epistemic Logic: Specifically, we can pick up an arbitrary round and build a finite epistemic model for such round such that the truth of epistemic formulas is preserved.

Proposition 1. *Given an arbitrary GDL-II description G and any round of playing $k \geq 0$ (with round 0 corresponding to the initial state), we can derive a finite epistemic model E such that this round of the game is characterized by E .*

Proof. Let $E(G) = \langle W, Ag, \{\sim_i : i \in Ag\}, V \rangle$ be constructed from G according to Definition 6, and assume that the game playing is at round k .

Based on $E(G)$, we construct a finite epistemic model $E = \langle W', \{\sim_i : i \in Ag'\}, V' \rangle$ for round k as follows:

1. W' is the set of any game development $\delta \in W$ with $len(\delta) = k + 1$;
2. Ag' is the same set of agents as Ag ;
3. $\sim'_i$ is the equivalence relation $\sim_i$ restricted on the new domain W' , i.e., $\sim'_i = \sim_i \cap (W' \times W')$;
4. V is a valuation function such that $P \in V'(\delta)$ iff $P \in V(\delta)$ for any atomic proposition P and $\delta \in W'$.

We show by induction on the structure of formula ϕ that for all $\delta \in W'$: $E, \delta \models \phi$ iff $E(G), \delta \models \phi$. The propositional cases follow from the fact that the valuation does not change. For the case of $\phi := K_i \psi$, by definition, we have that $E, \delta \models K_i \psi$ iff for all δ' , if $\delta \sim'_i \delta'$ then $E, \delta' \models \psi$. If two developments δ, δ' have different lengths, then any agent can distinguish them, so if $\delta \sim'_i \delta'$, then $len(\delta) = len(\delta') = k + 1$, which means that $\delta' \in W'$ as well. So by induction, for all δ' , if $\delta \sim'_i \delta'$ then $E, \delta' \models \phi$ iff for all δ' , if $\delta \sim_i \delta'$ then $E(G), \delta' \models \phi$; therefore $E, \delta \models K_i \psi$ iff $E(G), \delta \models K_i \psi$. For the case of $\phi := C_B \psi$, the reasoning

is similar as the developments in the transitive and reflexive closure of $\cup_{i \in B} \sim_i$ are also of the same length $k + 1$. $\square$

As a corollary, we can show, for example, that the round number is a common knowledge for all the agents. For instance, $E(G_{ca}), \delta \models \bigwedge_k (round(k) \rightarrow C_{\{A,B\}} round(k))$.

From Epistemic Models to GDL-II

We now look at the other direction and show that for any given finite multi-agent epistemic model E we can construct a valid GDL-II game description such that E arises when playing the game. As a matter of fact, a (very abstract) game can always be constructed where a single move suffices to bring about an arbitrary given epistemic model.

Theorem 1. *For an arbitrary finite multi-agent epistemic model $E = \langle W, \{\sim_i : i \in Ag\}, V \rangle$ a GDL-II game description G can be constructed such that E arises from playing G , namely E is isomorphic to a sub-model of $E(G)$ that describes the situation after the first move.*

Proof. Let $W = \{w_1, \dots, w_k\}$ and $Ag = \{1, \dots, n\}$. G can be constructed as shown in Figure 3: The game has $n + 1$ roles, namely, the n agents plus the standard 'random' role (line 0–1). Initially, 'random' has a legal move $select(w)$ for any world $w \in W$ (lines 3–7) while all other players can only do $noop$ (line 8). The move $select(w)$ results in a state in which all atomic propositions hold that are true in world w (line 14). This rule uses an explicit enumeration of all pairs (w, P) such that $P \in V(w)$ (line 12). Furthermore, in order to arrive at the desired epistemic structure, the players get to see *all* worlds in their equivalence class $\{w' : (w, w') \in \sim_i\}$ (line 18–19). This rule uses an explicit enumeration of all triples (i, w_a, w_b) such that $w_a \sim_i w_b$ (line 16).⁶

We show that G indeed gives E according to the semantics in Definition 1. The initial state is $s_0 = \{step(0)\}$, and then $G \cup s_0^{\text{true}}$ entails $legal(random, select(w_j))$ for all $j \in [1..k]$, and $legal(1, noop), \dots, legal(n, noop)$.

⁶We omit definitions for terminal and goal since they are not relevant here.

```

0 role(1). ... role(n).
1 role(random).
2
3 world(w1). ... world(wk).
4
5 init(step(0)).
6
7 legal(random,select(?w)) <= true(step(0)), world(?w).
8 legal(?r,noop) <= true(step(0)),
9                 role(?r),
10                distinct(?r,random).
11
12 val(w1,p1). ... val(wk,pm).
13
14 next(?P) <= does(random,select(?w)), val(?w,?P).
15
16 equiv(1,wa,wa). equiv(1,wa,wb). ... equiv(n,wx,wy).
17
18 sees(?r,class(?w2)) <= does(random,select(?w1)),
19                        equiv(?r,w1,w2).

```

Figure 3: A GDL-II description for finite epistemic model E

Accordingly, each agent in Ag can only do *noop*, while ‘random’ may select an arbitrary world from E . Define joint move $M^j := (noop, \dots, noop, select(w_j))$ and consider new states $s_x = u(M^x, s_0)$, and $s_y = u(M^y, s_0)$, corresponding to the developments $\delta_x = \langle s_0, M^x, s_x \rangle$ and $\delta_y = \langle s_0, M^y, s_y \rangle$. If $w_x \sim_i w_y$, then agent i gets to see both $class(w_x)$ and $class(w_y)$ in both states s_x and s_y , in which case the agent cannot distinguish δ_x from δ_y because also his action is the same in both M_1 and M_2 . On the other hand, if $w_x \not\sim_i w_y$ then agent i can distinguish the two developments based on his percepts. Altogether this process gives us an epistemic game model $E(G)$. Then we define a standard epistemic model $E' = \langle W', \{\sim'_i : i \in Ag\}, V' \rangle$ from $E(G)$ as follows: W' is the set of all developments of length 2 from $E(G)$; $\sim'_i$ and V' are restrictions of $\sim_i$ and V on W' respectively.

Now E and E' are isomorphic: Each world $w_j \in W$ corresponds to the state s_j and hence to the development $\delta_j \in W'$. $(w_x, w_y) \in \sim_i$ iff $(\delta_x, \delta_y) \in \sim'_i$ for agent i . And for all atomic proposition P , $P \in V(w_j)$ iff $P \in V'(\delta_j)$. $\square$

We remark that Theorem 1 should not be interpreted as “GDL-II theories and epistemic logic theories are equally expressive.” To make it more precise, by a GDL-II theory we mean a set of GDL-II rules, and by an epistemic logic theory we mean a set of epistemic formulas. From a well-defined GDL-II description (namely a set of GDL-II rules), one can derive a game model according to its semantics. But given an epistemic formula ϕ , suppose it is satisfiable, i.e., there is an epistemic model E and a world w such that $(E, w) \models \phi$, then there are infinitely many models that satisfy this formula (as you can always create a new model by adding a single world without connecting to other worlds to model E). So the main difference is that a GDL-II description determines a unique, dynamic game model, whereas an epistemic formula determines a class of static epistemic mod-

els. What Theorem 1 does entail is this: Given an arbitrary epistemic formula, for any finite epistemic model for this formula, we can construct a GDL-II description G such that this epistemic model arises at some point from playing G .

Discussion

We come back to the restriction of finiteness on epistemic models. Recall that in general game playing, the finiteness of a game model comes from various restrictions described earlier in this paper. If we lift any of them, like for example the finiteness of legal moves, then we can end up with infinite models.

As an illustrative example, consider the following situation: there are two agents, who are each told a natural number such that two numbers differ by 1. Between them the agents know the difference but the numbers themselves are private information. An infinite game model arises from this situation since the set of possible pairs of natural numbers, $\{(x, y) : x, y \in \mathbb{N}, |x - y| = 1\}$, is infinite.

Figure 4 gives a possible description (call it G_{inf}) of this situation as a game in GDL-II. The game has 3 roles: agent A, agent B, and “Nature” (line 0). Natural numbers are defined (line 1–2) by the predicate *num()*, using the function *succ*(x) to encode the successor of a number.⁷ Initially, random chooses among the (infinitely many) moves *select*($x, succ(x)$) and *select*(*succ*(x), x), for each natural number x (line 7–10), while agents A and B can only do *noop* (line 11–13). The move *select*(x, y) results in a state in which propositions $pA(x)$ and $pB(y)$ are true (line 15–16). Furthermore, the agents get to see their own numbers (line 18–19).⁸

We can derive an infinite epistemic model $E(G_{inf}) = \langle W, \{\sim_A, \sim_B\}, V \rangle$ for step 1 of game G_{inf} . Let $\delta_{(x,y)} = \langle s_0, (noop, noop, select(x, y)), \{A_x, B_y\} \rangle$ (where A_x and B_y respectively abbreviate *has*(A, x) and *has*(B, y)). It is easy to see that

- $W = \{s_0\} \cup \{\delta_{(x,y)} : x, y \in \mathbb{N}, |x - y| = 1\}$
- $\delta_{(x,y)} \sim_A \delta_{(x',y')}$ iff $x = x'$
- $\delta_{(x,y)} \sim_B \delta_{(x',y')}$ iff $y = y'$
- $V(s_0) = \{step(0)\}$
- $V(\delta_{(x,y)}) = \{A_x, B_y\}$.

Now we can use epistemic logic to check some interesting properties. For example, suppose agent A and B hold 3 and 4, respectively. Both agents know that “A’s number is less than 6” (represented as $\bigvee_{x < 6} A_x$) and “B’s number is less than 6” (represented as $\bigvee_{x < 6} B_x$), but this is not a common knowledge. Formally, we can show that

$$E(G_{inf}), \delta_{(3,4)} \models K_A(\bigvee_{x < 6} A_x \wedge \bigvee_{x < 6} B_x)$$

⁷Note that *num*(*succ*(x)) <= *num*(x) is non-standard GDL because the clause violates the *recursion restriction* (Love *et al.* 2006), which has been introduced to avoid infinite term growth through recursion.

⁸Again we omit definitions for *terminal* and *goal* since they are not relevant here.

```

0 role(A). role(B). role(random).
1
2 num(0).
3 num(succ(x)) <= num(x).
4
5 init(step(0)).
6
7 legal(random, select(?x, succ(?x)) <= true(step(0)),
8                               num(?x)).
9 legal(random, select(succ(?x), ?x) <= true(step(0)),
10                              num(?x)).
11 legal(?r, noop) <= true(step(0)),
12                               role(?r),
13                               distinct(?r, random).
14
15 next(has(A, ?x)) <= does(random, select(?x, ?y)).
16 next(has(B, ?y)) <= does(random, select(?x, ?y)).
17
18 sees(A, ?x) <= does(random, select(?x, ?y)).
19 sees(B, ?y) <= does(random, select(?x, ?y)).

```

Figure 4: A GDL-II description of infinite game G_{inf} .

and

$$E(G_{inf}), \delta_{(3,4)} \not\models C_{\{A,B\}}(\bigvee_{x < 6} A_x \wedge \bigvee_{x < 6} B_x).$$

As a matter of fact, for any number $y \geq 6$ we can show that

$$E(G_{inf}), \delta_{(3,4)} \models K_A(\bigvee_{x < y} A_x \wedge \bigvee_{x < y} B_x)$$

but

$$E(G_{inf}), \delta_{(3,4)} \not\models C_{\{A,B\}}(\bigvee_{x < y} A_x \wedge \bigvee_{x < y} B_x).$$

Our results in this paper could, in principle, be extended to infinite games provided a non-standard extension of GDL-II can be used to describe such games, since our proofs do not depend on the finiteness of game models.

Apart from theoretical results, we will also be interested in investigating a more practical side of the problem. With an epistemic framework for GDL-II, we are now able to reason about epistemic properties of games. For example, given that agents may have only partial observation ability, it is easy to construct games in which agents do not have sufficient information to derive their legal moves; this may render a game unfair or even not playable. Therefore, a desirable property of a game may be to avoid such a situation, and we can use the epistemic structure to verify that a GDL-II description obeys this property. We express this property in the basic epistemic language by the formula for agent i ,

$$\phi_i = \bigwedge_m (legal(i, m) \rightarrow K_i legal(i, m))$$

To check such properties systematically amounts to the following model checking problem: given a GDL-II description G , a round number k , and an epistemic formula ϕ , verify that $E(G), \delta_{\leq k} \models \phi$ for all δ of length $\leq k$. In this case, we

certainly have to restrict ourselves to finite games. Further investigations will include the computational complexity of this model checking problem, and an extension of the basic epistemic language with temporal or dynamic operators. We will leave these for further research.

Conclusion

In this paper, we analyzed the epistemic structure and expressiveness of GDL-II in terms of modal epistemic logic and presented two results: (1) The operational semantics of GDL entails that the situation in any round of a game can be characterized by a multi-agent epistemic model, (2) GDL is sufficiently expressive to model any situation that can be described by a finite multi-agent epistemic model. We also discussed that it is also possible to extend these results into infinite models if GDL allows to describe infinite games.

In an accompanying paper we show how GDL-II can be formally translated into the Situation Calculus as a first-order axiomatisation that allows players to reason about their percepts and what they know about the legality and effects of moves based on the game description (Schiffel and Thielscher 2011). Also we have showed that, by relating the extended Game Description Language to the universal, mathematical concept of extensive-form games, any such game can be described faithfully (Thielscher 2011). Other related work describes the use of Alternating-time Temporal Logic to represent and verify properties of general games (Ruan *et al.* 2009), but this is restricted to original GDL and hence to games with perfect information. There is of course a large body of work on the epistemic structure of imperfect-information games, but ours is the first application of this line of research to formally analyze the epistemic structure behind the general Game Description Language GDL-II.

Acknowledgement

Michael Thielscher is the recipient of an Australian Research Council Future Fellowship (project number FT 0991348). He is also affiliated with the University of Western Sydney.

References

- R.J. Aumann and A. Brandenburger. Epistemic conditions for Nash equilibrium. *Econometrica*, 63:1161–1180, 1995.
- R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi. *Reasoning About Knowledge*. The MIT Press: Cambridge, MA, 1995.
- M. Genesereth, N. Love, and B. Pell. General game playing: Overview of the AAI competition. *AI Magazine*, 26(2):62–72, 2005.
- J.Y. Halpern and M.Y. Vardi. The complexity of reasoning about knowledge and time. In *Proceedings 18th ACM Symposium on Theory of Computing*, pages 304–315, 1986.
- J. Hintikka. *Knowledge and Belief*. Cornell University Press, Ithaca, NY, 1962.
- N. Love, T. Hinrichs, D. Haley, E. Schkufza, and M. Genesereth. General Game Playing: Game Description Language Specification. Technical Report LG–2006–01, Stanford University, 2006.

ford Logic Group, Computer Science Department, Stanford University, 353 Serra Mall, Stanford, CA 94305, 2006. Available at: games.stanford.edu.

R.C. Moore. A formal theory of knowledge and action. In J.F. Allen J., Hendler, and A. Tate, editors, *Readings in Planning*, pages 480–519. Morgan Kaufmann Publishers, San Mateo, CA, 1990.

B. Pell. *Strategy Generation and Evaluation for Meta-Game Playing*. PhD thesis, Computer Laboratory, University of Cambridge, 1993.

J. Pitrat. A general game playing program. In N. Findler and B. Meltzer, editors, *Artificial Intelligence and Heuristic Programming*, pages 125–155, Edinburgh University Press, 1971.

A.S. Rao and M.P. Georgeff. Modeling rational agents within a BDI-architecture. In James Allen, Richard Fikes, and Erik Sandewall, editors, *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning (KR'91)*, pages 473–484, San Mateo (CA), 1991. Morgan Kaufmann.

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.

S. Schiffel and M. Thielscher. Reasoning about general games described in gdl-ii. In *Proceedings of the 25th AAAI Conference on Artificial Intelligence (AAAI-11)*, San Francisco, CA, USA, August 2011.

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

M. Thielscher. The general game playing description language is universal. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, Barcelona, Spain, July 2011.

Heuristic Interpretation of Predicate Logic Expressions in General Game Playing

Daniel Michulke

Department of Computer Science
Dresden University of Technology
daniel.michulke@mailbox.tu-dresden.de

Abstract

Unlike traditional game playing, General Game Playing (GGP) is concerned with agents capable of playing classes of games. Given the rules of an unknown game, the agent is supposed to play well without human intervention. For this purpose, agent systems using game tree search need to automatically construct a state value function to guide search.

This state value function is often either probabilistic as in Monte-Carlo Systems and thus most likely unable to compete with deterministic functions in games like chess; or it is expensive in construction due to feature generation and learning-based selection and weighting mechanisms.

In this work we present an alternative method that derives features from the goal conditions stated in the game rules, avoiding thereby the disadvantages of other approaches. The paper structures and generalizes some known approaches, shows new ways of deriving features and demonstrates how to use these features as part of an evaluation function. Experiments demonstrate both a high effectiveness and generality.

1 Introduction

In General Game Playing, agents are supposed to play games they have never seen before. In a typical competition setup the agent receives the rules of a game and has a few minutes time until the corresponding match starts. Since the games are arbitrary deterministic games with complete information (such as Chess, Go, Tic-Tac-Toe, Rock-Paper-Scissors), the agent cannot rely on any preprogrammed behavior. Instead, it has to come up with a strategy that fits the game.

An important aspect of this strategy is the state evaluation function that evaluates states and is used to guide the agent to states with a high value. There are two basic types of evaluation functions:

Monte-Carlo based functions as used in (Finnsson and Björnsson 2008) evaluate a state by performing playouts, that is, playing random moves until a terminal state is reached. After several of such playouts, the average of the different match outcomes is considered the state value. The problem is that this evaluation assumes a random opponent behavior and stands no chance against informed and therefore non-random opponents in games with well-studied evaluation functions such as Chess. Extensions to the approach

such as (Finnsson and Björnsson 2010) moderate this effect, but do not solve the problem.

The alternative is a deterministic evaluation based on the state itself. To obtain such an evaluation function, agents such as (Kuhlmann, Dresner, and Stone 2006; Clune 2007) derive candidate features from expressions in the game description, evaluate them to obtain a measure of usefulness for the game and finally put the features together, possibly weighted, to form an evaluation function. While the approach is perfectly reasonable, the evaluation of candidate features is usually time-consuming which is especially critical in a competition setup.

Another type of deterministic evaluation avoids this problem by using a function that is essentially a fuzzified version of the goal condition as stated in the game rules (Schiffel and Thielscher 2007; Michulke and Thielscher 2009). After transforming the goal condition to a propositional fuzzy evaluation function, it is improved by substituting specific expressions by more expressive features. In contrast to the feature detection and weighting approach, the features applied have a direct justification since they occur in the goal condition. Also, they need no weights assigned as the fuzzy goal condition provides the context for dealing with the feature output, i.e. weights them implicitly.

In this work we will present a generalized feature detection mechanism that builds on the latter two approaches. We show how to detect and apply features for the use in such an evaluation function. In contrast to other works published until now, we focus on a clear algorithmic description of 1. the construction process of the evaluation function and 2. the use of the evaluation function. Note that in this paper we do not intend to present a competitive GGP agent since the topic of straightforward feature detection and weighting is complex enough. Instead, we aim to present a general technique of deriving an evaluation function from a given domain theory. We believe the techniques presented should be part of the repertoire of any non-probabilistic agent since the resulting evaluation function represents a sound starting point for state evaluation without the disadvantages related to standard approaches for detecting, weighting and combining features.

The features discussed will cover the most important features used in the cited sources, introduce some features not yet used and, most importantly, show how to detect, confirm

and apply these features without a costly feature evaluation and weighting phase. An evaluation will prove both generality and utility of the features.

The rest of this work is structured as follows: In the remainder of this section we will briefly introduce GDL and show in the next section how the goal condition of the game can be used to construct a simple evaluation function using (propositional) t-norm fuzzy logics and ground-instantiation. We proceed by introducing features in section 3 that we distinguish as expression features and fluent features. We evaluate these features in section 4 and summarize our results in section 5.

Game Description Language The language used for describing the rules of games in the area of General Game Playing is the Game Description Language (Love et al. 2008) (GDL). The GDL is an extension of Datalog with functions, equality, some syntactical restrictions to preserve finiteness, and some predefined keywords.

The following is a partial encoding of a Tic-Tac-Toe game in GDL. In this paper we use Prolog syntax where words starting with uppercase letters stand for variables and the remaining words are constants.

```

1 role(xplayer). role(oplayer).
2 init(cell(1,1,b)). init(cell(1,2,b)).
3 init(cell(2,3,b)). ... init(cell(3,3,b)).
4 init(control(xplayer)).
5 legal(P, mark(X, Y)) :-
6   true(control(P)), true(cell(X, Y, b)).
7 legal(P, noop) :- role(P), not true(control(P)).
8 next(cell(X, Y, x)) :- does(xplayer, mark(X, Y)).
9 next(cell(X, Y, o)) :- does(oplayer, mark(X, Y)).
10 next(cell(X, Y, C)) :-
11   true(cell(X, Y, C)), C \= b.
12 next(cell(X, Y, b)) :- true(cell(X, Y, b)),
13   does(P, mark(M, N)), (X \= M ; Y \= N).
14 goal(xplayer, 100) :- line(x).
15 ...
16 terminal :- line(x) ; line(o) ; not open.
17 line(P) :- true(cell(X, 1, P)),
18   true(cell(X, 2, P)), true(cell(X, 3, P)).
19 ...
20 open :- true(cell(X, Y, b)).

```

The first line declares the roles of the game. The unary predicate `init` defines the properties that are true in the initial state. Lines 5-7 define the legal moves of the game, e.g., `mark(X, Y)` is a legal move for role `P` if `control(P)` is true in the current state (i.e., it's `P`'s turn) and the cell `X, Y` is blank (`cell(X, Y, b)`). The rules for predicate `next` define the properties that hold in the successor state, e.g., `cell(M, N, x)` holds if `xplayer` marked the cell `M, N` and `cell(M, N, b)` does not change if some cell different from `M, N` was marked. Lines 14 to 16 define the rewards of the players and the condition for terminal states. The rules for both contain auxiliary predicates `line(P)` and `open` which encode the concept of a line-of-three and the existence of a blank cell, respectively.

We will refer to the arguments of the GDL keywords `init`, `true` and `next` as fluents. In the above example, there are two different types of fluents, `control(X)` with `X`

$\in \{\text{xplayer}, \text{oplayer}\}$ and `cell(X, Y, C)` with `X, Y` $\in \{1, 2, 3\}$ and `C` $\in \{b, x, o\}$. Any subset of the set of these fluents constitutes a state s . By applying the **next** rules on the current state s and the set of terms **does**(P, A) for each role P and its selected action A , one can derive the successor state s' of s .

2 The Evaluation Function

The evaluation function takes as argument a state and returns a value that indicates the degree of preference of the state. The function is used to find the best among different actions by deriving the successor state(s) induced by that action, evaluating them and selecting the action that leads to the best-valued successor state(s).

Since in General Game Playing each game is different, there is no generally useful evaluation function and an agent has to construct a function automatically depending on the game. One possibility is to take the game's goal function that maps states to a goal value. However, since goal function in GDL is "crisp", it does not contain information to what extent a state fulfills the underlying goal condition. Moreover, it most often only distinguishes between terminal states. By fuzzifying the goal function both problems can be addressed.

Construction After receiving the rules we preprocess the goal conditions of the game to obtain an evaluation function. Since each goal condition describes an existentially quantified predicate logic expression, our algorithm operates on these expressions. The following expansion algorithm takes as first argument an existentially quantified expression and transforms it recursively into a tree-like structure that represent the evaluation function:

```

1 expand(true(Fluent), true(Fluent)) :- !.
2 expand(not(Expr), not(Child)) :- !,
3   expand(Expr, Child).
4 expand(Expr, and(Children)) :-
5   is_conjunction(Expr, Conjunctions), !,
6   expand_children(Conjunctions, Children).
7 expand(Expr, or(Children)) :-
8   is_disjunction(Expr, Disjunctions), !,
9   expand_children(Disjunctions, Children).
10 expand(Expr, R) :-
11   is_game_specific_pred(Expr, Expansions), !,
12   (Expansions = [OneExpansion] ->
13     expand(OneExpansion, R)
14   ;
15     R = or(Children)
16     expand_children(Expansions, Children)
17   ).
18 expand(Expr, reason(Expr)).
19 expand_children([], []).
20 expand_children([C|Children], [E|Expansions]) :-
21   expand(C, E),
22   expand_children(Children, Expansions).

```

The predicates `is_conjunction/2`, `is_disjunction/2` and `is_game_specific_pred/2` analyze whether the expression is what the predicate expects it to be and returns, if confirmed, as second argument a list of the constituting expressions.

So given `Expr=(a(b), c(X), d)` it would hold `is_conjunction(Expr, [a(b), c(X), d])` while `is_disjunction(Expr, Conjuncts)` would fail since `Expr` is not a disjunction.

Obviously, the algorithm needs a little more to work the intended way. For instance, the predicate `is_conjunction/2` must not split conjunctions where the conjuncts share variables. So an expression like `is_conjunction((a(X), b(X), c(Y)), Conjuncts)` should yield a list of two conjuncts, namely `(a(X), b(X))` and `c(Y)`. Consequently, the expression `(a(X), b(X))` is not seen as a conjunction by our algorithm, but rather as an expression that will be delegated to the reasoner. Also we must not expand recursive expressions in `is_game_specific_pred/2`.

In the case of Tic-Tac-Toe, we could e.g. pass the expression `goal(xplayer, 100)` as first argument to the `expand` algorithm and receive a structure that represents the fuzzified evaluation of whether a given state fulfills the winning condition for role `xplayer`. We will refer to such a structure as evaluation tree.

State Evaluation The following is an example how to use this evaluation tree to obtain a single value that represents a propositional fuzzy evaluation of the goal condition applied on the state. The evaluation tree is passed as first argument and the state we want to evaluate as second argument. As last argument we receive the fuzzy value. All predicates with the prefix `fz_` and the predicates `true/1` and `false/1` are defined along with the fuzzy logic in the next section.

```

1 eval(not(Child), S, V) :-
2   eval(Child, S, V1), not(V1, V).
3 eval(true(Fluent), S, V) :-
4   fluent_holds(Fluent, S),
5   !, fz_true(V).
6 eval(true(_Fluent), _, V) :- fz_false(V).
7 eval(and([], _, V) :- !, true(V).
8 eval(and([C|Children]), S, V) :-
9   eval(C, S, V1),
10  eval(and(Children), S, V2),
11  fz_and(V1, V2, V).
12 eval(or([], _, V) :- !, false(V).
13 eval(or([C|Children]), S, V) :-
14  eval(C, S, V1),
15  eval(or(Children), S, V2),
16  fz_or(V1, V2, V).
17 eval(reason(Expr), S, V) :-
18  expression_holds(Expr, S),
19  !, fz_true(V).
20 eval(reason(_Expr), _, V) :- fz_false(V).

```

Here we evaluate all leafs of the evaluation function structure (fluents and expressions) to the value defined by `fz_true` if they hold in the current state and to the `fz_false` value otherwise. These values are passed recursively up the evaluation structure using the predicates `fz_and`, `fz_or` and `not` and the value at the root node represents the fuzzy evaluation of the goal condition on the state. The predicates `true` and `false` just represent the upper and lower bound of the interval of truth values and must be set in accordance with the fuzzy logic.

Underlying Fuzzy Logic A useful propositional fuzzy logic could be any t-norm fuzzy logic, e.g.:

```

1 fz_true(0.9). true(1).
2 fz_false(0.1). false(0).
3 not(X, Y) :- Y is 1 - X.
4 fz_and(X, Y, Z) :- Z is Y * X.
5 fz_or(X, Y, Z) :- Z is X + Y - X * Y.

```

Fluxplayer (Schiffel and Thielscher 2007) also operates on the interval $[0, 1]$ of truth values, but uses an instance of the Yager family of t-norms.

We believe that the class of applicable fuzzy logics can be extended to more general classes than just t-norm fuzzy logics. In fact, for the GGP setting any “approximately correct” logic can be used since we will discuss features that return incorrect values with respect to the used fuzzy semantic, yet these features contribute well to the evaluation quality of a state. For the remainder of this paper we assume the t-norm fuzzy logic as given above.

Given the expansion and evaluation algorithms together with an appropriate propositional fuzzy logic, we can evaluate the degree of truth of the goal condition on a particular state. Practically, we obtain a measure to compare two states by comparing their state values and we can assume that the state with the higher state value is closer to the goal than the other state. Note however, the evaluation function up to now builds merely upon the logical structure of the goal expression. Distances or quantitative evaluations are ignored.

Ground-Instantiation Since the domains of all variables occurring in a game description are finite, it is possible to ground-instantiate a game by substituting any expression containing a variable with the disjunction of the expressions where that variable is substituted by all terms in the domain of the variable. We use ground-instantiation to deal with expressions that cannot be expanded further. In this way, it is e.g. possible to break down variable-sharing conjunctions to a disjunction of standard conjunctions and thus increase the part of the original predicate logic expression that can be evaluated using fuzzy logic. Note, however, that the number of ground-instantiations of an expression may grow exponentially with the number of variables. It is therefore sometimes necessary to use a mixed approach, instantiating some variables while leaving others uninstantiated. Expressions uninstantiated for whatever reason are directly evaluated by the reasoner.

After ground-instantiating an expression we may encounter (so called static) predicates in the resulting ground expressions that do not depend on the current state of the game. We can immediately evaluate them to true or false and hence do not need to include them in the evaluation function.

Our basic principle for constructing the evaluation tree is to keep all expressions uninstantiated for as long as possible to avoid growing the size of the evaluation function.

3 Features

We will present several features and show how to detect them in a predicate logic expression. This will require us to modify the `expand` algorithm given above. Due to space

restrictions we cannot give a formal description of the requirements an expression needs to fulfill in order to be identified as a feature. However, we will discuss them informally in the corresponding paragraph. For each feature discussed we will give an example on how to utilize it by presenting additional rules for the above `eval` algorithm.

The features themselves represent heuristics and there are always situations in which the features produce the opposite of what a perfect state evaluation function would return. However, this problem is a general problem of game playing agents. It is addressed by using only features that predominantly improve the evaluation quality and by using many features such that over- and underestimations of feature values neutralize themselves.

3.1 Expression Features

We start with expression features that are derived from expressions containing variables, namely fluents with variables, conjunctions of conjuncts that share variables, and predicates containing variables. For the detection of expression features we specialize the `expand` algorithm such that instead of further expansion via ground-instantiation or simple delegation to the reasoner, we interpret the expression directly as feature, and add this feature as leaf to the evaluation tree. This has one two main effects: First, by avoiding ground-instantiation the number of nodes in the evaluation tree is smaller and the function therefore faster to evaluate. And second, the resulting feature is more expressive than its otherwise expanded alternative.

In other words, evaluation becomes faster and estimates better at the same time.

Solution Cardinality

```
1 open :- true(cell(X, Y, b)).
```

The basic element for querying the current state of the match is the `true` statement with a fluent as argument. Since we apply ground-instantiation as late as possible, we may encounter fluents with non-ground arguments. An example is the above rule from the game Tic-Tac-Toe. The `open/0` predicate is used to determine whether there are blank cells and whether the match can therefore continue.

We impose a first interpretation on the body of the rule by counting the number of solutions in the expression. So instead of evaluating the expression to true or false, we evaluate it to false if there are no solutions, but give it a higher value for the more solutions there are. Assuming that states are relatively stable and the set of true fluents does not change radically from one state to another, a higher solution cardinality indicates that the expression is more likely to be true in successor states of the state under inspection. Besides, the evaluation function is smaller than its ground-instantiated variant (one node instead of 9 in the case of Tic-Tac-Toe) and the evaluation is therefore also faster than a nine-fold query of each individual ground fluent `cell(1, 1, b), ... cell(3, 3, b)`.

To use the feature, we change the `eval(true(Fluent), S, V)` rule in our evaluation

function such that it is equal to the value of a standard disjunction of the ground fluents.

For the `open/0` example, the evaluation would look like:

```
1 eval(true(cell(X, Y, b)), S, V) :-
2   count_solutions(cell(X, Y, b), S, SolCnt),
3   count_instances(cell(X, Y, b), Inst),
4   fz_generalized_or(SolCnt, Inst, V).
```

`count_instances/2` returns the number of valid instances for first argument (in this case 9) and `fz_generalized_or/3` is a helper that calculates the fuzzy value of a disjunction that is `Inst-SolCnt` times false and `SolCnt` times true.

We apply this interpretation whenever we encounter fluents with variables or expressions that we would otherwise pass to the reasoner. This feature is the same as the one Clune proposed (Clune 2007) under the same name. In his approach, however, solution cardinality features were constructed by first considering all expressions as solution cardinality features, then evaluating specific feature properties (stability, correlation to goal values and computation cost) on a number of test states and finally removing all candidate features that do not exhibit the desired properties to a sufficient degree. In contrast, our approach is much faster since the set of candidate features is greatly reduced and the correlation to goal values is implicitly determined through the context the fuzzy logic provides.

Order Conjunctions with shared variables adhere to a number of patterns. One example is a goal expression in the game Checkers which states

```
1 goal(white, 100) :-
2   true(piece_count(white, W)),
3   true(piece_count(black, B)),
4   greater(W, B).
```

The `piece_count` fluent keeps track of the number of pieces each role has and `greater` is the axiomatized variant of the greater-than relation. Thus white achieves 100 points if it has more pieces than black.

Since domains typically have no more than a few hundred elements, we can quickly prove that the relation expressed by the predicate `greater/2` is antisymmetric, functional, injective and transitive and thus conclude that it represents a total order, possibly with the exception of the reflexive case. Therefore, we can map all elements in the order to natural numbers. Assuming that the fluents `piece_count(white, _)` and `piece_count(black, _)` occur exactly once in each state, we can identify an expression of the above type as order expression. Generally spoken, we identify an expression as order feature if we encounter a conjunction with sharing variables where at least one fluent variable occurs in a static predicate that represents a total order. In addition we require the fluents in the conjunction to occur exactly once in any state.

We evaluate the above example the following way:

```
1 eval(order(goal(white, 100)), S, V) :-
2   fluent_holds(piece_count(white, W), S),
```

```

3 fluent_holds(piece_count(black, B), S),
4 term_to_int(greater_2, W, WI),
5 term_to_int(greater_2, B, BI),
6 domain_size(greater_2, Size),
7 V is 0.5 + (WI-BI)/(2*Size).

```

The advantage of this type of order heuristics is most importantly a finer granularity in comparisons of elements: The return value V is higher for higher differences between the terms W and B and lower if both are approximately equal. A ground-instantiated version of this evaluation would not be able to distinguish between differences of +2 or +20. Besides, we reduce the size of the evaluation function by avoiding ground-instantiation. In this case, since there are 13 possible values of the `piece_count` fluent (0 to 12 pieces) for each role, we express 169 ground evaluations by one heuristic order expression.

The only disadvantage is that we do not see an easy way to express this relation while sticking to the truth values required by our fuzzy logic, thus we lose correctness at this point.

A structurally different feature with the same expressiveness is also detected by (Clune 2007) who looks for comparative features if solution cardinalities are found. However, it is unclear if his agent is able to exploit orders on other quantitative statements. Our approach is similar to that of (Schiffel and Thielscher 2007).

Relative Distances I - Distance Between Fluents Another pattern for conjunctions with shared variables is that of the relative distance. If two fluents occur in a conjunction and share the same variables at the same argument positions then these are candidates for relative distances. An example is the winning rule for `inky` in the game Pac-Man (“pacman3p”):

```

1 goal(inky, 100) :-
2   true(location(pacman, X, Y)),
3   true(location(inky, X, Y)).

```

In order to confirm that this pattern can be interpreted as relative distance pattern, we have to make sure that it is possible to calculate distances on the argument positions of the fluent where the variables occur. We do this by identifying, how the arguments of the fluent evolve over time. If, for instance, the fluent `location/3` represents a metric board where Pac-Man can move north, then there is most probably a rule in the GDL description of the game such that

```

1 next(location(pacman, X, Z)) :-
2   true(location(pacman, X, Y)),
3   a_predicate(Y, Z).

```

By identifying static predicates such as `a_predicate/2`, one can derive a graph where all valid instantiations of `a_predicate/2` are the edges.

Most often, the relation represented by `a_predicate/2` is an antisymmetric, functional, injective and irreflexive and hence the domain a successor domain. The transitive closure of this domain can then be represented by natural numbers and distances easily calculated. Much like the in order

heuristics, a relative distance interpretation of the above example could look like this:

```

1 eval(relative_dist(goal(inky, 100)), S, V) :-
2   fluent_holds(location(pacman, XP, YP), S),
3   fluent_holds(location(inky, XI, YI), S),
4   term_to_int(location_3_arg_2, XP, XPI),
5   term_to_int(location_3_arg_2, XI, XII),
6   domain_size(location_3_arg_2, XSize),
7   XDist is abs(XPI-XII)/XSize,
8   term_to_int(location_3_arg_3, YP, YPI),
9   term_to_int(location_3_arg_3, YI, YII),
10  domain_size(location_3_arg_3, YSize),
11  YDist is abs(YPI-YII)/YSize,
12  metric(XDist, YDist, Dist),
13  V is 1-Dist.

```

Basically, we evaluate the distances per argument, combine these using a metric and give a higher return for lower distances.

There are also predicates that do not represent a functional or injective relation. In this case we cannot construct a mapping of the domain elements of the binary static predicate to the natural numbers. To use the predicate information anyway, we construct a graph where all relations given by the predicate are considered edges of the graph. The shortest path in the graph between two arbitrary domain elements represents the distance. If there is no path, then we consider the distance infinite. Instead of using the domain size to normalize the distance, we then have to use the longest of all shortest distances between any two domain elements.

Note that we may also detect information regarding the direction: The relation expressed by `a_predicate(Y, Z)` represents, in fact, a directed edge from Y to Z , assuming that Z occurs in the fluent in the head of the next rule and Y in the body. This means that in the above evaluation lines 7 and 11 are only right if Pac-Man and Inky can move north and south. In case, they can only move in one direction, the distance is either infinite or the absolute of the determined relative distance.

Finally, the above evaluation only works if the fluents containing variables unify with exactly one instance in each state. In case there is no such fluent in the state, the distance is infinite and the result therefore $V = 0$. If there are more instances, we can calculate the pairwise relative distances and aggregate them using the average or the minimum.

The advantage of this feature are again a reduction of the size of the evaluation tree and a much greater expressiveness: Both the uninstantiated and the ground-instantiated version of e.g. the goal in Pac-Man return a constant value and thus provides no feedback in cases where Pac-Man and Inky are not on the same square. Only a relative distance evaluation returns useful fuzzy values in this case.

Though theoretically feasible in other approaches, distances between variable fluents were mentioned directly only in (Kuhlmann, Dresner, and Stone 2006) where it is used as a candidate feature. However, it remains unclear when he chooses to use the feature and how he uses the feature output in the evaluation function. In contrast, both is taken care of in our approach since we derive it from a specified expression and integrate its output via fuzzy logic.

3.2 Fluent Features

Another type of features that we identify are fluent features. This type of feature is derived from fluents and modifies an existing `true(Fluent)` expression: The standard `fluent_holds/2` evaluation is substituted by a more complicated procedure, rendering the evaluation typically more expensive but also more fine-grained.

Relative Distances II - Distance To Fixed Fluents A specialization of the relative distance between two fluents is the relative distance towards a fixed point. In this pattern we do not find two fluents with shared variables but a single fluent without variables. The predicate `timeout/0` is part of the goal condition of a version of MummyMaze (“mummymaze2p-comp2007”):

```
1 timeout :- true(step(50)).
```

The step fluent argument is increased via a static successor predicate that we identify as described in Relative Distances I. Hence the evaluation in this case is just a simplified version of the evaluation for relative distances between fluents. Another example could be a modified Pac-Man where Pac-Man has to reach a specific coordinate to win the match, e.g.

```
1 goal(pacman, 100) :-
2   true(location(pacman, 8, 8)).
```

In fact, we can generalize this pattern to all fluents, such that for every fluent that is evaluated by `fluent_holds/2` we may try to determine whether we find an order over some of its arguments. If confirmed, we use distance estimation on the fluent. The strength of the distance estimation lies in the fact that although only `location(pacman, 8, 8)` occurs as goal in the goal condition, we can derive useful information from *arbitrary* fluents `location(pacman, X, Y)` holding in the current state. The evaluation is based on the hypothesis that fluents where $X \approx 8$ and $Y \approx 8$ are more likely to lead to the desired goal fluent.

Since ground fluents occur frequently in virtually every goal condition, we must however be aware of side effects. While relative distances in Tic-Tac-Toe pose no problem as the cell coordinates are not connected via binary successor predicates, the situation is different in a Connect Four variant where the following next rule connects vertically adjacent cells:

```
1 next(cell(X, Y2, red)) :-
2   does(red, drop(X),
3     cell(X, Y2, blank),
4     succ(Y1, Y2),
5     cell(X, Y1, red)).
```

The correct interpretation of the rule states that if a cell is occupied by red, the above cell can be occupied by red as well. However, the distance interpretation may see player red move from cell $(X, Y1)$ to cell $(X, Y2)$ just as Pac-Man does. The difference by which we can distinguish both interpretations lies in the fact that in Connect Four all reachable ground fluents of the form `cell(X, Y, red)` occur in the goal condition (as part of the definition of a line of four discs), while in a game with a distant goal there are only a

few fluents that represent the goal, most do not occur in the goal condition.

A second problem arises for fluents with variables. In the example of Pawn Whopping (“pawn_whopping”) the player wins if one of his pawns (symbolized by an $\times$) reaches any cell (variable Y) of the 8th rank, symbolized by the 8.

```
1 goal(x, 100) :- true(cell(Y, 8, x)).
```

Again, we find there are static successor predicates that impose an order over the first two arguments of the fluent, enabling us to calculate distances. However, here we have a conflict between features, as the expression could be evaluated using the solution cardinality interpretation and the distance interpretation. Of course, only distance interpretation makes sense since the goal is to have a pawn at the 8th rank and the match ends once the goal is achieved. A solution cardinality of 2 will thus never occur.

We distinguish both patterns generally by applying the solution cardinality only if all ordered arguments (that is, arguments of the fluent where an underlying order could be found) of the fluent are variables since such an expression typically counts the number of pieces on a board. In all other situations we use the distance interpretation.

Persistence Another detail for improving the evaluation of ground-instantiated fluents is their persistence. Consider the following example from Tic-Tac-Toe:

```
1 next(cell(X, Y, C)) :-
2   true(cell(X, Y, C)), C \= b.
```

The rule states that once a cell is marked (with an $\times$ or an $\circ$), the cell remains marked for the rest of the match. We call this fluent feature “persistence” as the fact encoded by the fluent persists through the rest of the match. We distinguish between persistent true and persistent false fluents: Once a persistent true fluent holds, it holds in all successor states. In contrast, if a persistent false fluent does not hold in the current state, then it also does not hold in its successor states. An example for a persistent false fluent is the fluent `cell(X, Y, b)` representing unmarked (blank) cells in Tic-Tac-Toe.

Persistence can be used in several ways to improve the evaluation function: First, persistent fluents have a higher impact on a future state than non-persistent fluents. Hence, a higher evaluation is justified.

```
1 eval(true(Fluent), S, V) :-
2   fluent_holds(Fluent, S), !,
3   (persistent_true(Fluent) -> true(V);
4     fz_true(V)).
5 eval(true(Fluent), _, V) :-
6   (persistent_false(Fluent) -> false(V);
7     fz_false(V)).
```

Beside persistent fluents being more stable than their non-persistent counterparts, persistence can also speed up the evaluation of a state: Once a persistent false fluent holds, any conjunction with this fluent as positive (i.e. non-negated) conjunct is also persistent false. Therefore, we can skip evaluating other conjuncts in the same conjunction. This same

effect holds for negative occurrences of persistent true fluents.

Therefore we modify the evaluation function as follows:

```

1 eval(and([]), S, V) :- true(V) .
2 eval(and([C|Children], S, V) :-
3   eval(C, S, V1),
4   (V1 == 0 ->
5     V = 0
6   ) ;
7   eval(and(Children), S, V2),
8   fz_and(V1, V2, V)
9 ).

```

Naturally the idea is analogously applicable for persistent true fluents in disjunctions.

Note that distance estimations that return an infinite distance are also persistent false fluents. For proving that fluents are persistent we use the approach presented in (Thielscher and Voigt 2010).

4 Evaluation

We evaluate the effectiveness of each feature proposed by applying them in a number of games. Our hypothesis is that they increase the probability of winning against a benchmark player at least in some games. As candidate games we used only games played in the GGP championships 2005-2009. The players had a 60 seconds preparation time and 10 seconds for each move. After half of the matches, roles were switched (e.g. the party playing black plays white and vice versa) to eliminate advantages of specific roles. Both agents use the same search algorithm (α - β search).

As evaluation function we use a neural network that correctly represents propositional logic and presents similar fuzzy properties as e.g. t-norm fuzzy logic (Michulke and Thielscher 2009). We further limited the evaluation function structure to depth 8 and size 500 to cut off the most disadvantageous parts of the function. The rationale behind is that nodes in depths higher than 8 have little impact on the state evaluation, but are still expensive to evaluate. For the same reason we skip the expansion of expressions if the resulting function would surpass 500 nodes. The values of the limits were determined empirically and ensure a reasonable evaluation speed of several hundred states per second. Any remaining unexpanded expression was delegated to the reasoner.

The evaluation was performed by setting up a player against a handicapped version of itself. The handicap was realized by deactivating one of the features discussed in this paper. Both players ran on the same machine, a Core 2 Duo E 8500 at 3.16GHz with 4GB RAM, each player had 1.5GB RAM to its avail.

The left chart of Figure 1 shows the results of 40 matches in the given games. The capital letters indicate what feature was turned off at the handicapped player and refer to the expression features *solution cardinality* (SC), *order* (Ord) and *distance between fluents* (DistBtw), and the fluent features *distance towards a fixed fluent* based on natural numbers (ND) or graphs (GD), and *persistence* (Pers). The length of the bar shows to what extent the win rate was shifted in favor

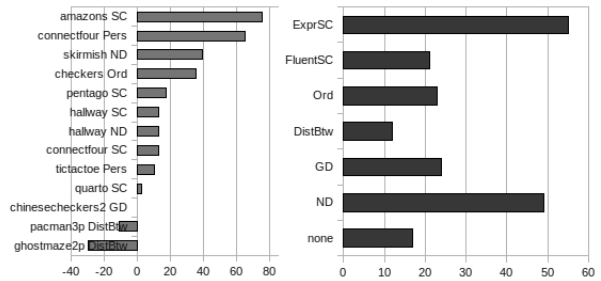


Figure 1: Left: Win Rate Increase against Handicapped Player, Right: #Features Detected in 198 Games

of the standard player when playing against the handicapped version. E.g. a value of 20% means that instead of winning 50% of all won points in the matches, the standard version now wins 60%. Consequently, the handicapped version just won 40 %.

For comparison, the chance of flipping 40 ideal coins and getting 26 times or more heads is 4.06%. So there is 4.06% chance that a win rate increase of $26/20 - 1 = 30\%$ is a mere coincidence and both agents play equally well.

We can see that in most of the games the win rate increases against the handicapped agent. The decreasing performance in the worst three games is distance related and has a simple reason: Depending on the type of search, distance information can be obtained otherwise. If e.g. our evaluation function has no distance information in Pac-Man, then all states are evaluated equal. Our architecture therefore uses the maximum of all values reachable within the search horizon to tie-break the situation. This means that once a goal state is found (even though it would require the opponent to play in our favor) the state evaluation tends towards this state. Therefore in games of limited complexity this tie-breaking mechanism in combination with the additional computational effort to calculate the distance is responsible for the underperformance. This argument is supported by the fact that in more complex games distance actually does make a difference.

The right side of Figure 1 shows in how many games features of the given type appear. ExprSC and FluentSC here distinguish between solution cardinality applied to expressions passed to the reasoner and those applied to fluents with variables¹. We can see that there were 17 games where no feature was found. Among these are four Tic-Tac-Toe variants where persistence could not be proved within the given amount of time. All other games without features had ground goals (e.g. four Nim versions and four Blocksworld versions). Note that in these cases our fuzzy logic itself already provides good search guidance.

5 Summary

We presented a general and integrated method of how to transform a predicate logic expression to an evaluation func-

¹The complete set of games can be found under <http://euklid.inf.tu-dresden.de:8180/ggpserver>

tion. We focused on detecting features in the expression, describing other conditions that must be met to use the features and proposing a number of short algorithms that show how to use the feature information. In contrast to other general game players, no expensive feature evaluation phase was needed.

We showed the benefits of applying such features on a set of specific games and measured, how general the features are on a set of 198 games submitted to the Dresden GGP server up to January 2011.

For the future, we consider it crucial to also measure the effects of the features on run-time and decide, depending on the complexity of the game, whether the feature is advantageous or not. Besides, we believe that an architecture that dynamically decides what interpretation to use is maybe suited best for the different interpretations. An idea in this scenario would be to see first whether a fluent or an expression holds. If it holds, return a value representing how often it holds and how stable (i.e. persistent) it is. If it doesn't, evaluate a fluent using distances and an expression using partial evaluation based on e.g. how many conjuncts of the expression hold. Finally, there is no reason to assume that distances cannot be calculated on the arguments of predicates, leaving space for further improvement.

References

- Clune, J. 2007. Heuristic evaluation functions for general game playing. In *AAAI*. Vancouver: AAAI Press.
- Finnsen, H., and Björnsson, Y. 2008. Simulation-based approach to general game playing. In *AAAI*. AAAI Press.
- Finnsen, H., and Björnsson, Y. 2010. Learning simulation control in general game-playing agents. In Fox, M., and Poole, D., eds., *AAAI*. AAAI Press.
- Kuhlmann, G.; Dresner, K.; and Stone, P. 2006. Automatic Heuristic Construction in a Complete General Game Player. In *Proceedings of the Twenty-First National Conference on Artificial Intelligence*, 1457–62. Boston, Massachusetts, USA: AAAI Press.
- Love, N.; Hinrichs, T.; Haley, D.; Schkufza, E.; and Genssereth, M. 2008. General game playing: Game description language specification. Technical Report March 4. The most recent version should be available at <http://games.stanford.edu/>.
- Michulke, D., and Thielscher, M. 2009. Neural networks for state evaluation in general game playing. In *ECML PKDD '09: Proceedings of the European Conference on Machine Learning and Knowledge Discovery in Databases*, 95–110. Berlin, Heidelberg: Springer-Verlag.
- Schiffel, S., and Thielscher, M. 2007. Fluxplayer: A successful general game player. In *Proceedings of the National Conference on Artificial Intelligence*, 1191–1196. Vancouver: AAAI Press.
- Thielscher, M., and Voigt, S. 2010. A temporal proof system for general game playing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, 1000–1005. Atlanta: AAAI Press.

A Forward Chaining Based Game Description Language Compiler

Abdallah Saffidine and Tristan Cazenave

LAMSADE, Université Paris-Dauphine
Paris, France

Abstract

We present a first attempt at the compilation of the Game Description Language (GDL) using a bottom-up strategy. GDL is transformed into a normal form in which rules contain at most two predicates. The rules are then inverted to allow forward chaining and each predicate will turn into a function in the target language. Adding a fact to the knowledge base corresponds to triggering a function call.

Permanent and persistent facts are detected and enable to speed up the deductions. The resulting program performs playouts at a competitive level. Experimental results show that a speed-up of more than 40% can be obtained without sacrificing scalability.

1 Introduction

General Game Playing (GGP) has been described as a *Grand Artificial Intelligence (AI) Challenge* (Genesereth and Love 2005; Thielscher 2009) and it has spanned research in several directions. Some works aim at extracting knowledge from the rules (Clune 2007), while GGP can also be used to study the behavior of a general algorithm on several different games (Méhat and Cazenave 2010). Another possibility is studying the possible interpretation and compilation of the Game Description Language (GDL) (Waugh 2009; Kissmann and Edelkamp 2010) in order to process game events faster.

While the third direction mentioned does not directly contribute to AI, it is important for several reasons. It can enable easier integration with playing programs and let other researchers work on GGP without bothering with interface writing, GDL interpreting and such non-AI details. Having a faster state machine may move the speed bottleneck of the program from the GGP module to the AI module and can help performance distinction between different AI algorithms. Finally, as GDL is a high level language, compiling the rules to a fast state machine and extracting knowledge from the rules are sometimes similar things. For instance, factoring a game (Günther, Schiffel, and Thielscher 2009) can be considered a part of the compilation scheme.

GDL is compiled into forward chaining rules that contain only two predicates. As an optimization, a so-called *temporization* procedure is carried on in which permanent and persistent predicates are detected. The whole compilation

process is very quick even and the resulting program performs playouts at a competitive level.

1.1 Game Automaton

The underlying model of games assumed in GGP is the Game Automaton (GA). It is a general and abstract way of defining games to encompass puzzles and multiplayer games, be it turn-taking or simultaneous. Informally, a GA is a kind of automaton with an initial state and at least one final state. The set of outgoing transitions in a state is the cross-product of the legal moves of each player in the state (non-turn players play a *no-operation* move). Final states are labelled with a reward for each player.

1.2 Objective and previous work

We focus on the compilation of rulesheets in the GDL into GAs. More precisely, the compiler described in this work takes a game rulesheet written in GDL as an input and outputs an OCAML module that can be interfaced with our playing program also written in OCAML. The module and the playing program are then compiled to native code by the standard OCAML compiler (Leroy et al. 1996) so that the resulting program runs in reasonable time. The generated module exhibits a GA-like interface. Figure 1 sums up the normal usage of our compiler and Figure 2 details the various steps between a rulesheet and an executable.

The usual approach to the compilation of GDL is to compile it to PROLOG and to use resolution to interpret it (Méhat and Cazenave 2011). Other approaches have been tried such as specialization and compilation of the resolution mechanism to the C language (Waugh 2009).

1.3 Outline of the paper

The remaining of this article is organized as follows: we first recall the definition of the Game Description Language, then we present the organization of the knowledge base. Section 4 describes the various passes used by our compiler to generate target code from GDL source code. Finally, we briefly present some experimental considerations.

2 Game Description Language

The Game Description Language (Love, Hinrichs, and Genesereth 2006) is based on DATALOG and allows to define

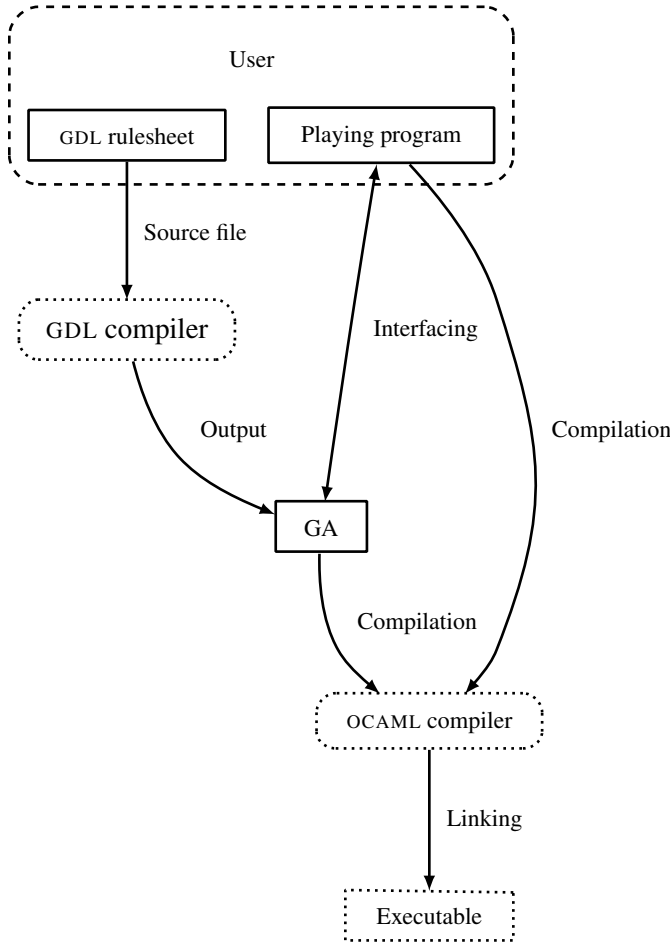


Figure 1: Interactions between a user and the GDL compiler.

Name	Arity	Appearance
does	2	body
goal	2	base, body, head
init	1	base, head
legal	2	base, body, head
next	1	head
role	1	base
terminal	0	base, body, head
true	1	body

Table 1: Predefined predicates in GDL with their arity and restriction on their appearance.

a large class of GAs (see Section for a formal definition of a GA). It is a rule based language that features function constants, negation-as-a-failure and variables. Some predefined predicates confer the dynamics of a GA to the language.

2.1 Syntax

A limited number of syntactic constructs appear in GDL.¹ Predefined *predicates* are presented in Table 1. Function constants may appear and have a fixed arity determined by context of the first appearance. Logic operators are simply `or`, `and` and `not`; they appear only in the body of rules. Existentially quantified variables may also be used bearing some restrictions defined in Section 2.1. Rules are compose of a head term and a body made of logic terms.

A GDL source file is composed of a set of grounded terms that we will call B for *base facts* and a set of rules. The Knowledge Interchange Format is used for the concrete syntax of GDL.

The definition of GDL (Love, Hinrichs, and Genesereth 2006) makes sure that each variable of a negative literal also appears in a positive literal. The goal of this restriction is probably to make efficient implementations of GDL easier. Indeed, it is possible to wait until every variable in a negative literal are bound before checking if the corresponding fact is in the knowledge base. Put another way, it enables to deal with the negation by only checking for ground terms in the knowledge base. This property is called *safety*.

2.2 Semantics

The base facts B defined in the source file are always considered to hold. The semantics also make use of the logical closure over the rules defined in the files, that is at a time τ , the rules allow to deduce more facts that are true at τ based on facts that are known to hold at τ .

The semantics of a program in the GDL can be described through the GA formalism as follows

- The set of players participating to the game is the set of arguments to the predicate `role`.
- A state of the GA is defined by a set of facts that is closed under application of the rules in the source files.

¹We depart a bit from the presentation in (Love, Hinrichs, and Genesereth 2006) to ease the sketch of our compiler.

- The initial state is the closure over the facts that are arguments to the predicate `init`.
- Final states are those in which the fact `terminal` holds.
- For each player p and each final state s , exactly one fact of the form `goal( $p, o_p$ )` holds. We say that $0 \leq o_p \leq 100$ is the reward for player p in s . The outcome o in the final state is the tuple $(o_{p_1}, \dots, o_{p_k})$.
- For each player p and each state s , the legal moves for p in a state s are $L_s(p) = \{m_p \mid \text{legal}(p, m_p) \text{ holds in } s\}$
- The transition relation is defined by using the predicates `does` and `next`. For a move $m = (m_{p_1}, \dots, m_{p_k})$ in a state s , let q be the closure of the following set of facts : $s \cup \{\text{does}(p_1, m_{p_1}), \dots, \text{does}(p_k, m_{p_k})\}$. Let n be the set of fact f such that `next( $f$ )` holds in n . The resulting state of applying m to s is the closure of the set $\{\text{true}(f) \mid f \in n\} \cup B$.

2.3 Stratification

Game rules written in GDL are stratified. It means that all the facts of the lower strata have to be deduced before starting to deduce the facts of the upper strata. Stratification is a way of coping with negation as a failure. When using backward chaining the user does not need to pay much attention to the stratification of GDL. However when using forward chaining, stratification is important since the absence of a fact cannot be concluded until every deduction on the lower strates have been carried on.

3 Runtime environment

3.1 Managing the knowledge base

In a bottom-up evaluation scheme, it is necessary to keep a data structure containing the ground facts that were found to hold in the current state. This knowledge base constitutes the biggest part of the runtime of our system. It needs to satisfy the following interface.

- An `add` procedure taking a ground fact f and adding f to the knowledge base.
- An `exists` function taking a ground fact f and indicating whether f belongs to the knowledge base.
- A `unify` function taking a fact f with free variables and returning the set of ground facts belonging to the knowledge base that can be unified with f .

It is possible to implement such a module based on a hashing of the ground facts. It allows an implementation of `add` and `exists` in constant time, but `unify` is linear in the number of present ground facts as it consists in trying to unify every fact present in the knowledge base one after another.

It is also possible to implement such a module based on a trie by first linearizing the ground facts into lists. `unify` is in this case logarithmic, but `add` and `exists` are slower.

We implemented both approaches and the first one performed slightly better. It is probably far from optimal but a better alternative could not be found yet.

3.2 Temporization of the facts

Permanent facts are facts that are present at the beginning of a game in the rule and that never change during a game. For example in some games there is a need to compare small numbers. A set of facts describes the comparison of all pairs of numbers. These facts never change and they are always present. It is not necessary to remove them and deduce them again after playing a move. Separating them from the set of facts that can change and considering them as always true can gain some time.

Persistent facts (Thielscher and Voigt 2010) are facts that always remain true once they are deduced. They are not present at the beginning of a game. They are deduced after some moves, and once they have been deduced they are present in all the following states. An example of a persistent fact is the alignment of two pieces of the same color in the game `doubletictactoe`. Once an alignment has been deduced, it will remain valid until the end of the game. So avoiding to deduce it again can gain some time.

We call *ephemeral facts* the remaining facts. That is, ephemeral facts can appear at some point in the knowledge base and then disappear as the game proceeds.

3.3 Labelling the predicates

Let $\mathbb{B}$, $\mathbb{P}$, and $\mathbb{E}$ be respectively the sets of permanent, persistent, and ephemeral facts that can appear in game. In the following, we assume the sets are disjoint and every possible fact belongs to either set.

It is not straightforward to compute these sets at compile time without instantiating the rules. On the other hand, it is possible to use approximation for these sets. We will call $\mathbb{B}'$, $\mathbb{P}'$, and $\mathbb{E}'$ the approximated sets. An approximation is *conservative* if the following relations hold : $\mathbb{E} \subseteq \mathbb{E}'$, $\mathbb{E} \cup \mathbb{P} \subseteq \mathbb{E}' \cup \mathbb{P}'$, and $\mathbb{E} \cup \mathbb{P} \cup \mathbb{B} \subseteq \mathbb{E}' \cup \mathbb{P}' \cup \mathbb{B}'$.

By default, it is possible to consider that every fact is ephemeral. This approximation takes $\mathbb{E}' = \mathbb{E} \cup \mathbb{P} \cup \mathbb{B}$ and $\mathbb{P}' = \mathbb{B}' = \emptyset$ and is clearly conservative. Distinguishing persistent facts can accelerate the resulting engine since it acts as some sort of tabling or memoization. Similarly, when a game is restarted, persistent facts are removed from the knowledge base but permanent facts can be kept.

A more elaborate approximation consists in labelling each predicate with b , p , or e and assigning a fact f to $\mathbb{B}'$, $\mathbb{P}'$ or $\mathbb{E}'$ depending on the predicate it is based on. We use the fixpoint procedure presented in Algorithm 1. The algorithm begins with pattern matching to have a few starting persistent labels and asserts that predicates in the upper strates are considered ephemeral. The fixpoint part then deduces the label of the predicate in a conclusion part of a rule based on the labels of the hypotheses. It uses the following order on temporal labels : $b < p < e$. Permanent labels are first obtained from rules with no hypothesis. The update operator in the fixpoint procedure is increasing which ensures its termination. Finally, predicates that could not be proved to belong to $\mathbb{B}$ or $\mathbb{P}$ are labelled ephemeral.

Input: set of rules Γ
Result: A mapping from predicates to $\{b, p, e\}$
 Start with an empty mapping;
 Bind next to e ;
for each rule $next(q(args)) \leftarrow q(args)$ **do**
 Bind q to p ;
end
for each rule $q \leftarrow hyps$ **do**
 if hyps contains a negative literal **then**
 Bind q to e ;
 end
end
while A fixpoint has not been reached **do**
 for each rule $q \leftarrow r \wedge \dots \wedge r'$ **do**
 if $r \wedge \dots \wedge r'$ are bound **then**
 let β be the set of bindings for $r \wedge \dots \wedge r'$;
 Add the current binding of q to β if it exists;
 Bind q to the maximal element of β or to b
 if β is empty;
 end
 end
end
for each rule $q \leftarrow \dots$ **do**
 if q is not bound **then**
 Bind q to e ;
 end
end
return the completed mapping

Algorithm 1: Fixpoint to label predicate with temporal status. We sometimes omit predicate arguments for the sake of presentation.

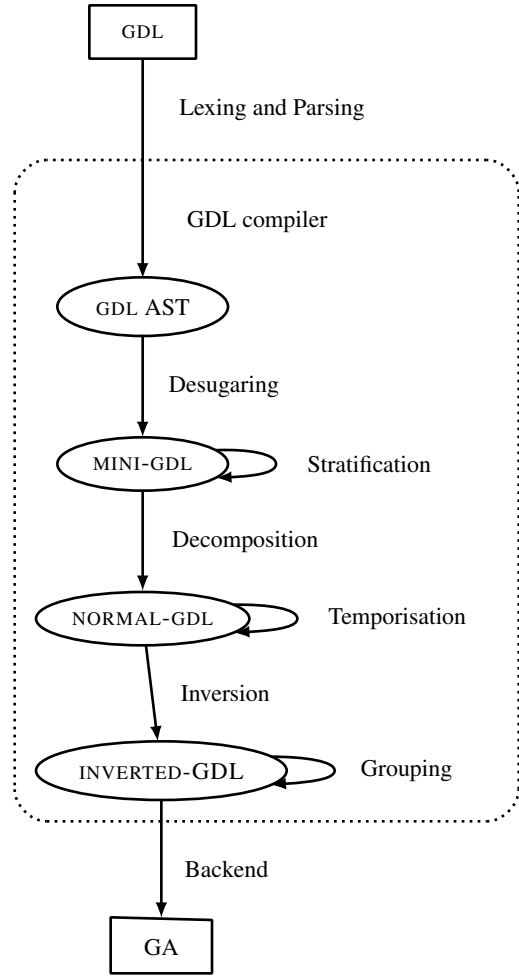


Figure 2: Steps and transformations between a GGP program written in GDL and the GA written in the output language.

4 Intermediate languages

Translating GDL programs to programs in the target language can be decomposed into several steps (see Figure 2). Each of these steps corresponds to the translation from one language to another. We used three intermediate languages in this work. The first one, MINI-GDL, is a version of GDL without syntactic sugar. In the second intermediate language, NORMAL-GDL, the rules are decomposed until a normal form with at most two hypotheses per rule is reached. The transition between a declarative language and an imperative one takes place when the program is transformed into INVERTED-GDL. Finally the program in the INVERTED-GDL is transformed in an abstract syntax tree of the target language.

4.1 Desugaring

MINI-GDL is a subset of GDL that has the same expressive power. For instance, there is no equal predicate in GDL and many rulesheets use the negation of the distinct pred-

icate to express equality. On the other hand, a literal in a MINI-GDL rule is therefore either a regular predicate, the negation of a regular predicate or the special *distinct* predicate. Disjunctions in rules are no longer possible.

The right hand side of a rule in GDL contains a logical formula made of an arbitrary nesting of conjunctions, disjunctions and negations.² The first step in transforming a rule from GDL to MINI-GDL is to put it in Disjunctive Normal Form (DNF).

A rule in DNF can now be split over as many subrules as the number of disjunctions it is made of. Indeed a rule with a conclusion c and a right hand side made of the disjunction of two hypotheses h_1 and h_2 is logically equivalent to two rules with h_1 and h_2 as hypotheses and the same conclusion c : $\{c \leftarrow h_1 \vee h_2\} \equiv \{c \leftarrow h_1, c \leftarrow h_2\}$.

A rule involving equalities can be turned into an equivalent rule without any equality. The transformation is made of two recursive processes, a substitution and a decomposition. When we are faced with an equality between t_1 and t_2 in a rule r , either at least one of the two terms is a variable (wlog. we will assume t_1 is a variable) or both are made of a function constant and a list of subterms. In the former case the substitution takes place: we obtain an equivalent rule by replacing every instance of t_1 in r by t_2 and dropping the equality. In the latter case, if the function constants are different then the equality is unsatisfiable and r cannot fire. Otherwise, we can replace the equality between t_1 and t_2 by equalities between the subterms of t_1 and the subterms of t_2 . Note that function constants with different arities are always considered to be different. We can carry this operation until the obtained rule does not have any equality left.

4.2 Ensuring stratification

We have seen in Section 2.3 that it was necessary to take care of stratification in our bottom-up approach. We start by labelling each predicate with its strate number. The labels can be obtained with a simple fixpoint algorithm that we do not detail. Then, we can use the following trick: we create a new predicate `strate(s)` for each possible strate s , we modify slightly the rules so that before the negation of each predicate p labelled with strate s the predicate `strate(s)` appears.

For instance, assume in the rule `foo ← bar ∧ ¬ baz` that the predicate `baz` is labelled with strate 1. The transformation results in the rule `foo ← bar ∧ strate(1) ∧ ¬ baz`.

After the rules are thus explicitly stratified, the evaluation scheme becomes straightforward. Apply the rules to obtain every possible fact, then add the fact corresponding to the first strate to the knowledge base, apply the rules again, then add the fact corresponding to the second strate and so on until the last strate fact is added.

4.3 Decomposition

GDL is built upon DATALOG, therefore techniques applied to DATALOG are often worth considering in GDL. Liu and

²Although there are some restriction on the negation possibilities.

Stoller (2009) presented a decomposition such that each rule in normal form is made of at most two literals in the right hand side.

Let $r = c \leftarrow t_1 \wedge t_2 \wedge t_3 \wedge \dots \wedge t_n$ be a rule with $n > 2$ hypotheses. We create a new term t_{new} and replace r by the following two rules. $r_1 = t_{\text{new}} \leftarrow t_1 \wedge t_2$ and $r_2 = c \leftarrow t_{\text{new}} \wedge t_3 \wedge \dots \wedge t_n$. Since variables can occur in the different terms and in c , t_{new} needs to carry the right variables so that c is instantiated with the same value when r is fired and when r_1 then r_2 are fired. This is achieved by embedding in t_{new} exactly the variables that appear on the one hand in t_1 or t_2 and on the other hand in c or any of $t_3, \dots, t_n$. The fact that variables that appear in t_1 or t_2 but not in $t_3, \dots, t_n$ or c do not appear in t_{new} ensures that the number of intermediate facts is kept relatively low.

The decomposition of rules calls for an order of the literals, the simplest such order is the one inherited from the MINI-GDL rule. However, it is necessary that the safety property (see Section 2.1) holds after the rules are decomposed. Consequently, literals might need to be reordered so that every variable appearing in a negative literal m appears in a positive literal before m . The programmer who wrote the game in Knowledge Interchange Format (KIF) might have ordered the literals to strive for efficiency or the literals might have been reordered by optimizations at the MINI-GDL stage. In order to minimize interferences with the original ordering, only negative literals are moved.

4.4 Inversion

After the decomposition is performed, the *inversion* transformation takes place. Each predicate p will generate a function in the target language. This function would in turn trigger the functions corresponding to head of rules in the body of which p appeared. The arguments of the target function correspond to the arguments of the predicate in NORMAL-GDL.

The inversion transformation must also take into account the fact that a given predicate can naturally appear in several rule bodies. Such a predicate need still to be translated into a single function in the target language. Therefore, an important step of the inversion transformation is to associate to each function constant f the couples (rule head, remaining rule body) of the rules that can be triggered by f .

4.5 Target language

Once the game has been translated to INVERTED-GDL, it can be processed by the back-end to have a legitimate target language program. Our implementation generates OCAML code, but it is relatively straightforward to extend it to other target languages, provided the appropriate runtime is written.

OCAML (Leroy et al. 1996) is a compiled and strongly typed functional programming language supporting imperative and object oriented styles. Some key features of OCAML simplify the back-end, particularly algebraic data types and pattern matching.

Game	YAP	GaDeLaC	Factor
Breakthrough	1395	340	24%
Connect4	3249	990	30%
Nim1	26066	17500	67%
Sum15	36329	37300	103%
Roshambo2	68252	70900	104%
Bunk_t	17620	22950	130%
Doubletictactoe	17713	23600	133%
Tictactoeserial	8248	11200	135%
Tictactoe	31370	45800	146%

Table 3: Comparison of the number of random playouts performed in 30 seconds by YAP and GaDeLaC based engines.

5 Experimental results

The usual interpretation of GDL is done through an off the shelf PROLOG interpreter such as YAP (Costa et al. 2006). We implemented the proposed compiler using an OCAML back-end. We named it GaDeLaC and performed two sets of experiments described hereafter. The experiments were run on a 2.5 GHz Intel Xeon.

Table 2 shows the time needed in seconds (s) to transform various GDL source files to the corresponding OCAML files and the time needed by the OCAML compiler to compile the resulting files into an executable. We also provide the size of the main files involved in kilo-byte (KB). Namely the original GDL source file with extension `.gdl`, the translated OCAML file with extension `.ml` and the executable file.

As can be seen from Table 2, the proposed compilation scheme is pretty efficient since the whole process takes less than one second for typical games. The compilation scheme proposed in this article scales very well as can be seen from the time needed to compile the most stressing games `Duplicatestatelarge` and `Statespacelarge` which is less than thirty seconds.

We then need to measure the speed of the resulting executable. A simple benchmark to test the speed of a GGP engine is to play a large number of games using a random policy. The second set of experiments (see Table 3) consists in counting the number of random playouts that can be played from the initial state during 30 seconds. Jean Méhat was kind enough to provide us with comparative data from his state of the art player Ary (Méhat and Cazenave 2011) which uses YAP to interpret GDL. The comparative data was obtained on a 2 GHz Intel CPU.

To gain reasonable confidence in the data presented in Table 3, the following care was taken. Each number in the GaDeLaC column is the average of ten runs of 30 seconds each. As can be expected, GaDeLaC is not bug free, however, we recorded the average playout length as well as the frequency of each possible game outcome and compared these statistics to the equivalent ones provided in the Ary data set. The playout statistics of GaDeLaC matched those of Ary for each game in Table 3.

GaDeLaC is only a proof-of-concept implementation but the results presented in Table 3 are very encouraging. A GaDeLaC based engine is significantly faster than a PRO-

LOG based engine on several games but it is also slower on a couple of other games. Unfortunately we do not have any definitive characterization of game rules that would allow to decide whether a resolution-based engine would be quicker than a bottom-up engine.

6 Discussion and future work

The translation in GaDeLaC keeps the first-order nature of the rules, as a consequence the improvement factor is less than what Kevin Waugh could obtain in Waugh (2009), on the other hand using an instantiation pass to preprocess the rules such as suggested by Kissmann and Edelkamp (2010) remains a possibility and is likely to lead to further acceleration.

Several extensions to this work are anticipated. Devising a more adapted runtime data structure would surely allow a considerable speed gain. Partial instantiation of well-chosen predicate would allow more predicates to be considered persistent without compromising scalability. To the best of our knowledge this work is the first first-order forward chaining approach to GGP and GDL rulesheets are tested and optimized with resolution based engines, it could therefore be interesting to see if optimizing the GDL rules of a game towards bottom-up based engines would change much the rulesheet and accordingly the performance. It might also be interesting to test methods to direct the bottom-up evaluation from the deductive databases community, for instance *magic sets* (Kemp, Stuckey, and Srivastava 1991) or *demand transformation* (Tekle and Liu 2010, Section 4) could prove appropriate. Finally, writing back-ends for different languages is envisioned, for instance, generating C code might improve the performance even further and would enhance compatibility with other artificial players.

7 Conclusion

We have presented a bottom-up based Game Description Language compiler. It transforms GDL into rules that have only two conditions. It then uses OCAML to perform forward chaining with these rules. It performs playouts at a speed competitive with Ary for most of the games we tested. This is a promising result since more optimizations are still possible.

Acknowledgements

The authors would like to thank Jean Méhat for contributing comparative data from his competitive player, Peter Kissmann for his insights on the intricacies of the Game Description Language, and Bruno De Fraine for his various advice. The authors would also like to thank the anonymous reviewers for their detailed comments.

References

- [1] Clune, J. 2007. Heuristic evaluation functions for general game playing. In *AAAI*, 1134–1139. AAAI Press.
- [2] Costa, V.; Damas, L.; Reis, R.; and Azevedo, R. 2006. YAP Prolog user’s manual. *Universidade do Porto*.

Game	Size of the .gdl file (KB)	Size of the .ml file (KB)	Size of the object file (KB)	Transformation time GDL → OCAML (s)	Compilation time OCaml → object file (s)
Breakthrough	3.6	48	115	0.024	0.46
Tictactoe	3.2	22	55	0.012	0.27
Duplicatetestsmall	3.7	47	108	0.022	0.43
Duplicatetestmedium	23	337	755	0.170	3.6
Duplicatetestlarge	73	1100	2500	0.980	28
Statespacesmall	2.0	34	76	0.018	0.33
Statespacemedium	13	236	516	0.108	2.3
Statespacelarge	45	825	1700	0.620	16

Table 2: Measuring the compilation times and file sizes.

- [3] Genesereth, M., and Love, N. 2005. General game playing: Overview of the AAAI competition. *AI Magazine* 26:62–72.
- [4] Günther, M.; Schiffel, S.; and Thielscher, M. 2009. Factoring general games. In *Proceedings of the IJCAI-09 Workshop on General Game Playing (GIGA'09)*, 27–34.
- [5] Kemp, D.; Stuckey, P.; and Srivastava, D. 1991. Magic sets and bottom-up evaluation of well-founded models. In *Proceedings of the 1991 Int. Symposium on Logic Programming*, 337–351.
- [6] Kissmann, P., and Edelkamp, S. 2010. Instantiating general games using prolog or dependency graphs. In *KI 2010: Advances in Artificial Intelligence*, 255–262. Springer.
- [7] Leroy, X.; Doligez, D.; Garrigue, J.; Rémy, D.; and Vouillon, J. 1996. The Objective Caml system. *Software and documentation available from <http://pauillac.inria.fr/ocaml>*.
- [8] Liu, Y. A., and Stoller, S. D. 2009. From datalog rules to efficient programs with time and space guarantees. *ACM Trans. Program. Lang. Syst.* 31(6):1–38.
- [9] Love, N. C.; Hinrichs, T. L.; and Genesereth, M. R. 2006. General Game Playing: Game Description Language specification. Technical report, LG-2006-01, Stanford Logic Group.
- [10] Méhat, J., and Cazenave, T. 2010. Combining UCT and nested Monte-Carlo search for single-player general game playing. *IEEE Trans. on Comput. Intell. and AI in Games* 2(4):271–277.
- [11] Méhat, J., and Cazenave, T. 2011. A Parallel General Game Player. *KI-Künstliche Intelligenz* 25(1):43–47.
- [12] Tekle, K., and Liu, Y. 2010. Precise complexity analysis for efficient datalog queries. In *Proceedings of the 12th international ACM SIGPLAN symposium on Principles and practice of declarative programming*, 35–44. ACM.
- [13] Thielscher, M., and Voigt, S. 2010. A temporal proof system for general game playing. In *AAAI*.
- [14] Thielscher, M. 2009. Answer set programming for single-player games in general game playing. In *ICLP*, 327–341. Springer.
- [15] Waugh, K. 2009. Faster state manipulation in general games using generated code. In *Proceedings of the IJCAI-09 Workshop on General Game Playing (GIGA'09)*.

Extending the General Game Playing Framework to Other Languages

Xinxin Sheng, David Thuent

North Carolina State University
xsheng@ncsu.edu, thuent@ncsu.edu

Abstract

General Game Playing (GGP) research aims at building automated intelligent computer agents that accept declarative descriptions of arbitrary games at run time and are able to use such descriptions to play effectively without human intervention. This paper extends the Game Description Language (GDL) to more easily describe complicated games, including social and financial games with diverse goals. The GDL extension allows adding functions, defined in other programming languages such as Java, C++, Lisp, to the general game playing architecture. We used the three-player, simultaneous financial game Farmer, a standard GGP game, as an example to illustrate the need for and the benefits of the GDL extension. We also show the extension of the GDL is critical for adding coalition games to the general game playing area.

Introduction

General Game Playing (GGP) research aims at building automated intelligent computer agents that accept declarative descriptions of arbitrary games at run time, and are able to use such descriptions to play effectively without human intervention. GGP as a research topic was started at Stanford University in 2005. The GGP competition has been held annually at the AAAI conference since then. Many successful GGP players have emerged (Clune 2007; Kuhlmann and Stone 2007; Finnsson and Bjornsson 2008; Schiffel and Thielscher 2007; Sheng and Thuent 2011a; Mehat and Cazenave 2011; Möller *et al.* 2011).

GGP is designed to include a wide variety of games, including one-player games, two-player games, multiple-player games, turn-taking or simultaneous games, zero-sum or non-zero-sum games, etc. The players can have conflicting interests or compete for scarce resources. The GGP design should allow coalition games as well but they have not been addressed as such and are not included. There are about 40 kinds of games in the GGP database with more than 200 variations that are created by slight modifications. Many of these games have been played in the AAAI general game playing competitions. For these competitions no human intervention is allowed, no matter

whether the agent plays a new game that the world has never seen before, or an old game that has already been solved. The agents have to automatically adapt not only to play the games but obviously more so, to play well.

The General Game Playing framework is a multi-agent system with one game server agent coordinating several game playing agents. The player agents communicate through the game manager but not with each other. The game manager starts the automated game play by sending out the game information and setting up the playing time. The game manager provides the player agents with the initial state of the game, the set of game rules, and the agent's own role in the game. With such information, the agents use knowledge reasoning to calculate what are the legal actions to take.

The GGP games are written in Game Description Language (GDL) (Love *et al.* 2005), which is written in KIF (Genesereth and Fikes 1992), a subset of First Order Logic (FOL). GDL can be understood as a specification language for a large class of multi-agent environments, working as formal laws to guide the agents to understand the game rules and their own roles so that they can participate in the games (Schiffel and Thielscher 2009). All agents participating in the game accept game rules as provided and take proper actions to win. The game manager validates and broadcasts all actions submitted by different player agents and thereby moves the game forward to the next stage.

A game played in the GGP framework can be understood as a finite state transition system with one initial state and at least one terminal state. The game rules are the transition functions between the states. A game starts from the initial state and transits from one state to the next precisely in response to the actions taken.

GDL grammar and syntax can be used to define constants, variables, functions, operators, game states and game rules. Constants can be relation constants such as *adjacent*, or object constants, such as *King* (in chess). The types of constants are determined by the context. Variables begin with a ? and the values are determined in the logic

resolution process. All operators are in prefix notation, including the negation. The state of the game is described with a group of propositions that are true in that state. A game rule is a prefix implication. The head of the rule is a relation constant with a number of arguments. The body of the rule contains zero or more literals, or negative literals. Recursion is allowed, but with limitations to avoid unbounded growth.

The Game Description Language is quite powerful and has defined numerous different games. However, as a language written in the First Order Logic, GDL suffers the limitation of the FOL as well. This paper explains the limitation of the current framework of the GDL in Section 2. The extension of the GDL is proposed in Section 3. And the application usage of the extension is illustrated in Section 4.

Limitations of the GDL

Game playing is limited by time and space resources. Most games are not exhaustively searchable. The agents need to allocate the limited time to logic reasoning, game tree search, the action payoff evaluation, machine learning, and game states transition, etc. Logic reasoning participates in all these operations. Small performance improvements in the reasoning process can be greatly amplified to expedite searches and learning and can therefore generate huge advancements in the agent's performance. We have proposed using hash tables to improve the reasoning efficiency in previous research (Sheng and Thunte 2010a).

In addition to the logic resolution data structure, we found out how a game is written affects the reasoning performance as well. For example, to determine if there exists a line in Tic-Tac-Toe, the function can be written either as "find out all lines on the board, and check if one is owned by myself", or as "find out all my pieces, and check if any three formed a line". The computational complexity is different in these two expressions. In big games, some expressions can become so computationally demanding that the desired function can not be reasonably defined.

The Financial Game Farmer

The Farmer game is a three-player simultaneous non-zero-sum financial game especially written for GGP. Traditional computer game research covers mainly two-player turn-taking games, but the Farmer game sets up tasks for three players (*Alice*, *Barney* and *Charlie*) to make as much money as possible in a certain number of iterations (there are 10, 20, 40 iterations versions in GDL of Farmer game in which the game terminates at the designated iteration and rewards are calculated). Each player tries to maximize his own net worth. We found this game very interesting

and there are several publications discussing it (Sheng and Thunte 2011a; Sheng and Thunte, 2011b).

The three players in Farmer each start with 25 gold pieces. The players begin with no commodities on hand. The initial market price for wheat is 4 gold pieces, flour 10, cotton 7 and cloth 14 gold pieces. After every step of the game, the market price of each commodity automatically increases by 1 even if nothing is bought or sold. In addition to the automatic increase, the product price of that commodity decreases by 2 in the next iteration for every distinct selling action (maximum decrease of 6 in one step for the three player game) of that commodity, and increases by 2 for every buying action of that commodity.

The Farmer game contains a simplified skeleton of the manufacturer financial activities, such as investing in infrastructure (buy farm or build factory), buying raw material from the market, selling the product to the market, and keeping the inventory down, etc. It also contains the elements of the stock market, such as an inflation indicator (price increases by 1 in every step.), price affected by market activities (price goes down by 2 for every selling action, and up by 2 for every buying action), the player with the most money wins, etc. The Farmer game opens the possibility of using the general game playing techniques to explore and perhaps even solve skeletal real world financial or social problems.

Burden of the Natural Numbers

In a financial game with items to buy and sell, arithmetic addition and subtraction are used to calculate the gold amount. Natural numbers are not defined as part of the GDL. Instead, the arithmetic operations are defined as part of the game rules.

```

1. (number 0) //define unary relation number
2. (number 1)
3. ...
4. (number 300)
5.
6. (succ 0 1) //define the sequential relation
7. (succ 1 2)
8. ...
9. (succ 299 300)
10.
11. (<= (sum ?x 0 ?x) //define function addition
12.      (number ?x))
13. (<= (sum ?x ?y ?z) //define recursive function
14.      addition
15.      (succ ?x ?m)
16.      (succ ?n ?y)
17.      (sum ?m ?n ?z))

```

Figure 1. Arithmetic addition defined with GDL in the Farmer game.

Figure 1 is part of the Farmer game that defines the addition operations of natural numbers. Bold indicates the first definition of a function.

The Farmer game defines the unary relation *number*, and the order sequence of the numbers. The sum function in line 11 defines $\{x+0 = x\}$, and line 13 is the recursion version of the sum function. In this piece of code, only natural numbers between 0 and 300 are defined. The addition calculation is extremely tedious. To calculate $(1+299)$, for example, the game rules as defined in line 13 need to calculate $(2+298)$ in line 16, then $(3+297)$ etc. After performing 300 layers of recursion, the result $\{(300+0) = 300\}$ can be calculated from the base definition in line 11, and the stack is popped until $(1+299)$ value returns. In this schema, it is not even possible to calculate $(2+300)$. Even though both numbers are defined in the game, the sum is not defined. Suppose we can define such a big set of numbers that we are completely confident that it is enough for the game to use. In this case, let the numbers defined be n , the time complexity for the number or succ function are $O(n)$, so the recursion addition calculation as defined in line 13 has the time complexity of $O(n^2)$. A simple arithmetic addition operation is expected to be conducted in $O(1)$ time. The current GDL structure is obviously not satisfactory in terms of computational requirements.

Confusion of the Comparison

Although natural numbers are not defined as a GDL component, they are used in the goal definition loosely in many games. For example, in Tic-Tac-Toe, when a player forms a line, it is rewarded with 100 points with the rule $(\leq (\text{goal ?player } 100) (\text{line ?player}))$, and is rewarded with 0 points when the opponent forms a line. In both situations, the numbers were used without being defined, how can the agent realize 100 is more desirable than 0?

In logic operations, which term is used does not affect the logic relations as long as the changes are consistent. Anything that is not defined as a keyword in the GDL is substitutable. In fact, in the GGP competitions, to avoid human cheating, all the competition games were scrambled into random, meaningless letter combinations. The agent, operating correctly, can perform the same logic inference on the scrambled version. As long as the logic relations are the same, the scrambled version of the game will be treated the same. For example, the Tic-Tac-Toe state description (cell 1 1 blank), which means a ternary relation cell that relates a row number, a column number, and a content designator, is logically equivalent to the description (position first first empty), or scrambled version (tsmpzz ryy ryy ffgh), as long as the terms are replaced consistently in all other occurrences of the game rules (Sheng and Thunte 2010a).

Since substitution is not supposed to change the logic relations, the GDL code $(\leq (\text{goal ?player } 100) (\text{line ?player}))$ can be substituted to $(\leq (\text{goal ?player } sss) (\text{line ?player}))$ as long as all the definitions and usages of the constant 100 are substituted with *sss* universally. In many games, 100 (or 0) is used without definition. Although the numerical goal rewards have never be scrambled in the past competitions, being not defined as keywords but used as regular logic symbols, they are subjected to the scramble operation just like any other symbol. Now the agent is facing a logic resolution loophole: is the goal *sss* more desirable than the goal *ttt*?

In some situations where the players are given partial credit, such as moving four pieces into the designated place and getting 25 points for each piece, it is not possible to choose one goal over another without knowing the values of the goals.

To define the goals more rigorously, the game need to define unary relation *number* as in Figure 1. Then define a *greater* function for the agent to compare the value of 100 and 0. The agent can order all the goal definitions and determine which one is the most desirable. Only in this way, can the agent calculate the partial credit for completing partial tasks too.

Challenge of Infinity

The finite set of natural numbers is hard enough for the GDL to handle. When the content of the game element involves an infinite set, the existing structure faces new challenges.

In the Farmer game, the market price of each commodity automatically increases with the inflation factor every iteration as time goes by. Suppose the inflation factor is 10% of the current price. Then the wheat price with no transaction influence is 4 gold in the first step, 4.4 in the second, 4.84 in the third, 5.324 in forth, and so on.

In the GDL, it is still possible to define the real number set by defining the computational elements, such as the carry flag, the borrow flag, the negative sign, etc. But for the real number infinity set, there does not exist a reasonable way to describe such a scenario.

We can go on with the limitations of GDL and how to use it better to define an infinity set by proposing a overflow threshold just like all computer systems do. But the discussion of how to define more powerful scenarios with the GDL misses the point. The GDL structure is for accommodating heterogeneous games in one multi-agent system for the player agents to play. Re-writing of these computer games with GDL is not only cumbersome but also not necessary.

Extension of the GDL

GDL describes the basic game elements such as game initialization (keyword *init*), state transition (keyword *next*), legal actions (keyword *legal*), game roles (keyword *role*), game goals (keyword *goal*), game termination (keyword *terminal*), etc. Researchers have written about forty games with more than 200 variations with the GDL.

The GDL has never meant to be a closed system. There have been efforts to extend the GDL to describe new groups of games. New keywords *sees* and *random* were introduced in order to add incomplete information games (Thielscher 2011). New keywords *message*, *receive*, *send*, *time* were added to describe market trading scenarios (Thielscher and Zhang, 2009). Adding new keywords and a new group of games to the GGP family certainly enriches this area. However, the process can be generalized in a manner that makes new game definitions much easier, simpler, and clearer.

We propose adding the keywords *include* and *use* into the GDL. Just like importing packages in the Java language, or including the header files in C++, the *include* keyword in GDL serves as an interface allowing area specific definitions and functions to be shared and quoted without affecting the generality of the GDL. The *include* keyword serves as the information facade that bridges the game logic representations described with GDL and the area specific contents. The keyword *use* is to call external functions with parameters.

The major contribution of the extension is that an existing game does not have to be completely re-written into GDL for the general game player to play. A game does not have to be written in one computer language either. Certain game information (see section 5 for more details) can be defined and calculated in the user's favorite programming languages and passed back and forth through a command line interface.

Implementation of the GDL Extension

We now use the piece of code in Figure 1 of the Farmer game again as an example for how arithmetic can be introduced with the include interface. Suppose the market price for wheat is 4 now and the price will rise 10% every iteration. Alice wants to know how much the price will be six iterations later. The rules in the Farmer game need three arithmetic functions: add, exponential, and multiply. Suppose the three functions are written in Java, C++, and Lisp, respectively. The pseudo-code of the GDL extension can be written as:

```

1. ----- Code written in GDL -----
2. include mathFromJava http://server/app/math1.jar
3. include mathFromCpp http://server/app/math2.exe
4. include mathFromLisp http://server/app/math3.lsp
5.
6. use mathFromJava (java research.ggp.MathApp)
7. mathFromJava (add ?p1 ?p2 ?p3)
8. //call Java jar through command line parameters
9.
10. use mathFromCpp (math2.exe)
11. mathFromCpp (exponential ?p1 ?p2 ?p3)
12. //call C++ executable through command line
13.
14. use mathFromLisp (load math3.lsp)
15. mathFromLisp (multiply ?p1 ?p2 ?p3)
16. //call Lisp executable through command line
17.
18. <=(inflation ?oldPrice ?rate ?step ?newPrice)
19.   (add ?rate 1 ?inflation) // inflation rate is 1+10%
20.   (exponential ?inflation ?step ?totalRate)
21.   (multiply ?oldPrice ?totalRate ?newPrice)
22.
23. //-----Code written in GGP agent for parsing-----
24.
25. Concatenate use sentence into command line with
26.   known parameters assignments, e.g. concat line 6
27.   and line 7 into (java research.ggp.MathApp add
28.   10% 1 ?p3)
29.
30. Process result list as returned by standard output
31.
32. //-----Code written in Java-----
33. -
34. package research.ggp //java code
35. class MathApp{
36. public static void main(String[] args) {
37.   switch(args[0]) { //function selection
38.     case "add": add(args); break;
39.     case "function2": function2(args); break;
40.     case "function3": function3(args); break;
41.   } //end of main
42.
43. static void double[] transform(String[] args){
44.   //parse the arguments into an array with proper
45.   type.
46.   for(int i=0; i<args.length;i++){
47.     result[index] = doubleValueOf(args[index]);
48.   }
49.   return result;
50. }
51.
52. static void function add(String[] args) {
53.   double ps[] = transform(args);
54.   out.println("%s %s %s %s", ps[0], ps[1], ps[2],
55.   ps[1]+ps[2]); //send result to standard output,
56.   where the GGP agent collects results. (add 3 5
57.   ?p3) will return (add 3 5 8)
58. } //end of class

```

```

48.
49. //-----Code written in C++ -----
50. //omit function selection and parameter list parse
51. void exponential(char *argv[]) {
52.     double ps[] = transform(char *argv[]);
53.     cout <<ps[0]<<" "<<ps[1]<<" "<<ps[2]<<" "
        ps[1]^ps[2] << endl; }
54.
55. //-----Code written in Lisp-----
56. //omit function selection and parameter list parse
57. (defun multiply(args)
58.   (setq result (* (second args) (third args)))
59.   (remove '?p3 args)
60.   (append args (list result)))

```

Figure 2. Pseudo code for extending GDL to call external resources.

The pseudo code shows a straightforward example of how to include and use the external resources in different computer languages. The GDL code (line 1-21) is available to all players. The executables of different languages are available at some public URL for all players too. Depending on the language that the GGP agents use, the agent needs to add additional keyword parsing ability to parse include and use, just as it parses other keywords. The GDL code needs to be parsed into a command line to call the public executables with parameters (line 24). The parsing process assigns values to known parameters, and concatenates them into a command line to call the target executables. The parsing schema is independent of the language in which the GGP agent is written, or the language in which the executable is written. The agent collects standard output as the results returned (line 25). When calling, not all parameters need to be instantiated (e.g. line 24). The undecided parameters' value will be filled by the external calculation. When there are multiple assignments of the parameters, the external functions can be called multiple times, each time with one assignment list.

Instead of defining the numbers, the number sequence, and number addition as in Figure 1, the GDL add function is not defined in the game rules. Instead, the GDL uses the *include* keyword to refer to a foreign resource and pass three parameters (two instantiated) with the *use* keyword. The same calculation that took $O(n^2)$ in Figure 1 can be done in $O(1)$ time now.

The foreign resource can be a shared file, a URL, or a database library. It is accessible by all players. The foreign data structure can be a list of facts, functions, or calculations. We have tested Java, C++ and Lisp. The same experiments can easily be done in other programming languages as well. The area of the specific knowledge can be game related such as resource utilization or more generally, eCommerce or a social scenario.

Note that the GGP agent does not need a corresponding compiler to invoke the executables. Only the running environment for the executables needs to be set up, just like .exe on Windows, or bash on Linux. Additional steps need to be taken for cross platform implementation such as data transfer from Windows to Linux, or to Mac, etc. In the future, the executable invoking process can be generalized into a more universal solution such as using web services or cloud computing.

To summarize, as long as the parameters can be passed back and forth through the information façade, the game functions can perform the query to complete the logic resolution. Where the data are stored and how the data are organized are not the GDL's concern any more.

Impact of the GDL Extension

The modifications and extensions of the GDL broaden the GGP area and raise the set of potential applications to a new level. The GDL is sufficiently expressive to model many relations. However, the fact that GDL is capable of describing many real world scenarios does not mean it can be done in a concise or effective manner and it certainly does not help or force the game designer to do so in a reasonable manner. Section 2.2 is an example of a tedious definition of a small group of natural numbers and addition operations. When it comes to the game goal determination or infinite real number calculation, the existing GDL structure is hindering rather than enabling new games be brought to the general game playing field. Since there exist arithmetic libraries described in other computer programming languages, redefining everything again in GDL not only adds little values to the GGP system, but also is discouraging people from turning to the GGP system for problem solving. The *include* and *use* keyword would introduce the arithmetic library and thereby make additional games possible in the GGP. Coalition games are often useful to better simulate and solve real world problems in financial, social, or industrial scenarios. With the new keywords *include* and *use* to open the interface with other languages, existing information or data from other domains can be easily hooked to the GGP engine.

GDL Extension Application

The external information provided via the *include* and *use* keywords is used only for the knowledge resolution and reasoning process within the game description. Different game agents are given the same game to play; therefore the same external functions to use. The logic resolution performance will be improved in a way that the external library changed "how the game is written" and will benefit all players. But the decision making and agents'

performance are still completely determined by the game playing algorithms.

At the beginning of the section 3, we mentioned that the GDL extension would benefit market trading economic and incomplete information games. Coalition games are another important area of game playing research that would benefit from the GDL extension.

Coalition Games

A Coalition Game with transferable utility is defined as (N, v) , where N is finite set of players; $v : 2^N \rightarrow \mathbb{R}$ associates with each subset $S \subseteq N$ a real-valued payoff $v(S)$ (Leyton-Brown and Shoham 2008). The inherent features of payoff calculations in coalition games essentially forced the GDL extension.

There have been multi-player games that require playing in groups in the GGP area. The six-player 3D Tic-Tac-Toe (see Tic-Tac-Toe 3D 6-player) requires players to play in groups of three. Each player picks the coordinate value in one dimension. The three dimensions picked by the three players determine the final position of the piece. The four player Othello (see Othello 4-player) game is similar. Four players are divided in groups of two, with each player picking one coordinate. Unlike these two games that are played in groups, coalition games require that each agent can play the games both by itself and in groups. There have not been any published coalition games in GGP.

The extension of the GDL makes describing coalition games possible. Using the extended GDL, we can explore more realistic economic and societal problems. The focus of this section is not a study of the coalition games as such but an example of the use of the GDL extension in coalition games. The results show our extensions generated reasonable results for an eCommerce coalition game.

Coalition Farmer Game

The current Farmer game is defined as a win-or-lose game. The player that ended up with the most money at the end of the game wins with 100 points awarded. If the player owns the same amount of money as another player, or less money than another player, he loses with zero points awarded.

However, the nature of the Farmer game is indeed an eCommerce game with financial achievements. In a more realistic scenario, when three players started with the same amount of money and face the same market rules, they do not have to become the richest among the three to be rewarded. The amount of money the player owns is the real reward. How much the player earned during the process matters. The player would be happier with 100 gold pieces and finishing in second place, than with 60 gold pieces and finishing in the first place. The metrics of winning is different in reality.

In addition to the amount of gold pieces earned, the three players can be more cooperative than competitive. In the open market setting with a fixed inflation number, buying a small amount in each step, holding till the end of the game, then selling together at the last step can be an easy solution for all three players to get rich. This is because pushing the price high every step with a purchase action benefits all players. In a coalition scenario like this, the goal is not beating the other players anymore, but to maximize the net worth of the entire coalition, without severely punishing any individual players of course.

The Farmer game can be rewritten into a coalition game with the GDL extension, where the performance metrics can be the total money earned by the three players together. The current inflation factor in the farmer game is set to 1 gold every iteration for simplicity. If the step-wise inflation is re-defined as 10% of the current price, there exist no reasonable way in the current GDL framework to describe such game factors without forcing the proposed extension of importing external libraries. The extension is not only convenient, but also necessary to define practical, realistic, and applicable coalition games in GGP.

With the extension of the GDL, we added the gold pieces from each of the players to get the total gold pieces earned by the three players for a coalition game. When each of the three players are trying to make their own best net worth (individually and collectively), the total value of the game (number of gold pieces as presented on the X-axis in Figure 3) increases with more cooperation.

Four experimental scenarios of the Farmer game with 1000 simulations each were analyzed. In Scenario 1, an intelligent GGP agent Alice plays against random dummy players Barney and Charlie. Alice is using the machine learning algorithms as proposed in previous research (Sheng and Thuente 2011a). Alice earns almost twice as much as Barney or Charlie which shows the value of the intelligent agent. In Scenario 2, both Alice and Barney are using intelligent decision making for actions while Charlie remains a random player. Here both Alice and Barney earn twice as much as Charlie and Alice increases her earning over her earnings in Scenario 1. Scenario 3 has all three players using intelligent agents to make strategic decisions to maximize their individual income. In Scenario 3 both Alice and Barney improve their earnings over Scenarios 1 and 2 where they were the only one or two, respectively, trying to optimize their earnings. Scenario 4 has Alice and Barney in a coalition playing against the intelligent player Charlie. Compared to the Scenario 3, Alice and Barney both focus on maximizing the sum of their earnings. It turned out by working in coalition, the individual player's earning, the coalition earning, and the three players' total earning all increased.

The obvious observation is that when the individual players get better, the total income of the group increases

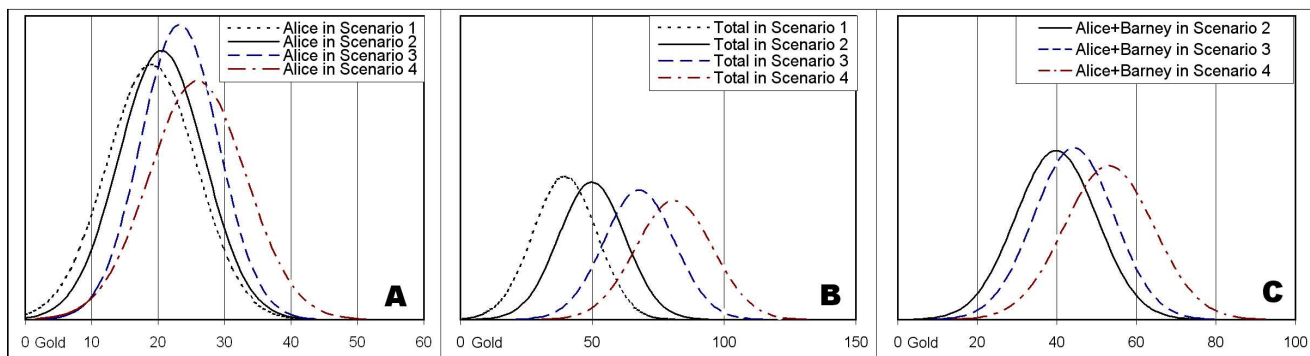


Figure 3. The normal distribution of the amount of the money the player made in different scenarios, based on 1000 matches each. (A) is gold pieces earned by player Alice; (B) is the total gold pieces of all three players; (C) is the collective results of player Alice and Barney.

(Figure 3B). In addition, if we single out Alice's income in the three scenarios, it is notable that as the other players are getting more and more capable, the income of Alice increases instead of decreases (Figure 3A). Such observations verify that it is possible to pursue the player's own interest without hurting other players in the Farmer game and therefore the Farmer game is more cooperative than competitive. We showed that, for this game, maximizing individual agent's payoff actually improves the total payoff.

To make it a more truly coalition game, we rewrite the Farmer so that “winning” requires the collective net value must be at least some number of gold pieces. We have learned from previously playing of the game in the Scenario 2 with the intelligent Alice and Barney versus random Charlie that the average collective income of Alice and Barney is 40 gold pieces (Figure 3C). So if the total of Alice and Barney is above the average 40, it is considered a coalition win for both players. In the new coalition game design, when we apply the same feature identification algorithm as in the previous research (Sheng and Thuente 2010b), a different set of features and weights were identified to guide the game tree search. For example, the action group sell inventory together is identified as important, because the price drops after each selling action so the inventory is devalued by another player's selling action. When Alice and Barney are cooperating in Scenario 4, the coalition income exceeded the addition of the individual income in Scenario 3.

To summarize, the GDL extension enables complex payoff distributions and therefore allows the new coalition version of the Farmer game, in which each of the three or more players can have their own strategy or a group strategy. The players can play individually to determine how much they can win, or in coalition with the definition of winning for the game being the collective winnings of any given coalition. Scenario 4 in Figure 3C gives the results of the coalition of Alice and Barney where they are actively

cooperating to maximize the sum of their winnings. Scenario 4 provides the highest winnings for any of the examined renditions of the 10 iteration Farmer game. The collective achievements do not necessarily conflict with individual payoffs. Depending on the games rules, there exist situations where maximizing collective payoffs would strengthen individual payoffs in turn and vice versa.

Discussion

Since the emergence of the GGP area, the focus of the research has been on how to build a good agent. Yet this paper discusses how to incorporate more GGP games. The GDL extension opens the door for extending the GGP into new, realistic, coalition eCommerce applications. The coalition Farmer game is an example of using the modifications and extensions of the GDL. The coalition games broaden the GGP research area as we study how the dynamics change from competitive to coalition games. Fresh insights will be gained in both the specific game and GGP.

In one GGP game, the logic knowledge can be divided into two kinds. There is static information that does not change from the beginning to the end of the game (e.g. the board adjacency) and dynamic information that changes from step to step. With the GDL extension, theoretically, only game elements such as the game state, player's role, goals, state transition, legal actions, etc. need to be defined in GDL and all other computational intensive functions can be imported from or defined in user's favorite languages. However, the time spent passing parameters back and forth is considered game time cost as well. Most static information can be defined outside with the expectation of improving performance since the parameters are only passed once. The arithmetic calculations that have extremely high cost within GDL should be defined outside GDL. The minor or moderate calculation that demand frequent changes as the game state changes need to

handled with caution. It is up to the game writers' judgment to determine if keeping the function definition in or outside the GDL is more efficient.

It is possible that certain GGP agents in the field today will be affected by the extension and lose their advantage in the competition. In fact, the GGP competition does not have to implement the proposed extension, as long as the researchers do not mind writing every new game with GDL for the competition. But the GDL extension proposed in this paper allows people to contribute to the GGP area without being experts in GDL. Therefore, it would give agents access to more games outside of GDL and hence attract more people into the GGP area in the long run. As GGP agents become more powerful and with this GDL extension, GGP agents may contribute to solutions in diverse domains.

The purpose of extending GDL goes beyond number calculation, even beyond introducing coalition games. It is the first step of setting up a system in which an existing game written in other programming languages does not need to be completely rewritten into the GDL to let the GGP agent play. We believe as long as the basic game elements such as initial state, player's role, legality, state transition, and goals are re-defined in GDL, a new game can be added to the GGP family without importing the data relation, computation and storage. This is one direction of our future research.

Acknowledgement

The authors gratefully acknowledge the assistance of Arpan Chakraborty concerning questions on the Lisp interface in section 3.1.

References

- Clune, J. 2007. Heuristic Evaluation Functions for General Game Playing. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence*, 1134-1139, Vancouver, Canada.
- Finsson, H., and Bjornsson, Y. 2008. Simulation-Based Approach to General Game Playing. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence*, 259-264, Chicago.
- Genesereth, M., and Fikes, R. 1992. *Knowledge Interchange Format Version 3.0 Reference Manual*. Logic Group Report Logic-92-1, Computer Science Department, Stanford University, Stanford, CA.
- Kuhlmann G., and Stone P. 2007. Graph-Based Domain Mapping for Transfer Learning in General Games. In *Proceedings of the Eighteenth European Conference on Machine Learning*, Warsaw, Poland.
- Leyton-Brown, K., and Shoham, Y. 2008. *Essentials of Game Theory*. Morgan and Claypool Publishers.
- Love, N.; Hinrichs, T.; Haley, D.; Schkufza, E.; and Genesereth, M. 2005. *General Game Playing: Game Description Language*

Specification. Technical report, Computer Science Department, Stanford University, Stanford, CA.

Mehat, J., and Cazenave, T. 2011. A Parallel General Game Player. *KI - Künstliche Intelligenz*, 25(1): 43-47, Springer.

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

Othello 4-player. http://euklid.inf.tu-dresden.de:8180/ggpserver/public/view_game.jsp?name=othello-fourway

Schiffel, S., and Thielscher, M. 2009. A Multiagent Semantics for the Game Description Language. In *Proceedings of the International Conference on Agents and Artificial Intelligence*, Porto, Portugal.

Schiffel, S., and Thielscher, M. 2007. Fluxplayer: A Successful General Game Player. In *Proceedings of the 22nd AAAI Conference on Artificial Intelligence*, 1191-96, Vancouver, Canada.

Sheng, X., and Thuenste, D. 2011a. Decision Tree Learning in General Game Playing. In *Proceedings of Artificial Intelligence and Applications*, 192-199, Innsbruck, Austria.

Sheng, X., and Thuenste, D. 2011b. Using Decision Trees for State Evaluation in General Game Playing. *KI - Künstliche Intelligenz*, 25(1): 53-56, Springer.

Sheng, X., and Thuenste, D. 2010a. Using Hash Tables to Expedite Knowledge Reasoning in the General Game Playing Agent. In *Proceedings of International Conference on Advanced Topics in Artificial Intelligence*, 91-96, Thailand.

Sheng, X., and Thuenste, D. 2010b. Predicative Sub-goal Analysis in a General Game Playing Agent. In *Proceedings of International Conference on Web Intelligence and Intelligent Agent Technology*, 423-427, Toronto, Canada.

Thielscher, M. 2011. GDL II. *KI - Künstliche Intelligenz*, 25(1): 63-66, Springer.

Thielscher, M., and Zhang, D. 2009. From General Game Descriptions to a Market Specification Language for General Trading Agents. *Workshop on Trading Agent Design and Analysis*.

Tic-Tac-Toe 3D 6-player. http://euklid.inf.tu-dresden.de:8180/ggpserver/public/view_game.jsp?name=tictactoe_3d_6player

First-Order Logic with Counting for General Game Playing

Łukasz Kaiser
CNRS & LIAFA
Paris

Łukasz Stafiniak
Institute of Computer Science
University of Wrocław

Abstract

General Game Players (GGPs) are programs which can play an arbitrary game given only its rules and the Game Description Language (GDL) is a variant of Datalog used in GGP competitions to specify the rules. GDL inherits from Datalog the use of Horn clauses as rules and recursion, but it too requires stratification and does not allow to use quantifiers. We present an alternative formalism for game descriptions which is based on first-order logic (FO). States of the game are represented by relational structures, legal moves by structure rewriting rules guarded by FO formulas, and the goals of the players by formulas which extend FO with counting. The advantage of our formalism comes from more explicit state representation and from the use of quantifiers in formulas. We show how to exploit existential quantification in players' goals to generate heuristics for evaluating positions in the game. The derived heuristics are good enough for a basic alpha-beta agent to win against state of the art GGP.

Introduction

The goal of General Game Playing (GGP) is to construct a program able to play an arbitrary game on an expert level without additional information about the game. Currently, the focus in GGP is put on playing non-stochastic games with perfect information, such as Chess or Checkers, while extensions to games with imperfect information and stochastic behavior, such as Poker, have been developed only recently (Thielscher 2010). Even for perfect-information zero-sum games, the task of a general game player is a formidable one, as evidenced by the introduction of various new techniques in each annual AAAI GGP Competition.

During the GGP competition, games are specified in the Game Description Language (GDL), which is a variant of Datalog, cf. (Genesereth and Love 2005). A successful GGP agent must reason about the rules of the game and extract from them game-specific knowledge, such as heuristic functions used to evaluate positions during the game. To facilitate the creation of good general players, GDL was designed as a high-level, declarative language. Still, in recent years (see GGP Competition results 07-10¹) the players which only derive evaluation functions based on reasoning, cf. (Clune 2008), have often lost against Monte-Carlo based players, which rely far less on symbolic deduction and more

on optimized tree-searching with Monte-Carlo simulations for evaluating positions (Finnsson and Björnsson 2008).

In this paper, we present another formalism for game description, which is even higher-level and more declarative than GDL, and we exploit it to beat a state of the art GGP player using only minimax search with logically derived heuristics for position evaluation. The introduced description language also allows to specify games in a more compact and, in our opinion, more natural way than GDL. The state of a game in our formalism is represented by a relational structure and legal moves are given by structure rewriting rules guarded by formulas of first-order logic (FO). The payoffs the players receive at the end are described by counting terms, which extend FO in a way similar to the one presented in (Grädel and Gurevich 1998). These terms can be manipulated easier than GDL descriptions of payoffs.

From the counting terms in the payoffs and the constraints of the moves we derive heuristics for evaluating game positions. These heuristics are good enough for a basic game player, Toss, which only performs a minimax search with alpha-beta pruning, to beat a state of the art GGP agent, Fluxplayer. Fluxplayer has ranked among the top 5 in each GGP competition in the last 5 years and was the only one of the top players we could access online for testing. While Toss does not win in all the games we tested, it generally plays on par with Fluxplayer in games not well-suited for simple minimax (e.g. in Connect5) and decisively outperforms Fluxplayer in other games (e.g. in Breakthrough).

Relational Structure Rewriting

The state of the game is represented in our formalism by a finite relational structure, i.e. a labeled directed hypergraph. A relational structure $\mathfrak{A} = (A, R_1, \dots, R_k)$ is composed of a universe A (denoted by the same letter, no fraktur) and a number of relations. We write r_i for the arity of R_i , so $R_i \subseteq A^{r_i}$. The *signature* of $\mathfrak{A}$ is the set of symbols $\{R_1, \dots, R_k\}$.

The moves of the players are described by *structure rewriting rules*, a generalized form of term and graph rewriting. Structure rewriting has been introduced in (Rajlich 1973), for games in (Kaiser 2009), and is most recognized in graph rewriting and software engineering communities, where it is regarded as easy to understand and verify.

In our setting, a rule $\mathcal{L} \rightarrow_s \mathfrak{R}$ consists of two finite relational structures, $\mathcal{L}$ and $\mathfrak{R}$, over the same signature, and of

¹cadi.a.ru.is/wiki/public:cadiaplayer:main

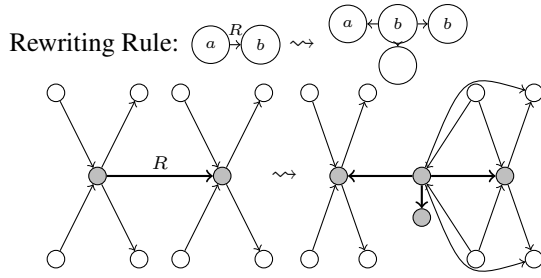


Figure 1: Rewriting rule and its application to a structure.

a partial function $s : \mathfrak{R} \rightarrow \mathfrak{L}$ specifying which elements of $\mathfrak{L}$ will be substituted by which elements of $\mathfrak{R}$. With each rule, we will also associate a set of relations symbols τ_e to be embedded exactly, as described below.

Definition 1. Let $\mathfrak{A}$ and $\mathfrak{B}$ be two relational structures over the same signature. A function $f : \mathfrak{A} \hookrightarrow \mathfrak{B}$ is a τ_e -embedding if f is injective, for each $R_i \in \tau_e$ it holds that

$$(a_1, \dots, a_{r_i}) \in R_i^{\mathfrak{A}} \Leftrightarrow (f(a_1), \dots, f(a_{r_i})) \in R_i^{\mathfrak{B}},$$

and for $R_j \notin \tau_e$ it holds that

$$(a_1, \dots, a_{r_j}) \in R_j^{\mathfrak{A}} \Rightarrow (f(a_1), \dots, f(a_{r_j})) \in R_j^{\mathfrak{B}}.$$

A τ_e -match of the rule $\mathfrak{L} \rightarrow_s \mathfrak{R}$ in another structure $\mathfrak{A}$ is a τ_e -embedding $\sigma : \mathfrak{L} \hookrightarrow \mathfrak{A}$.

Definition 2. The result of an application of $\mathfrak{L} \rightarrow_s \mathfrak{R}$ to $\mathfrak{A}$ on the match σ , denoted $\mathfrak{A}[\mathfrak{L} \rightarrow_s \mathfrak{R}/\sigma]$, is a relational structure $\mathfrak{B}$ with universe $(A \setminus \sigma(L)) \cup R$, and relations given as follows. A tuple $(b_1, \dots, b_{r_i})$ is in the new relation $R_i^{\mathfrak{B}}$ if and only if either it is in the relation in $\mathfrak{A}$ already, $(b_1, \dots, b_{r_i}) \in R_i^{\mathfrak{A}}$, or there exists a tuple in the previous structure, $(a_1, \dots, a_{r_i}) \in R_i^{\mathfrak{A}}$, such that for each j either $a_j = b_j$ or $a_j = \sigma(s(b_j))$, i.e. either the element was there before or it was matched and b_j is the replacement as specified by the rule. Moreover, if $R_i \in \tau_e$ then we require in the second case that at least one b_l was already in the original structure, i.e. $b_l = a_l$ for some $l \in \{1, \dots, r_i\}$.

To understand this definition it is best to consider an example, and one in which elements are both added and copied is presented in Figure 1. The labels a and b on the right-hand side of the rewriting rule depict the partial function s .

Logic and Constraints

The logic we use for specifying properties of states is an extension of first-order logic with real-valued terms and counting operators, cf. (Grädel and Gurevich 1998).

Syntax. We use first-order variables $x_1, x_2, \dots$ ranging over elements of the structure, and we define formulas φ and real-valued terms ρ by the following grammar ($n, m \in \mathbb{N}$).

$$\begin{aligned} \varphi &:= R_i(x_1, \dots, x_{r_i}) \mid x_i = x_j \mid \rho < \rho \\ &\mid \neg \varphi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \exists x_i \varphi \mid \forall x_i \varphi, \\ \rho &:= \frac{n}{m} \mid \rho + \rho \mid \rho \cdot \rho \mid \chi[\varphi] \mid \sum_{\bar{x} \models \varphi} \rho. \end{aligned}$$

Semantics. Most of the above operators are defined in the well known way, e.g. $\rho + \rho$ is the sum and $\rho \cdot \rho$ the product of two real-valued terms, and $\exists x \varphi(x)$ means that there exists an element a in the universe such that $\varphi(a)$ holds. Among less known operators, the term $\chi[\varphi]$ denotes the characteristic function of φ , i.e. the real-valued term which is 1 for all assignments for which φ holds and 0 for all other. The term $\sum_{\bar{x} \models \varphi} \rho$ denotes the sum of the values of $\rho(\bar{x})$ for all assignments of elements of the structure to $\bar{x}$ for which $\varphi(\bar{x})$ holds. Note that these terms can have free variables, e.g. the set of free variables of $\sum_{\bar{x} \models \varphi} \rho$ is the union of free variables of φ and free variables of ρ , minus the set $\{\bar{x}\}$.

The logic defined above is used in structure rewriting rules in two ways. First, it is possible to define a new relation $R(\bar{x})$ using a formula $\varphi(\bar{x})$ with free variables contained in $\bar{x}$. Defined relations can be used on left-hand sides of structure rewriting rules, but are not allowed on right-hand sides. The second way is to add *constraints* to a rule. A rule $\mathfrak{L} \rightarrow_s \mathfrak{R}$ can be constrained using two sentences (i.e. formulas without free variables): φ_{pre} and φ_{post} . In φ_{pre} we allow additional constants l for each $l \in \mathfrak{L}$ and in φ_{post} special constants for each $r \in \mathfrak{R}$ can be used. A rule $\mathfrak{L} \rightarrow_s \mathfrak{R}$ with such constraints can be applied on a match σ in $\mathfrak{A}$ only if the following holds: At the beginning, the formula φ_{pre} must hold in $\mathfrak{A}$ with the constants l interpreted as $\sigma(l)$, and, after the rule is applied, the formula φ_{post} must hold in the resulting structure with each r interpreted as the newly added element r (cf. Definition 2).

Structure Rewriting Games

One can in principle describe a game simply by providing a set of allowed moves for each player. Still, in many cases it is natural to specify the control flow directly. For this reason, we define games as labeled graphs as follows.

Definition 3. A *structure rewriting game* with k players is a finite directed graph in which each vertex, called *location*, is assigned a player from $\{1, \dots, k\}$ and k real-valued *payoff terms*, one for each player. Each edge of the graph represents a possible move and is labeled by a tuple

$$(\mathfrak{L} \rightarrow_s \mathfrak{R}, \tau_e, \varphi_{\text{pre}}, \varphi_{\text{post}}),$$

which specifies the rewriting rule to be used with relations to be embedded and a pre- and post-condition. Multiple edges with different labels are possible between two locations.

Play Semantics. A play of a structure rewriting game starts in a fixed initial location of the game graph and in a state given by a starting structure. The moving player chooses an edge and a match allowed by the label of the edge such that it can be applied, i.e. both the pre- and the post-condition holds. The play proceeds to the location to which the edge leads and the new state is the structure after the application of the rule on the chosen match. If in some location and state it is not possible to apply any of the rules on the outgoing edges, either because no match can be found or because of the constraints, then the play ends. Payoff terms from that location are evaluated on the state and determine the outcome of the game for all players.

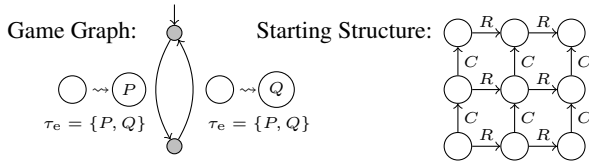


Figure 2: Tic-tac-toe as a structure rewriting game.

Example 4. Let us define Tic-tac-toe in our formalism. The starting structure has 9 elements connected by binary row and column relations, R and C , as depicted on the right in Figure 2. To mark the moves of the players we use unary relations P and Q , representing crosses and circles. The allowed move of the first player is to mark any unmarked element with P and the second player can mark with Q . Thus, there are two locations (gray) in the game graph, representing which player's turn it is, and two corresponding rules, both with one element on each side (left in Figure 2).

Observe that in both rules we require P and Q to be embedded. Note (see Definitions 1 and 2) that this ensures that no player can make a move where someone has already moved before. The two diagonals can be defined by

$$D_A(x, y) = \exists z(R(x, z) \wedge C(z, y)),$$

$$D_B(x, y) = \exists z(R(x, z) \wedge C(y, z))$$

and a line of three by

$$L(x, y, z) = (R(x, y) \wedge R(y, z)) \vee (C(x, y) \wedge C(y, z)) \vee (D_A(x, y) \wedge D_A(y, z)) \vee (D_B(x, y) \wedge D_B(y, z)).$$

Using these definitions, we specify the goal of the first player

$$\varphi = \exists x, y, z (P(x) \wedge P(y) \wedge P(z) \wedge L(x, y, z))$$

and the goal of the other player by an analogous formula φ' in which P is replaced by Q . The payoff terms are $\chi[\varphi] - \chi[\varphi']$ for the first player and $\chi[\varphi'] - \chi[\varphi]$ for the other, and to ensure that the game ends when one of the players has won, we take as a precondition of each move the negation of the goal formula of the other player.

The complete code for Tic-tac-toe, with the starting structure in a special syntax we use for grids with row and column relations, is given in Figure 3. Note that we write $:(\varphi)$ for $\chi[\varphi]$ and e.g. $Q:1 \{ \}$ for an empty relation Q of arity 1. Please refer to (Toss) for the complete input syntax and definitions of other, more complex games, e.g. Chess.

Type Normal Form

To derive evaluation heuristics from payoff terms, we first have to introduce a normal form of formulas which we exploit later in the construction. This normal form is in a sense a converse to the prenex normal form (PNF), because the quantifiers are pushed as deep inside the formula as possible. A very similar normal form has been used recently in a different context (Ganzow and Kaiser 2010). For a set of formulas Φ let us denote by $\mathcal{B}^+(\Phi)$ all positive Boolean combinations of formulas from Φ , i.e. define $\mathcal{B}^+(\Phi) = \Phi \mid \mathcal{B}^+(\Phi) \vee \mathcal{B}^+(\Phi) \mid \mathcal{B}^+(\Phi) \wedge \mathcal{B}^+(\Phi)$.

```

PLAYERS X, O
REL DgA(x, y) = ex z (R(x, z) and C(z, y))
REL DgB(x, y) = ex z (R(x, z) and C(y, z))
REL Row3(x, y, z) = R(x, y) and R(y, z)
REL Col3(x, y, z) = C(x, y) and C(y, z)
REL DgA3(x, y, z) = DgA(x, y) and DgA(y, z)
REL DgB3(x, y, z) = DgB(x, y) and DgB(y, z)
REL Conn3(x, y, z) =
  Row3(x, y, z) or Col3(x, y, z) or
  DgA3(x, y, z) or DgB3(x, y, z)
REL WinP() = ex x, y, z (P(x) and P(y) and P(z)
  and Conn3(x, y, z))
REL WinQ() = ex x, y, z (Q(x) and Q(y) and Q(z)
  and Conn3(x, y, z))

RULE Cross:
  [a | - | - ] -> [a | P(a) | - ]
  emb P, Q pre not WinQ()
RULE Circle:
  [a | - | - ] -> [a | Q(a) | - ]
  emb P, Q pre not WinP()
LOC 0 { PLAYER X
  PAYOFF {
    X: :(WinP()) - :(WinQ());
    O: :(WinQ()) - :(WinP())
  }
  MOVES [Cross -> 1] }
LOC 1 { PLAYER O
  PAYOFF {
    X: :(WinP()) - :(WinQ());
    O: :(WinQ()) - :(WinP())
  }
  MOVES [Circle -> 0] }
MODEL [ | P:1 { }; Q:1 { } | ] "
```

Figure 3: Tic-tac-toe in our game description formalism.

Definition 5. A formula is in TNF if and only if it is a positive Boolean combination of formulas of the following form

$$\tau = R_i(\bar{x}) \mid \neg R_i(\bar{x}) \mid x = y \mid x \neq y \mid \exists x \mathcal{B}^+(\tau) \mid \forall x \mathcal{B}^+(\tau)$$

satisfying the following crucial constraint: in $\exists x \mathcal{B}^+(\{\tau_i\})$ and $\forall x \mathcal{B}^+(\{\tau_i\})$ the free variables of *each* τ_i appearing in the Boolean combination *must contain* x .

We claim that for each formula φ there exists an equivalent formula ψ in TNF, and the procedure $\text{TNF}(\varphi)$ computes ψ given φ in negation normal form. Note that it uses sub-procedures DNF and CNF which, given a Boolean combination of formulas, convert it to disjunctive or respectively conjunctive normal form.

As an example, consider $\varphi = \exists x (P(x) \wedge (Q(y) \vee R(x)))$; This formula is not in TNF as $Q(y)$ appears under $\exists x$, and

$$\text{TNF}(\varphi) = (Q(y) \wedge \exists x P(x)) \vee \exists x (P(x) \wedge R(x)).$$

Procedure $\text{TNF}(\varphi)$

case φ is a literal **return** φ ;
case $\varphi = \varphi_1 \vee \varphi_2$ **return** $\text{TNF}(\varphi_1) \vee \text{TNF}(\varphi_2)$;
case $\varphi = \varphi_1 \wedge \varphi_2$ **return** $\text{TNF}(\varphi_1) \wedge \text{TNF}(\varphi_2)$;
case $\varphi = \exists x \psi$
 Let $\text{DNF}(\text{TNF}(\psi)) = \bigvee_i (\bigwedge_j \psi_j^i)$
 and $F_i = \{j \mid x \in \text{FreeVar}(\psi_j^i)\}$;
 return $\bigvee_i \left(\bigwedge_{j \notin F_i} \psi_j^i \wedge \exists x (\bigwedge_{j \in F_i} \psi_j^i) \right)$;
case $\varphi = \forall x \psi$
 Let $\text{CNF}(\text{TNF}(\psi)) = \bigwedge_i (\bigvee_j \psi_j^i)$
 and $F_i = \{j \mid x \in \text{FreeVar}(\psi_j^i)\}$;
 return $\bigwedge_i \left(\bigvee_{j \notin F_i} \psi_j^i \vee \forall x (\bigvee_{j \in F_i} \psi_j^i) \right)$;

Theorem 6. $\text{TNF}(\varphi)$ is equivalent to φ and in TNF.

The proof of the above theorem is a simple argument by induction on the structure of the formula, so we omit it here. Instead, let us give an example which explains why it is useful to compute TNF for the goal formulas.

Example 7. As already defined above, the payoff in Tic-tac-toe is given by $\exists x, y, z (P(x) \wedge P(y) \wedge P(z) \wedge L(x, y, z))$. To simplify this example, let us consider the payoff given only by row and column triples, i.e.

$$\varphi = \exists x, y, z (P(x) \wedge P(y) \wedge P(z) \wedge ((R(x, y) \wedge R(y, z)) \vee (C(x, y) \wedge C(y, z)))).$$

This formula is not in TNF and the DNF of the quantified part has the form $\varphi_1 \vee \varphi_2$, where

$$\varphi_1 = P(x) \wedge P(y) \wedge P(z) \wedge R(x, y) \wedge R(y, z),$$

$$\varphi_2 = P(x) \wedge P(y) \wedge P(z) \wedge C(x, y) \wedge C(y, z).$$

The procedure TNF must now choose the variable to first split on (this is discussed in the next section) and pushes the quantifiers inside, resulting in $\text{TNF}(\varphi) = \psi_1 \vee \psi_2$ with

$$\psi_1 = \exists x (P(x) \wedge \exists y (P(y) \wedge R(x, y) \wedge \exists z (P(z) \wedge R(y, z)))),$$

$$\psi_2 = \exists x (P(x) \wedge \exists y (P(y) \wedge C(x, y) \wedge \exists z (P(z) \wedge C(y, z)))).$$

In spirit, the TNF formula is thus more “step-by-step” than the goal formula we started with, and we exploit this to generate heuristics for evaluating positions below.

Heuristics from Existential Formulas

In this section, we present one method to generate a heuristic from an existential goal formula. As a first important step, we divide all relations appearing in the signature in our game into two sorts, *fluents* and *stable relations*. A relation is called *stable* if it is not changed by any of the structure rewriting rules which appear as possible moves, all other relations are *fluent*. We detect stable relations by a simple syntactic analysis of structure rewriting rules, i.e. we check which relations from the left-hand side remain unchanged on the right-hand side of the rule. It is a big advantage of our

formalism in comparison to GDL that stable relations (such as row and column relations used to represent the board) can so easily be separated from the fluents.

After detecting the fluents, our first step in generating the heuristic is to compute the TNF of the goal formula. As mentioned in the example above, there is certain freedom in the TNF procedure as to which quantified variable is to be resolved first. We use fluents to decide this — a variable which appears in a fluent will be resolved before all other variables which do not appear in any fluent literal (we choose arbitrarily in the remaining cases).

After the TNF has been computed, we change each sequence of existential quantifiers over conjunctions into a sum, counting how many steps towards satisfying the whole conjunction have been made. Let us fix a factor $\alpha < 1$ which we will discuss later. Our algorithm then changes a formula in the following way.

$$\begin{aligned} & \exists x_1 (\vartheta_1(x_1) \wedge \exists x_2 (\vartheta_1(x_2, x_1) \wedge \dots \wedge \exists x_n (\vartheta_n(x_n, \overline{x_i})) \dots)) \\ & \quad \downarrow \\ & \sum_{x_1 | \vartheta_1(x_1)} (\alpha^{n-1} + \sum_{x_2 | \vartheta_2(x_2, x_1)} (\alpha^{n-2} + \dots (\alpha + \sum_{x_n | \vartheta_n(x_n, \overline{x_i})} 1) \dots)) \end{aligned}$$

The sub-formulas $\vartheta_i(x_i, \overline{x})$ are in this case conjunctions of literals or formulas which contain universal quantifiers. The factor α defines how much more making each next step is valued over the previous one. When a formula contains disjunctions, we use the above schema recursively and sum the terms generated for each disjunct.

To compute a heuristic for evaluating positions from a payoff term, which is a real-valued expression in the logic defined above, we substitute all characteristic functions, i.e. expressions of the form $\chi[\varphi]$, by the sums generated for φ as described above.

Example 8. Consider the TNF of the simplified goal formula for Tic-tac-toe presented in the previous example and let $\alpha = \frac{1}{4}$. Since the TNF of the goal formula for one player has the form $\psi_1 \vee \psi_2$, we generate the following sums:

$$\begin{aligned} s_1 &= \sum_{x | P(x)} \left(\frac{1}{8} + \sum_{y | P(y) \wedge R(x, y)} \left(\frac{1}{4} + \sum_{z | P(z) \wedge R(y, z)} 1 \right) \right), \\ s_2 &= \sum_{x | P(x)} \left(\frac{1}{8} + \sum_{y | P(y) \wedge C(x, y)} \left(\frac{1}{4} + \sum_{z | P(z) \wedge C(y, z)} 1 \right) \right). \end{aligned}$$

Since the payoff is defined by $\chi[\varphi] - \chi[\varphi']$, where φ' is the goal formula for the other player, i.e. with Q in place of P , the total generated heuristic has the form

$$s_1 + s_2 - s'_1 - s'_2,$$

where s'_1 and s'_2 are as s_1 and s_2 but with P replaced by Q .

Finding Existential Descriptions

The method described above is effective if the TNF of the goal formulas has a rich structure of existential quantifiers. But this is not always the case, e.g. in Breakthrough the goal formula for white has the form $\exists x (W(x) \wedge \neg \exists y C(x, y))$, because $\neg \exists y C(x, y)$ describes the last row which the player

is supposed to reach. The general question which presents itself in this case is how, given an arbitrary relation $R(\bar{x})$ (as the last row above), can one construct an existential formula describing this relation. In this section, we present one method which turned out to yield useful formulas at least for common board games.

First of all, let us remark that the construction we present will be done only for relations defined by formulas which do not contain fluents. Thus, we can assume that the relation does not change during the game and we use the starting structure in the construction of the existential formula.

Our construction keeps a set C of conjunctions of stable literals. We say that a subset $\{\varphi_1, \dots, \varphi_n\} \subseteq C$ describes a relation $Q(\bar{x})$ in $\mathfrak{A}$ if and only if Q is equivalent in $\mathfrak{A}$ to the existentially quantified disjunction of φ_i 's, i.e. if

$$\mathfrak{A} \models Q(\bar{x}) \iff \mathfrak{A} \models \bigvee_i (\exists \bar{y}_i \varphi_i),$$

where $\bar{y}_i$ are all free variables of φ_i except for $\bar{x}$.

Our procedure extends the conjunctions from C with new literals until a subset which describes Q is found. These extensions can in principle be done in any order, but to obtain compact descriptions in reasonable time we perform them in a greedy fashion. The conjunctions are ordered by their hit-rank, defined as

$$\text{hit-rank}_{\mathfrak{A}, Q(\bar{x})}(\varphi) = \frac{|\{\bar{x} \in Q \mid \mathfrak{A} \models \exists \bar{y} \varphi(\bar{x})\}|}{|\{\bar{x} \mid \mathfrak{A} \models \exists \bar{y} \varphi(\bar{x})\}|},$$

where again $\bar{y} = \text{FreeVar}(\varphi) \setminus \bar{x}$. Intuitively, the hit-rank is the ratio of the tuples from Q which satisfy (existentially quantified) φ to the number of all such tuples. Thus, the hit-rank is 1 if φ describes Q and we set the hit-rank to 0 if φ is not satisfiable in $\mathfrak{A}$. We define the $\text{rank}_{\mathfrak{A}, Q}(\varphi, R(\bar{y}))$ as the maximum of the $\text{hit-rank}_{\mathfrak{A}, Q}(\varphi \wedge R(\bar{y}))$ and the $\text{hit-rank}_{\mathfrak{A}, Q}(\varphi \wedge \neg R(\bar{y}))$. The complete procedure is summarized below.

Procedure ExistentialDescription($\mathfrak{A}, Q$)

```

 $C \leftarrow \{\top\}$ 
while no subset of  $C$  describes  $Q(\bar{x})$  in  $\mathfrak{A}$  do
  for a stable relation  $R(\bar{y})$ , conjunction  $\varphi \in C$ 
    with maximal  $\text{rank}_{\mathfrak{A}, Q}(\varphi, R(\bar{y}))$  do
       $C \leftarrow (C \setminus \{\varphi\}) \cup \{\varphi \wedge R(\bar{y}), \varphi \wedge \neg R(\bar{y})\}$ 
  end
end

```

Since it is not always possible to find an existential description of a relation, let us remark that we stop the procedure if no description with a fixed number of literals is found. We also use a tree-like data structure for C to check the existence of a describing subset efficiently.

Example 9. As mentioned before, the last row on the board is defined by the relation $Q(x) = \neg \exists y C(x, y)$. Assume that we search for an existential description of this relation on a board with only the binary row and column relations (R and C) being stable, as in Figure 2. Since adding a row literal will not change the hit-rank, our construction will be

adding column literals one after another and will finally arrive, on a 3×3 board, at the following existential description: $\exists y_1, y_2 (C(y_1, y_2) \wedge C(y_2, x))$. Using such formula, the heuristic constructed in the previous section can count the number of steps needed to reach the last row for each pawn, which is important e.g. in Breakthrough.

Alternative Heuristics with Rule Conditions

The algorithm presented above is only one method to derive heuristics, and it uses only the payoff terms. In this section we present an alternative method, which is simpler and uses also the rewriting rules and their constraints. This simpler technique yields good heuristics only for games in which moves are monotone and relatively free, e.g. for Connect5.

Existential formulas are again the preferred input for the procedure, but this time we put them in prenex normal form at the start. As before, all universally quantified formulas are either treated as atomic relations or expanded, as discussed above. The Boolean combination under the existential quantifiers is then put in DNF and, in each conjunction in the DNF, we separate fluents from stable relations. After such preprocessing, the formula has the following form:

$$\exists \bar{x} ((\vartheta_1(\bar{x}) \wedge \psi_1(\bar{x})) \vee \dots \vee (\vartheta_n(\bar{x}) \wedge \psi_n(\bar{x}))),$$

where each $\vartheta_i(\bar{x})$ is a conjunction of fluents and each $\psi_i(\bar{x})$ is a conjunction of stable literals.

To construct the heuristic, we will retain the stable subformulas $\psi_i(\bar{x})$ but change the fluent ones $\vartheta_i(\bar{x})$ from conjunctions to sums. Formally, if $\vartheta_i(\bar{x}) = F_1(\bar{x}) \wedge \dots \wedge F_k(\bar{x})$ then we define $s_i(\bar{x}) = \chi[F_1(\bar{x})] + \dots + \chi[F_k(\bar{x})]$, and let $\delta_i(\bar{x}) = F_1(\bar{x}) \vee \dots \vee F_k(\bar{x})$ be a formula checking if the sum $s_i(\bar{x}) > 0$. The guard for our heuristic is defined as

$$\gamma(\bar{x}) = (\psi_1(\bar{x}) \vee \dots \vee \psi_n(\bar{x})) \wedge (\delta_1(\bar{x}) \vee \dots \vee \delta_n(\bar{x}))$$

and the heuristic with parameter m by

$$\sum_{\bar{x} \mid \gamma(\bar{x}) \wedge \text{move}(\bar{x})} (s_1(\bar{x}) + \dots + s_n(\bar{x}))^m.$$

The additional formula $\text{move}(\bar{x})$ is used to guarantee that at each element matched to one of the variables $\bar{x}$ it is still possible to make a move. This is done by converting the rewrite rule into a formula with free variables corresponding to the elements of the left-hand side structure, removing all the fluents F_i from above if these appear negated, and quantifying existentially if a new variable (not in $\bar{x}$) is created in the process. The following example shows how the procedure is applied for Tic-tac-toe.

Example 10. For Tic-tac-toe simplified as before (no diagonals), the goal formula in PNF and DNF reads:

$$\begin{aligned} \exists x, y, z \big(& (P(x) \wedge P(y) \wedge P(z) \wedge R(x, y) \wedge R(y, z)) \\ & \vee (P(x) \wedge P(y) \wedge P(z) \wedge C(x, y) \wedge C(y, z)) \big). \end{aligned}$$

The resulting guard is thus, after simplification,

$$\begin{aligned} \gamma(x, y, z) = & ((R(x, y) \wedge R(y, z)) \vee (C(x, y) \wedge C(y, z))) \\ & \wedge (P(x) \vee P(y) \vee P(z)). \end{aligned}$$

Since the structure rewriting rule for the move has only one element, say u , on its left-hand side, and $\tau_e = \{P, Q\}$ for this rule, the formula for the left-hand side reads $l(u) = \neg P(u) \wedge \neg Q(u)$. Because P appears as a fluent in γ we remove all occurrences of $\neg P$ from l and are then left with $\text{move}(u) = \neg Q(u)$. Since we require that a move is possible from all variables, the derived heuristic for one player with power 4 has the form

$$h = \sum_{x,y,z \mid \gamma(x,y,z) \wedge \neg Q(x) \wedge \neg Q(y) \wedge \neg Q(z)} (\chi[P(x)] + \chi[P(y)] + \chi[P(z)])^4.$$

Since the payoff expression is $\chi[\varphi] - \chi[\varphi']$, where φ' is the goal formula for the other player, we use $h - h'$ as the final heuristic to evaluate positions.

Experimental Results

The described algorithms are a part of (Toss), an open-source program implementing various logic functions as well as the presented game model and a GUI for the players. Toss contains an efficient implementation of CNF and DNF conversions and formula simplifications (interfacing a SAT solver, MiniSAT), and a model-checker for FO, which made it a good platform for our purpose.

We defined several board games in our formalism and created move translation scripts to play them against a GGP agent, Fluxplayer (Schiffel and Thielscher 2007). The tests were performed on the Dresden GGP Server².

We played 4 different games, 20 plays each, to illustrate how our heuristics work in games of various kinds. In these plays, Toss uses a basic constant-depth alpha-beta search algorithm. For the results in Table 1, Toss was using heuristics generated with parameter $\alpha = \frac{1}{4}$ or $m = 4$ in the alternative case, and the choice of the heuristic was made based on branching – the alternative one was used for Connect5.

	Toss Wins	Fluxplayer Wins	Tie
Breakthrough	95%	5%	0%
Connect4	20%	75%	5%
Connect5	0%	0%	100%
Pawn Whopping	50%	50%	0%
Total	41.25%	32.5%	26.25%

Table 1: Toss against Fluxplayer on 4 sample games.

The final total score was 4350 vs. 3650 for Toss, but, as you can see, it strongly depends on the game played. One important problem in games with low branching, i.e. both in Connect4 and Pawn Whopping, is that all leaves in the search tree of the basic alpha-beta have the same height. Fluxplayer uses a more adaptive search algorithm, and in tests against Toss with a variable-depth search, Toss did not lose any game and the final score was 5350 vs. 2650.

²Toss svn release 1315 was used to play on `euklid.inf.tu-dresden.de:8180/ggpserver/`. The plays are in 4 tournaments, one for each tested game, and can be viewed on-line after selecting Toss under Players. The variable-depth search plays are presented in 4 additional tournaments, with “variable_depth” suffix, and this algorithm is used in Toss release 0.6.

Perspectives

We presented a formalism for describing perfect information games and an algorithm to derive from this formalism heuristics for evaluating positions in the game. These are good enough to win against a state of the art GGP player and thus prove the utility of the introduced formalism.

The use of counting FO formulas for goals and move constraints allows to derive many more heuristics than the two we presented. Probably, a weighted combination of heuristics constructed in various ways from both the payoffs and the constraints would outperform any single one by far. One of the problems with this approach is the necessity to automatically learn the weights, which on the one hand requires a lot of time, but on the other hand promises an agent which improves with each play. One technique which we would like to try to apply in this context is boosting, as in (Freund and Schapire 1995), with heuristics used as experts.

Since it is very easy to construct new interesting patterns when operating on formulas, we also plan to explore how such formulas can be used to prune the search space. We imagine for example a formula which dictates the use of strategies which only place a new stone in a distance of at most 3 from some already present one. Such pruning may not be optimal in itself, but it may be the only way to perform a deeper search in games with large branching, and maybe it can be combined with some form of broad shallow search. Generally, the use of counting FO formulas facilitates many tasks and opens many new possibilities for GGP.

References

- Clune, J. E. 2008. *Heuristic Evaluation Functions for General Game Playing*. Dissertation, UCLA.
- Finnsson, H., and Björnsson, Y. 2008. Simulation-based approach to general game playing. In *AAAI'08*. AAAI Press.
- Freund, Y., and Schapire, R. E. 1995. A decision-theoretic generalization of on-line learning and an application to boosting. In *EuroCOLT'95*, volume 904 of *LNCS*, 23–37.
- Ganzow, T., and Kaiser, Ł. 2010. New algorithm for weak monadic second-order logic on inductive structures. In *CSL'10*, volume 6247 of *LNCS*, 366–380. Springer.
- Genesereth, M. R., and Love, N. 2005. General game playing: Overview of the AAAI competition. *AI Magazine* 26(2):62–72.
- Grädel, E., and Gurevich, Y. 1998. Metafinite model theory. *Information and Computation* 140:26–81.
- Kaiser, Ł. 2009. Synthesis for structure rewriting systems. In *MFCS'09*, volume 5734 of *LNCS*, 415–427. Springer.
- Rajlich, V. 1973. Relational structures and dynamics of certain discrete systems. In *MFCS'73*, 285–292.
- Schiffel, S., and Thielscher, M. 2007. Fluxplayer: A successful general game player. In *AAAI'07*, 1191–1196. AAAI Press.
- Thielscher, M. 2010. A general game description language for incomplete information games. In *AAAI'10*, 994–999. AAAI Press.
- Toss. <http://toss.sourceforge.net>.

Translating the Game Description Language to Toss

Łukasz Kaiser
CNRS & LIAFA
Paris

Łukasz Stafiniak
Institute of Computer Science
University of Wrocław

Abstract

We show how to translate games defined in the Game Description Language (GDL) into the Toss format. GDL is a variant of Datalog used to specify games in the General Game Playing Competition. Specifications in Toss are more declarative than in GDL and make it easier to capture certain useful game characteristics. The presented translation must thus detect structural properties of games which are not directly visible in the GDL specification.

Introduction

General Game Playing (GGP) is concerned with the construction of programs able to play arbitrary games without specific information about the game present in the program. GGP programs compete against each other in the GGP Competition, where games are specified in the Game Description Language (GDL), cf. (Genesereth and Love 2005). A successful GGP agent must reason about the rules of the game and extract from them game-specific knowledge. To facilitate the creation of good general players, GDL was designed as a high-level, declarative language. Still, it is not directly clear from the GDL specification of a game what parts constitute a board, what the pieces are, etc. — the GDL defines only Datalog terms, no high-level game concepts.

For this reason, we introduce a new formalism, which is more tightly related to games and even higher-level and more declarative than GDL. In a companion paper (Kaiser and Stafiniak 2011) we show how these higher-level features of the introduced formalism can be used to build a competitive GGP player more easily than starting from GDL itself. But to compete against GGP players, it is necessary to translate games from GDL into the presented formalism, which in itself presents several challenges. In this work, we show how a good translation can be made, assuming certain restrictions on the GDL specification.

Games in the Toss Formalism

Since the Toss formalism is not very well known, we repeat here several definitions from the paper (Kaiser and Stafiniak 2011), but with support for concurrency and fixed-points.

The state of the game is represented in the Toss formalism by a finite relational structure, i.e. a labelled directed hypergraph. A relational structure $\mathfrak{A} = (A, R_1, \dots, R_k)$ is composed of a universe A and a number of relations $R_1, \dots, R_k$.

We denote the arity of R_i by r_i , so $R_i \subseteq A^{r_i}$. The *signature* of $\mathfrak{A}$ is the set of symbols $\{R_1, \dots, R_k\}$.

Relational Structure Rewriting

The moves of the players are described by *structure rewriting rules*, a generalised form of term and graph rewriting. Structure rewriting has been introduced in (Rajlich 1973) and is most recognised in graph rewriting and software engineering communities, where it is regarded as easy to understand and well suited for visual programming.

In our setting, a rule $\mathfrak{L} \rightarrow_s \mathfrak{R}$ consists of two finite relational structures, $\mathfrak{L}$ and $\mathfrak{R}$, over the same signature, and of a partial function $s : \mathfrak{R} \rightarrow \mathfrak{L}$ specifying which elements of $\mathfrak{L}$ will be substituted by which elements of $\mathfrak{R}$. With each rule, we will also associate a set of relations symbols τ_e to be embedded exactly, as described below.

Definition 1. Let $\mathfrak{A}$ and $\mathfrak{B}$ be two relational structures over the same signature. A function $f : \mathfrak{A} \rightarrow \mathfrak{B}$ is a τ_e -*embedding* if f is injective, for each $R_i \in \tau_e$ it holds that

$$(a_1, \dots, a_{r_i}) \in R_i^{\mathfrak{A}} \Leftrightarrow (f(a_1), \dots, f(a_{r_i})) \in R_i^{\mathfrak{B}},$$

and for $R_j \notin \tau_e$ it holds that

$$(a_1, \dots, a_{r_j}) \in R_j^{\mathfrak{A}} \Rightarrow (f(a_1), \dots, f(a_{r_j})) \in R_j^{\mathfrak{B}}.$$

A τ_e -*match* of the rule $\mathfrak{L} \rightarrow_s \mathfrak{R}$ in another structure $\mathfrak{A}$ is a τ_e -embedding $\sigma : \mathfrak{L} \rightarrow \mathfrak{A}$.

Definition 2. The result of an application of $\mathfrak{L} \rightarrow_s \mathfrak{R}$ to $\mathfrak{A}$ on the match σ , denoted $\mathfrak{A}[\mathfrak{L} \rightarrow_s \mathfrak{R}/\sigma]$, is a relational structure $\mathfrak{B}$ with universe $(A \setminus \sigma(L)) \dot{\cup} R$, and relations given as follows. A tuple $(b_1, \dots, b_{r_i})$ is in the new relation $R_i^{\mathfrak{B}}$ if and only if either it is in the relation in $\mathfrak{R}$ already, $(b_1, \dots, b_{r_i}) \in R_i^{\mathfrak{R}}$, or there exists a tuple in the previous structure, $(a_1, \dots, a_{r_i}) \in R_i^{\mathfrak{A}}$, such that for each j either $a_j = b_j$ or $a_j = \sigma(s(b_j))$, i.e. either the element was there before or it was matched and b_j is the replacement as specified by the rule. Moreover, if $R_i \in \tau_e$ then we require in the second case that at least one b_l was already in the original structure, i.e. $b_l = a_l$ for some $l \in \{1, \dots, r_i\}$.

Logic and Constraints

The logic we use for specifying properties of states is an extension of first-order logic with least and greatest fixed-points, real-valued terms and counting operators, cf. (Grädel 2007; Grädel and Gurevich 1998).

Syntax. We use first-order variables $x_1, x_2, \dots$ ranging over elements of the structure, second-order variables $X_1, X_2, \dots$ ranging over relations, and we define formulas $\varphi \in \mathcal{F}_{\mathfrak{A}}$ and real-valued terms ρ by the following grammar ($n, m \in \mathbb{N}$), with second-order variables restricted to appear only positively, as usual in the least fixed-point logic.

$$\begin{aligned} \varphi &:= R_i(x_1, \dots, x_{r_i}) \mid x_i = x_j \mid \rho < \rho \mid \neg \varphi \mid \varphi \vee \varphi \\ &\quad \mid \varphi \wedge \varphi \mid \exists x_i \varphi \mid \forall x_i \varphi \mid \text{lfp } X_i \varphi \mid \text{gfp } X_i \varphi, \\ \rho &:= \frac{n}{m} \mid \rho + \rho \mid \rho \cdot \rho \mid \chi[\varphi] \mid \sum_{\bar{x} \mid \varphi} \rho. \end{aligned}$$

Semantics. Most of the above operators are defined in the well known way, e.g. $\rho + \rho$ is the sum and $\rho \cdot \rho$ the product of two real-valued terms, and $\text{lfp } X \varphi(X)$ is the least fixed-point of the equation $X = \varphi(X)$. Among less known operators, the term $\chi[\varphi]$ denotes the characteristic function of φ , i.e. the real-valued term which is 1 for all assignments for which φ holds and 0 for all other. The term $\sum_{\bar{x} \mid \varphi} \rho$ denotes the sum of the values of $\rho(\bar{x})$ for all assignments of elements of the structure to $\bar{x}$ for which $\varphi(\bar{x})$ holds. Note that these terms can have free variables, e.g. the set of free variables of $\sum_{\bar{x} \mid \varphi} \rho$ is the union of free variables of φ and free variables of ρ , minus the set $\{\bar{x}\}$.

The logic defined above is used in structure rewriting rules in two ways. First, it is possible to define a new relation $R(\bar{x})$ using a formula $\varphi(\bar{x})$ with free variables contained in $\bar{x}$. Defined relations can be used on left-hand sides of structure rewriting rules, but are not allowed on right-hand sides. The second way is to add *constraints* to a rule. A rule $\mathfrak{L} \rightarrow_s \mathfrak{R}$ can be constrained using two sentences (i.e. formulas without free variables): φ_{pre} and φ_{post} . In φ_{pre} we allow additional constants l for each $l \in \mathfrak{L}$ and in φ_{post} special constants for each $r \in \mathfrak{R}$ can be used. A rule $\mathfrak{L} \rightarrow_s \mathfrak{R}$ with such constraints can be applied on a match σ in $\mathfrak{A}$ only if the following holds: At the beginning, the formula φ_{pre} must hold in $\mathfrak{A}$ with the constants l interpreted as $\sigma(l)$, and, after the rule is applied, the formula φ_{post} must hold in the resulting structure with each r interpreted as the newly added element r (cf. Definition 2).

Structure Rewriting Games

One can in principle describe a game simply by providing a set of allowed moves for each player. Still, in many cases it is natural to specify the control flow directly. For this reason, we define games as labelled graphs as follows.

Definition 3. A *structure rewriting game* with k players is a finite directed graph in which in each vertex, called *location*, we assign to each player from $\{1, \dots, k\}$ a real-valued *pay-off term*. Each edge of the graph represents a possible move and is labelled by a tuple

$$(p, \mathfrak{L} \rightarrow_s \mathfrak{R}, \tau_e, \varphi_{\text{pre}}, \varphi_{\text{post}}),$$

which specifies the player $p \in \{1, \dots, k\}$ who moves and the rewriting rule to be used with relations to be embedded and a pre- and post-condition. Multiple edges with different labels are possible between two locations.

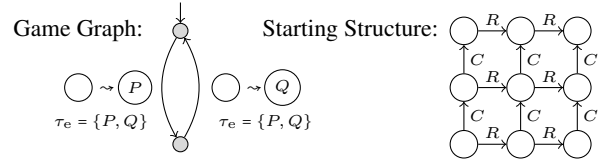


Figure 1: Tic-tac-toe as a structure rewriting game.

Play Semantics. A play of a structure rewriting game starts in a fixed initial location of the game graph and in a state given by a starting structure. A player chooses an edge on which she moves and a match allowed by the label of the edge such that it can be applied, i.e. both the pre- and the post-condition holds. The play proceeds to the location to which the edge leads and the new state is the structure after the application of the rule on the chosen match. If in some location and state it is not possible to apply any of the rules on the outgoing edges, either because no match can be found or because of the constraints, then the play ends. Pay-off terms from that location are evaluated on the state and determine the outcome of the game for all players.

Example 4. Let us define Tic-tac-toe in our formalism. The starting structure has 9 elements connected by binary row and column relations, R and C , as depicted on the right in Figure 1. To mark the moves of the players we use unary relations P and Q . The allowed move of the first player is to mark any unmarked element with P and the second player can mark with Q . Thus, there are two locations in the game graph (representing which player's turn it is) and two corresponding rules, both with one element on each side (left in Figure 1). The two diagonals can be defined by

$$D_A(x, y) = \exists z (R(x, z) \wedge C(z, y)),$$

$$D_B(x, y) = \exists z (R(x, z) \wedge C(y, z))$$

and a line of three by

$$\begin{aligned} L(x, y, z) = & (R(x, y) \wedge R(y, z)) \vee (C(x, y) \wedge C(y, z)) \vee \\ & (D_A(x, y) \wedge D_A(y, z)) \vee (D_B(x, y) \wedge D_B(y, z)). \end{aligned}$$

Using these definitions, we specify the goal of the first player

$$\varphi = \exists x, y, z (P(x) \wedge P(y) \wedge P(z) \wedge L(x, y, z))$$

and the goal of the other player by an analogous formula φ' in which P is replaced by Q . The payoff terms are $\chi[\varphi] - \chi[\varphi']$ for the first player and $\chi[\varphi'] - \chi[\varphi]$ for the other, and to ensure that the game ends when one of the players has won, we take as a precondition of each move the negation of the goal formula of the other player.

The complete code for Tic-tac-toe, with the starting structure in a special syntax we use for grids with row and column relations, is given in Figure 2. Note that we write $:(\varphi)$ for $\chi[\varphi]$ and e.g. $Q:1 \ \{\}$ for an empty relation Q of arity 1. Please refer to (Toss) for the complete input syntax and definitions of other, more complex games, e.g. Chess.

```

PLAYERS X, O
REL Row3(x, y, z) = R(x, y) and R(y, z)
REL Col3(x, y, z) = C(x, y) and C(y, z)
REL DgA(x, y) = ex z (R(x, z) and C(z, y))
REL DgB(x, y) = ex z (R(x, z) and C(y, z))
REL DgA3(x, y, z) = DgA(x, y) and DgA(y, z)
REL DgB3(x, y, z) = DgB(x, y) and DgB(y, z)
REL Conn3(x, y, z) =
  Row3(x, y, z) or Col3(x, y, z) or
  DgA3(x, y, z) or DgB3(x, y, z)
REL WinP() = ex x,y,z (P(x) and P(y) and P(z)
  and Conn3(x, y, z))
REL WinQ() = ex x,y,z (Q(x) and Q(y) and Q(z)
  and Conn3(x, y, z))

RULE Cross:
  [a | - | - ] -> [a | P (a) | - ]
  emb P, Q pre not WinQ()
RULE Circle:
  [a | - | - ] -> [a | Q (a) | - ]
  emb P, Q pre not WinP()
LOC 0 {
  PLAYER X {PAYOFF : (WinP()) - : (WinQ())
    MOVES [Cross -> 1] }
  PLAYER O {PAYOFF : (WinQ()) - : (WinP()) }
}
LOC 1 {
  PLAYER X {PAYOFF : (WinP()) - : (WinQ()) }
  PLAYER O {PAYOFF : (WinQ()) - : (WinP())
    MOVES [Circle -> 0] }
}
MODEL [ | P:1 {} ; Q:1 {} | ] "

. . .
. . .
. . .
"

```

Figure 2: Tic-tac-toe in the Toss formalism.

Game Description Language

The game description language, GDL, is a variant of Datalog used to specify games in a compact, prolog-like way. The GDL syntax and semantics are defined in (Genesereth and Love 2005; Love et al. 2008), we refer the reader there for the definition and will only recapitulate some notions here.

The state of the game in GDL is defined by the set of propositions true in that state. These propositions are represented by terms of limited height. The moves of the game, i.e. the transition function between the states, are described using Datalog rules — clauses define which predicates hold in the subsequent state. In this way a transition system is specified in a compact way. Additionally, there are 8 special relations in GDL: *role*, *init*, *true*, *does*, *next*, *legal*, *goal* and *terminal*, which are used to describe the game: initial state, the players, their goals, and thus like.

We say that *GDL state terms* are the terms that are possible arguments of *true*, *next* and *init* relations in a GDL specification, i.e. those terms which can define the state of the game. The *GDL move terms* are ground instances of the second arguments of *legal* and *does* relations, i.e. those terms which are used to specify the moves of the players.

The complete Tic-tac-toe specification in GDL is given in Figure 3. While games can be formalised in various ways in both systems, Figures 2 and 3 give natural examples of a formalisation, similar to several other games.

Notions Related to Terms

Since GDL is a term-based formalism, we will use the standard term notions, as e.g. in the preliminaries of (Comon et al. October 2007). We understand terms as finite trees with ordered successors and labelled by the symbols used in the current game, with leafs possibly labelled by variables.

Substitutions. A *substitution* is an assignment of terms to variables. Given a substitution σ and a term t we write $\sigma(t)$ to denote the result of applying σ to t , i.e. of replacing all variables in t which also occur in σ by the corresponding terms. We extend this notation to tuples in the natural way.

MGU. We say that a tuple of terms $\bar{t}$ is *more general* than another tuple $\bar{s}$ of equal length, written $\bar{s} \leq \bar{t}$, if there is a substitution σ such that $\bar{s} = \sigma(\bar{t})$. Given two tuples of terms $\bar{s}$ and $\bar{t}$ we write $\bar{s} \dot{=} \bar{t}$ to denote that these tuples *unify*, i.e. that there exists a substitution σ such that $\sigma(\bar{s}) = \sigma(\bar{t})$. In such case there exists a most general substitution of this kind, and we denote it by $\text{MGU}(\bar{s}, \bar{t})$.

Paths. A *path* in a term is a sequence of pairs of function symbols and natural numbers denoting which successor to take in turn, e.g. $p = (f, 1)(g, 2)$ denotes the second child of a node labelled by g , which is the first child of a node labelled by f . For a term t we write $t \downarrow_p$ to denote the subterm of t at path p , and that t has a path p , i.e. that the respective sequence of nodes exists in t with exactly the specified labels. Using $p = (f, 1)(g, 2)$ as an example, $f(g(a, b), c) \downarrow_p = b$, but $g(f(a, b), c) \downarrow_p$ is false. Similarly, for a formula φ , we write $\varphi(t \downarrow_p)$ to denote that t has path p and the subterm $r = t \downarrow_p$ satisfies $\varphi(r)$. A path can be an empty sequence ε and $t \downarrow_\varepsilon = t$ for all terms t .

For any terms t, s and any path p existing in t , we write $t[p \leftarrow s]$ to denote the result of *placing* s at path p in t , i.e. the term t' such that $t' \downarrow_p = s$ and on all other paths q , i.e. ones which neither are prefixes of p nor contain p as a prefix, t' is equal to t , i.e. $t' \downarrow_q = t \downarrow_q$. We extend this notation to sets of paths as well: $t[P \leftarrow s]$ places s at all paths from P in t .

Translation

In this section, we describe our main construction. Given a GDL specification of a game G , which satisfies the restrictions described in the last section, we construct a Toss game $T(G)$ which represents exactly the same game. Moreover, we define a bijection μ between the moves possible in G and in $T(G)$ in each reachable state, so that the following correctness theorem holds.

```

(role x)
(role o)
(init (cell a a b))
(init (cell b a b))
(init (cell c a b))
(init (cell a b b))
(init (cell b b b))
(init (cell c b b))
(init (cell a c b))
(init (cell b c b))
(init (cell c c b))
(init (control x))
(<= (next (control ?r)) (does ?r noop))
(<= (next (cell ?x ?y ?r))
    (does ?r (mark ?x ?y)))
(<= (next (cell ?x ?y ?c))
    (true (cell ?x ?y ?c))
    (does ?r (mark ?x1 ?y1))
    (or (distinct ?x ?x1) (distinct ?y ?y1)))
(<= (legal ?r (mark ?x ?y))
    (true (control ?r))
    (true (cell ?x ?y b)))
(<= (legal ?r noop) (role ?r)
    (not (true (control ?r))))
(<= (goal ?r 100) (conn3 ?r))
(<= (goal ?r 50) (role ?r)
    (not exists_line3))
(<= (goal x 0) (conn3 o))
(<= (goal o 0) (conn3 x))
(<= terminal exists_line3)
(<= terminal (not exists_blank))
(<= exists_blank (true (cell ?x ?y b)))
(<= exists_line3 (role ?r) (conn3 ?r))
(<= (conn3 ?r) (or (col ?r) (row ?r)
    (diag1 ?r) (diag2 ?r)))
(<= (row ?r)
    (true (cell ?a ?y ?r)) (nextcol ?a ?b)
    (true (cell ?b ?y ?r)) (nextcol ?b ?c)
    (true (cell ?c ?y ?r)))
(<= (col ?r)
    (true (cell ?x ?a ?r)) (nextcol ?a ?b)
    (true (cell ?x ?b ?r)) (nextcol ?b ?c)
    (true (cell ?x ?c ?r)))
(<= (diag1 ?r)
    (true (cell ?x1 ?y1 ?r))
    (nextcol ?x1 ?x2) (nextcol ?y1 ?y2)
    (true (cell ?x2 ?y2 ?r))
    (nextcol ?x2 ?x3) (nextcol ?y2 ?y3)
    (true (cell ?x3 ?y3 ?r)))
(<= (diag2 ?r)
    (true (cell ?x1 ?y5 ?r))
    (nextcol ?x1 ?x2) (nextcol ?y4 ?y5)
    (true (cell ?x2 ?y4 ?r))
    (nextcol ?x2 ?x3) (nextcol ?y3 ?y4)
    (true (cell ?x3 ?y3 ?r)))
(nextcol a b)
(nextcol b c)

```

Figure 3: Tic-tac-toe in the Game Description Language.

Theorem 5 (Correctness).

Let S be any state of G reached from the initial one by a sequence of moves $m_1 \dots m_n$. We write $\mu(S)$ for the state of $T(G)$ reached by $\mu(m_1) \dots \mu(m_n)$. The following conditions are satisfied.

- The function μ defines a bijection between the moves possible in S and in $\mu(S)$ for each player.
- If no move is possible in S (and in $\mu(S)$), then the payoffs in G evaluate to the same value as those in $T(G)$.

We will not prove this theorem here, but the construction presented below should make it clear why the exact correspondence holds. For the rest of this section let us fix the GDL game specification G we will translate. We begin by transforming G itself: eliminating variables clearly referring to players (i.e. arguments of positive role atoms, first arguments to positive does atoms and to legal) by substituting them by players of G (i.e. arguments of role facts), duplicating the clauses. From this transformed specification, we derive the elements of the Toss structure (next subsection), the relations (subsection after the next), the rewriting rules (further subsection) and finally the move translation function (last subsection).

Elements of the Toss Structure

By definition of GDL, the state of the game is described by a set of propositions true in that state. Let us denote by $\mathcal{S}$ the set of all GDL state terms which are true at some game state reachable from the initial state of G .

For us, it is enough to approximate $\mathcal{S}$ from above. To approximate $\mathcal{S}$, we currently perform an *aggregate payout*, i.e. a symbolic play in where all players take all their legal moves in a state. Since an approximation is sufficient, we check only the positive part of the legality condition of each move.

To construct the elements of the structure from state terms, and to make that structure a good representation of the game in Toss, we first determine which state terms always have common subtrees.

Definition 6. For two terms s and t we say that a set of paths P merges s and t if each $p \in P$ exists both in s and t and $t[P \leftarrow c] = s[P \leftarrow c]$ for all terms c . We denote by $d\mathcal{P}(s, t)$ the unique set P of paths merging s and t for which the size of $t[P \leftarrow c]$ is maximal and no subset of which merges s and t . Intuitively, $t[d\mathcal{P}(s, t) \leftarrow c]$ is the largest common subtree of s and t , the bigger its size the more similar s and t are.

Let Next_e be the set of next clauses in G with all atoms of does expanded (inlined) by the legal clause definitions, duplicating the next clause when more than one head of legal unifies with the does atom. Intuitively, these are expanded forms of clauses defining game state change.

For each clause $\mathcal{C} \in \text{Next}_e$, we select two terms $s_{\mathcal{C}}$ and $t_{\mathcal{C}}$ in the following way. The term $s_{\mathcal{C}}$ is simply the second part of the head of the clause (next $s_{\mathcal{C}}$). The term $t_{\mathcal{C}}$ is the argument of true in the body of $\mathcal{C}$ which is most similar to s in the sense of Definition 6, and of equally similar has smallest $d\mathcal{P}(s, t)$ (if there are several, we pick one in an arbitrary way).

We often use the word *fluent* for changing objects, and so we define the set of *fluent paths*, $\mathcal{P}_f$, in the following way. We say that a term t is a *negative true* in a clause $\mathcal{C}$ if it is the argument of a negative occurrence of `true` in $\mathcal{C}$. We write $\mathcal{L}(t)$ for the set of path to all constant leaves in t . The set

$$\mathcal{P}_f = \bigcup_{\mathcal{C} \in \text{Next}_e} d\mathcal{P}(s_{\mathcal{C}}, t_{\mathcal{C}}) \cup \bigcup_{\mathcal{C} \in \text{Next}_e, t_{\mathcal{C}} \text{ negative true in } \mathcal{C}} \mathcal{L}(t_{\mathcal{C}}).$$

Note that $\mathcal{P}_f$ contains all merge sets for the selected terms in Next_e clauses, and additionally, when $t_{\mathcal{C}}$ is a negative true, we add the paths to all constant leaves in $t_{\mathcal{C}}$.

Example 7. There are three `next` clauses in Figure 3. $\mathcal{C}_1$:

```
(<= (next (cell ?x ?y ?c))
    (true (cell ?x ?y ?c))
    (does ?r (mark ?x1 ?y1))
    (or (distinct ?x ?x1) (distinct ?y ?y1)))
```

does not lead to any fluent paths, as $(\text{cell } ?x ?y ?c)$ is $s_{\mathcal{C}_1} = t_{\mathcal{C}_1}$ and thus $d\mathcal{P}(s_{\mathcal{C}_1}, t_{\mathcal{C}_1}) = \emptyset$. The clause:

```
(<= (next (cell ?x ?y ?r))
    (does ?r (mark ?x ?y)))
```

expands to:

```
(<= (next (cell ?x ?y x))
    (true (control x))
    (true (cell ?x ?y b)))
(<= (next (cell ?x ?y o))
    (true (control o))
    (true (cell ?x ?y b)))
```

These generate the fluent path $(\text{cell}, 3)$. The clause:

```
(<= (next (control ?r)) (does ?r noop))
```

expands to:

```
(<= (next (control x))
    (not (true (control x))))
(<= (next (control o))
    (not (true (control o))))
```

These generate the fluent path $(\text{control}, 1)$ since $(\text{control } x)$ and $(\text{control } o)$ are negative trues. In the end $\mathcal{P}_f = \{(\text{cell}, 3), (\text{control}, 1)\}$.

The fluent paths define the partition of GDL state terms into elements of the Toss structures in the following way.

Definition 8. We define the *element mask equivalence* $\sim$ by:

$$t \sim s \iff t[P_f \leftarrow c] = s[P_f \leftarrow c] \text{ for all terms } c.$$

The set of elements A of the initial Toss structure $\mathfrak{A}$ consists of equivalence classes of $\sim$. For $a \in A$ we write $\llbracket a \rrbracket$ to denote the corresponding subset of equivalent terms from $\mathcal{S}$.

We define *paths within mask* $\mathcal{P}_m$ as such paths p that, for all $a \in A$, if, for any $t \in \llbracket a \rrbracket$, $t \downarrow_p$, then for all $s, t \in \llbracket a \rrbracket$, $s \downarrow_p = t \downarrow_p$. For $p \in \mathcal{P}_m$ we can therefore define the *mask subterm* $a \downarrow_p^m$ as $t \downarrow_p$ for $t \in \llbracket a \rrbracket$.

Example 9. Continuing the example of the Tic-tac-toe specification from Figure 3, we construct the set A . The terms in $\mathcal{S}$ are either $(\text{cell } s t p)$ or $(\text{control } q)$, where s and t range over a, b, c , p over x, o, b and q can be x or o . Since $\mathcal{P}_f = \{(\text{cell}, 3), (\text{control}, 1)\}$, we consider as

$\sim$ -equivalent all `cell` terms which differ only on p and all `control` terms which differ on q . Thus, the set A consists of 10 elements: the element a_{ctrl} for the single equivalence class of control terms, and 9 elements $a_{s,t}$ for the equivalence classes of $(\text{cell } s t p)$ with fixed s and t .

$$A = \{a_{ctrl}, a_{a,a}, a_{a,b}, a_{a,c}, a_{b,a}, a_{b,b}, a_{b,c}, a_{c,a}, a_{c,b}, a_{c,c}\}.$$

Note the similarity to the starting structure in Figure 1, up to the control element. The set of paths within masks for this specification is $\mathcal{P}_m = \{(\text{cell}, 1), (\text{cell}, 2)\}$.

Relations in the Structure

Having defined the elements A as equivalence classes of state terms, let us now define the relations in the initial structure $\mathfrak{A}$.

Subterm equality relations. For all pairs of paths $p, q \in \mathcal{P}_m$ we introduce the *subterm equality relation* $Eq_{p,q}$:

$$Eq_{p,q}(a_1, a_2) \iff a_1 \downarrow_p^m = a_2 \downarrow_q^m.$$

Fact relations. For all relations R of G that do not (directly or indirectly) depend on the state, and all tuples of paths $p_1, \dots, p_n \in \mathcal{P}_m$, we introduce the *fact relation* $R_{p_1, \dots, p_n}$:

$$R_{p_1, \dots, p_n}(a_1, \dots, a_n) \iff R(a_1 \downarrow_{p_1}^m, \dots, a_n \downarrow_{p_n}^m) \text{ in any state.}$$

Anchor predicates. For all paths $p \in \mathcal{P}_m$ and subterms $s = t \downarrow_p, t \in \mathcal{S}$, we introduce the *anchor predicate* $Anch_p^s(a)$:

$$Anch_p^s(a) \iff a \downarrow_p^m = s.$$

Fluent predicates. Let $\mathcal{S}^{\text{init}} = \{s \mid \text{init}(s) \in G\}$ be the set of state terms under `init`. For all paths $p \in \mathcal{P}_f$ and subterms $s = t \downarrow_p, t \in \mathcal{S}$, we introduce the *fluent predicate* $Flu_p^s(a)$:

$$Flu_p^s(a) \iff t \downarrow_p = s \text{ for some } t \in \llbracket a \rrbracket \cap \mathcal{S}^{\text{init}}.$$

Mask predicates. We say that a term m is a *mask term* if the paths to all variables of m are contained in $\mathcal{P}_m \cup \mathcal{P}_f$ and for each $p \in \mathcal{P}_m \cup \mathcal{P}_f$ if p exists in m then $m \downarrow_p$ is a variable. We say that m *masks* a term t if m is a mask term and there exists a substitution σ such that $\sigma(m) = t$. For all mask terms $m \in \mathcal{S}$ we introduce the *mask predicate* $Mask_m$. Mask predicates are similar to the anchor predicates, but instead of matching against a subterm, they match against the mask.

$$Mask_m(a) \iff m \text{ masks all } t \in \llbracket a \rrbracket.$$

Example 10. To list the relations derived for the Tic-tac-toe specification, recall that $\mathcal{P}_m = \{(\text{cell}, 1), (\text{cell}, 2)\}$ consists of two paths. To shorten notation, we will just use the index i for (cell, i) .

Subterm equality relations. The relation $Eq_{i,j}$ contains all pairs of elements for which the i th coordinate of the first one equals the j th coordinate of the second one. For example

$$Eq_{1,1} = \{(a_{a,a}, a_{a,a}), (a_{a,a}, a_{a,b}), (a_{a,a}, a_{a,c}), \dots, (a_{c,c}, a_{c,a}), (a_{c,c}, a_{c,b}), (a_{c,c}, a_{c,c})\}$$

describes the identity of the first coordinate of two cells.

Fact relations. The only relation in the example specification is `nextcol` and we thus get the relations `nextcoli,j`. For example, the relation

$$\text{nextcol}_{2,2} = \{(a_{a,a}, a_{a,b}), (a_{a,a}, a_{b,b}), (a_{a,a}, a_{c,b}), \dots, (a_{c,b}, a_{a,c}), (a_{c,b}, a_{b,c}), (a_{c,b}, a_{c,c})\}$$

contains pairs in which the second element is in the successive row of the first one. Note that, for example, the formula $Eq_{1,1}(x_1, x_2) \wedge \text{nextcol}_{2,2}(x_1, x_2)$ specifies that x_2 is directly right of x_1 in the same row.

Anchor predicates. Since the terms inside `cell` at positions 1 and 2 range over a, b, c , we get 6 anchor predicates $Anch_i^a, Anch_i^b, Anch_i^c$ for $i = 1, 2$. They mark the corresponding terms, e.g.

$$Anch_2^a = \{a_{a,a}, a_{b,a}, a_{c,a}\}$$

describes the bottom row.

Fluent predicates. The fluent paths $\mathcal{P}_f = \{(\text{cell}, 3), (\text{control}, 1)\}$ and the terms appearing there are b, x, o for $(\text{cell}, 3)$ and x, o for $(\text{control}, 1)$, resulting in 5 fluent predicates. For example, $Flu_{(\text{cell}, 3)}^o(a)$ will hold exactly for the elements a which are marked by the player o . In the initial structure, the only nonempty fluent predicates are

$$Flu_{(\text{cell}, 3)}^b = A \setminus \{a_{ctrl}\} \quad \text{and} \quad Flu_{(\text{control}, 1)}^x = \{a_{ctrl}\}.$$

Mask predicates. For the specification we consider, there are two mask terms: $m_1 = (\text{control } x)$ and $m_2 = (\text{cell } x \ y \ z)$. The predicate $Mask_{m_1} = \{a_{ctrl}\}$ holds exactly for the control element, and $Mask_{m_2} = A \setminus \{a_{ctrl}\}$ contains these elements of A which are not the control element, i.e. the board elements.

In Toss, *stable relations* are relations that do not change in the course of the game, and *fluents* are relations that do change. Roughly speaking, a fluent occurs in the symmetric difference of the sides of a structure rewrite rule. In the translation, the fluent predicates Flu_p^s are the only introduced fluents, i.e. these predicates will change when players play the game and all other predicates will remain intact.

Structure Rewriting Rules

To create the structure rewriting rule for the Toss game, we first construct two types of clauses and then transform them into structure rewriting rules. Let $(p_1, \dots, p_n)$ be the players in G , i.e. let there be $(\text{role } p_1)$ up to $(\text{role } p_n)$ facts in G , in this order.

Move Clauses By GDL specification, a legal joint move of the players is a tuple of player term – move term pairs which satisfy the `legal` relation. For a joint move $(m_1, \dots, m_n)$ to be allowed, it is necessary that there is a tuple of legal clauses $(\mathcal{C}_1, \dots, \mathcal{C}_n)$, with head of $\mathcal{C}_i$ being $(\text{legal } p_i \ l_i)$, and the `legal` arguments tuple being more general than the joint move tuple, i.e. $m_i \leq l_i$ for each $i = 1, \dots, n$.

The move transition is computed from the `next` clauses whose all `does` relations are matched by respective joint move tuple elements as follows.

Definition 11. Let $\mathcal{N}$ be a `next` clause. The $\mathcal{N}$ *does facts*, $d_1(\mathcal{N}), \dots, d_n(\mathcal{N})$, are terms, one for each player, constructed from $\mathcal{N}$ in the following way. Let $(\text{does } p_i \ d_i^j)$ be all `does facts` in $\mathcal{N}$.

- If there is exactly one d_i for player p_i we set $d_i(\mathcal{N}) = d_i$.
- If there is no `does fact` for player p_i in $\mathcal{N}$ we set $d_i(\mathcal{N})$ to a fresh variable.
- If there are multiple $d_i^1, \dots, d_i^k$ for player p_i we compute $\sigma = \text{MGU}(d_i^1, \dots, d_i^k)$ and set $d_i(\mathcal{N}) = \sigma(d_i^1)$.

We have $m_i \leq d_i(\mathcal{N})$ for each `next` clause $\mathcal{N}$ contributing to the move transition, since otherwise the body of $\mathcal{N}$ would not match the state enhanced with $(\text{does } p_i \ m_i)$ facts.

Example 12. In the Tic-tac-toe example, there are three clauses where the control player is o , which after renaming of variables look as follows.

$$\mathcal{N}_1 = (\leq (\text{next} (\text{control } x)) (\text{does } x \ \text{noop})),$$

$$\mathcal{N}_2 = (\leq (\text{next} (\text{cell } ?x2 \ ?y2 \ o)) (\text{does } o (\text{mark } ?x2 \ ?y2))),$$

$$\mathcal{N}_3 = (\leq (\text{next} (\text{cell } ?x3 \ ?y3 \ ?c)) (\text{true} (\text{cell } ?x3 \ ?y3 \ ?c)) (\text{does } o (\text{mark } ?x1 \ ?y1)) (\text{or} (\text{distinct } ?x3 \ ?x1) (\text{distinct } ?y3 \ ?y1))).$$

The `does facts` are, respectively,

$$\begin{aligned} d_1(\mathcal{N}_1) &= \text{noop} & \text{and} & & d_2(\mathcal{N}_1) &= x_{f_1}, \\ d_1(\mathcal{N}_2) &= x_{f_2} & \text{and} & & d_2(\mathcal{N}_2) &= (\text{mark } x_2 \ y_2), \\ d_1(\mathcal{N}_3) &= x_{f_3} & \text{and} & & d_2(\mathcal{N}_3) &= (\text{mark } x_1 \ y_1). \end{aligned}$$

Each rewrite rule of the translated game is generated from a tuple of legal clauses $\mathcal{C}_1, \dots, \mathcal{C}_n$ and a selection of `next` clauses $\mathcal{N}_1, \dots, \mathcal{N}_m$, with variables renamed so that no variable occurs in multiple clauses, and such that

$$l_i \doteq d_i(\mathcal{N}_1) \doteq \dots \doteq d_i(\mathcal{N}_m)$$

for each player p_i . We will consider all tuples $\overline{\mathcal{C}}, \overline{\mathcal{N}}$ for which the above MGU exists and we will denote it by $\sigma_{\overline{\mathcal{C}}, \overline{\mathcal{N}}}$. We apply $\sigma_{\overline{\mathcal{C}}, \overline{\mathcal{N}}}$ to the clauses and we will refer to the result simply as *the legal and next clauses of the rule*.

Technically, for completeness, we need to generate a rule for a set of `next` clauses even if we generate a rule for its superset, and then for correctness, we need to preclude application of the more general rule when the more concrete rule is applicable, adding `distinct` conditions to clauses of the otherwise more general rule. In the current implementation, we only consider maximal sets of `next` clauses.

Example 13. Let $\mathcal{C}_1 = \text{noop}$ and $\mathcal{C}_2 = (\text{mark } x \ y)$. The clauses $\mathcal{N}_1, \mathcal{N}_2, \mathcal{N}_3$ introduced above form a maximal set,

$$\begin{aligned} \sigma_{\overline{\mathcal{C}}, \overline{\mathcal{N}}} &= \{x_{f_1} \mapsto (\text{mark } x \ y), \quad x_{f_2} \mapsto \text{noop}, \\ &\quad x_2 \mapsto x, \quad y_2 \mapsto y, \quad x_1 \mapsto x, \quad y_1 \mapsto y\}. \end{aligned}$$

With all tuples $\overline{\mathcal{C}}, \overline{\mathcal{N}}$ selected and the MGU $\sigma_{\overline{\mathcal{C}}, \overline{\mathcal{N}}}$ computed, we are almost ready to construct the rewriting rules. Still, for a fixed tuple $\overline{\mathcal{C}}, \overline{\mathcal{N}}$, we first need to compute erasure clauses to prevent constructing too general rules in the end.

Erasure Clauses So far, we have not accounted for the fact that rewrite rules of Toss only affect the matched part of the structure, while the GDL game definition explicitly describes the construction of the whole successive structure. We will say that a `next` clause is a *frame clause* if and only if it contains a `true` relation applied to a term equal to the `next` argument. Negating the frame clauses from the tuple $\overline{\mathcal{N}}$ and transforming them into *erasure clauses* will keep track of the elements that possibly lose fluents and ensure correct translation.

From the frame clauses in $\sigma_{\overline{\mathcal{C}},\overline{\mathcal{N}}}(\mathcal{N}_1), \dots, \sigma_{\overline{\mathcal{C}},\overline{\mathcal{N}}}(\mathcal{N}_m)$, we select all (maximal) subsets J such that, clauses in J having the form $(\leq (\text{next } s_1) b_1)$, it holds

$$s_1 \doteq_f \dots \doteq_f s_{|J|},$$

i.e. the arguments of `next` unify. Note that we use $\doteq_f$ instead of the standard unification, and by that we mean that the variables shared with the `legal` clauses $\overline{\mathcal{C}}$ are treated as constants. The reason is that these variables are not local to the clauses and must therefore remain intact.

Intuitively, the selected sets J describe a partition of the state terms that could possibly be copied without change by the rule we will generate for $\overline{\mathcal{C}},\overline{\mathcal{N}}$.

Let us write ρ for the f -MGU of $s_1, \dots, s_{|J|}$. To compute the bodies of the erasure clauses, we negate the disjunction of substituted bodies of the frame clauses and bring this Boolean combination to disjunctive normal form (DNF), i.e. we compute conjunctions $e_1, \dots, e_l$ such that

$$\neg(\rho(b_1) \vee \dots \vee \rho(b_{|J|})) \equiv (e_1 \vee e_2 \dots \vee e_l).$$

As the head of each erasure clause we use $\rho(s_1) = \dots = \rho(s_{|J|})$, with the one technical change that we ignore the fluent paths in this term. We replace these fluent paths with `BLANK` and thus allow them to be deleted in case they are not preserved by other `next` clauses of the rule. Let us denote by h the term $\rho(s_1)$ after the above replacement. The erasure clauses $\mathcal{E}_{\overline{\mathcal{C}},\overline{\mathcal{N}}}(J) = \{(\leq h e_1) \dots (\leq h e_l)\}$, and we write $\mathcal{E}_{\overline{\mathcal{C}},\overline{\mathcal{N}}}$ for the union of all $\mathcal{E}_{\overline{\mathcal{C}},\overline{\mathcal{N}}}(J)$, i.e. for the set of all $\overline{\mathcal{C}},\overline{\mathcal{N}}$ erasure clauses.

Example 14. In our example, $\mathcal{N}_3$ and its counterpart for the other player are the only frame clauses in G . After negation, $\sigma(\mathcal{N}_3)$ splits into several clauses e_i . The relevant one is $(\leq (\text{next } (\text{cell } ?x3 ?y3 ?c)) (?x3 = ?x) (?y3 = ?y))$, i.e. $(\leq (\text{next } (\text{cell } ?x ?y ?c)))$. The resulting erasure clause is $(\leq (\text{next } (\text{cell } ?x ?y \text{ BLANK})))$. If no other clause had the form $(\leq (\text{next } (\text{cell } ?x ?y \dots)))$, this clause would cause the erasure of any fluent at coordinates (x, y) . Other erasure clauses derived from $\sigma(\mathcal{N}_3)$ turn out to be contradictory with remaining clauses, and thus will not contribute to any rewrite rule in the translation, due to filtering described below.

Rewriting Rule Creation For each suitable tuple $\overline{\mathcal{C}},\overline{\mathcal{N}}$ we have now created the unifier $\sigma_{\overline{\mathcal{C}},\overline{\mathcal{N}}}$ and computed the erasure clauses $\mathcal{E}_{\overline{\mathcal{C}},\overline{\mathcal{N}}}$. At this point, clauses $\overline{\mathcal{C}},\overline{\mathcal{N}}$ are optionally divided according to the player of the `does` relation atom in

them. To create the rules, we need to further partition the *rule clauses* $\sigma_{\overline{\mathcal{C}},\overline{\mathcal{N}}}(\mathcal{C}_i), \sigma_{\overline{\mathcal{C}},\overline{\mathcal{N}}}(\mathcal{N}_i)$ and $\mathcal{E}_{\overline{\mathcal{C}},\overline{\mathcal{N}}}$, and augment them with further conditions. The reason is that the prepared rule clauses may have different matches in different game states, while the Toss rule has to be built from all the rule clauses that would match when the Toss rule matches. Therefore, we need to build a Toss rule for each subset of rule clauses that are “selected” by some game state (i.e. are exactly the rule clauses matching in that state), but add to it separation conditions that prevent the Toss rule from matching in game states where more rule clauses can match.

We select groups of atoms (collected from rule clauses) that separate rule clauses, and generate a Toss rule candidate for every partition of the groups into true and false ones: we collect the rule clauses that agree with the given partition. The selected atoms, some negated according to the partition, form the separation condition.

For each candidate, we will now construct the Toss rule in two steps. In the first step we generate the *matching condition*: we translate the conjunction of the bodies of rule clauses and the separation condition. This translation follows the definitions of atomic relations presented in Section , but we skip the full definition here.

Later we *filter* the rule candidates by checking for satisfiability in the initial structure of the stable part of the matching condition.

In the second step, we build a Toss rewrite rule itself. From the heads of rule clauses of a rule candidate, we build the $\mathfrak{R}$ -structure: each `next` term, with its fluent paths replaced by `BLANK`, is an $\mathfrak{R}$ element, and the fluent predicates holding for the `next` state terms are the relations of $\mathfrak{R}$. The $\mathfrak{L}$ -structure and the precondition of the Toss rule is built from the matching condition, based on elements of $\mathfrak{R}$. Quantification over variables corresponding to $\mathfrak{R}$ elements (which are the same as $\mathfrak{L}$ elements) is dropped, and atoms involving only these variables and not occurring inside disjunctions are extracted to be relations tuples in $\mathfrak{L}$.

Having constructed and filtered the rewriting rule candidates, we have almost completed the definition of $T(G)$. Payoff formulas are derived by instantiating variables standing for the goal values. The formulas defining the terminal condition and specific goal value conditions are translated as mentioned before, from disjunctions of bodies of their respective clauses.

Translating Moves between Toss and GDL

To play as a GDL client, we need to translate legal moves from G into Toss rule embeddings for $T(G)$, and conversely, the rule embeddings from $T(G)$ into moves of G .

In the incoming move case, we augment the Toss rewrite rules with constraints provided in the incoming move, try to embed each of the augmented rules, and return the single rule that matches and its unique embedding. Augmenting the rule is done in the following simple way: If the head of a `legal` clause of the rule contains a variable v at path q , a Toss variable x was derived from a game state term t such that $t \downarrow_p = v$, and the incoming move has term s at path q , then we add $\text{Anch}_p^s(x)$ to the precondition.

To translate the outgoing move, we recall the heads of the `legal` clauses of the rule that is selected, as we only need to substitute all their variables. To eliminate a variable v contained in the head of a `legal` clause of the rule, we look at the rule embedding; if $x \mapsto a$, x was derived from a game state term t such that $t \downarrow_p = v$, and $a \downarrow_p^m = s$, then we substitute v by s . The move translation function μ is thus constructed.

Game Simplification in Toss

Games automatically translated from GDL, as described above, are verbose compared to games defined manually for Toss. They are also inefficient, since the current solver in Toss works fast only for sparse relations.

Both problems are remedied by joining co-occurring relations. Relations which always occur together in a conjunction are replaced by their join when they are over the same tuple. Analogically, we eliminate pairs of atoms when the arguments of one relation are reversed arguments of the other.

In an additional simplification, we remove an atom of a stable relation which is included in, or which includes, another relation, when an atom of the other relation already states a stronger fact. For example, if $Positive \subseteq Number$, then $Positive(x) \wedge Number(x)$ simplifies to $Positive(x)$, and $Positive(x) \vee Number(x)$ simplifies to $Number(x)$.

The above simplifications can be applied to any Toss definition. We perform one more simplification targeted specifically at translated games: We eliminate $Eq_{p,q}(x, y)$ atoms when we detect that $Eq_{p,q}$ -equivalence of x and y can be deduced from the remaining parts of the formula.

The described simplifications are stated in terms of manipulating formulas; besides formulas, we also apply analogous simplifications to the structures of the Toss game: the initial game state structure, and each $\mathcal{L}$ and $\mathcal{R}$ rule structures.

Experimental Results

The presented translation was implemented in (Toss), an open-source program with various logic functions as well as the presented game model and a GUI for the players.

After the simplification step described above, the translated games were very similar to the ones we defined manually in Toss. As far as a game could be translated (see the restrictions below), the resulting specification was almost as good for playing as the manual one. In Table 1 we show the results of playing a game translated from GDL against a manually specified one — the differences are negligible.

	Manual Wins	Translated Wins	Tie
Breakthrough	55%	45%	0%
Connect5	0%	0%	100%

Table 1: Manual against Translated on 2 sample games.

Discussion and Perspectives

The major restriction of the proposed translation scheme is its notion of fluent paths. It requires that the `next` clauses

build terms locally, referring among others to a similar original `true` term, instead of propagating changes only from different terms. Consider an example from a specification of the game Connect4:

```
(<= (next (cell ?c ?y2 b)) (succ ?y1 ?y2)
    (true (cell ?c ?y1 b)) (distinct ?y1 6))
```

Here, without further modifications, $(\text{cell}, 2)$ is incorrectly detected as a fluent path. A sufficient condition to meet this restriction is that fluent paths $d\mathcal{P}(s_C, t_C)$ point only to constant terms in s_C . A possible workaround for games violating this condition is guessing that the offending clause is a frame clause and ignoring it, as is the case above.

Even if a game meets the major restriction above, the translation may not handle it or may take a long time. The reason is that it does not use Toss defined relations: it either translates GDL definition relations extensively as stable relations in the game state structure, or it expands (inlines) them before translation. To lift this restriction, we need to use definitions in the translation in an efficient way and translate relations depending on game state as defined relations.

Finally, the number of elements in the resulting game state structure can be prohibitive. It is a consequence of mapping state terms to Toss elements in such way that all fluents are predicates. Currently, an element a is identified with a mask m and a ground substitution for its $\mathcal{P}_m$ paths. If the number of elements falling under a mask m is too large, we would instead like to identify elements by the mask plus ground substitution of a subset of its $\mathcal{P}_m$ paths (for subsets that together cover all its $\mathcal{P}_m$ paths). Fluents corresponding to $\mathcal{P}_f$ paths in m would then be of arity equal to the number of such subsets. We would then define $\llbracket \cdot \rrbracket : A \rightarrow 2^S$ in terms of masks and subsets of $\mathcal{P}_m$, which is closer to the implementation, rather than in terms of $\mathcal{P}_f$ as is done in this work.

References

- Comon, H.; Dauchet, M.; Gilleron, R.; Löding, C.; Jacquemard, F.; Lugiez, D.; Tison, S.; and Tommasi, M. October 2007. Tree automata techniques and applications.
- Genesereth, M. R., and Love, N. 2005. General game playing: Overview of the AAAI competition. *AI Magazine* 26(2):62–72.
- Grädel, E., and Gurevich, Y. 1998. Metafinite model theory. *Information and Computation* 140:26–81.
- Grädel, E. 2007. Finite model theory and descriptive complexity. In *Finite Model Theory and Its Applications*. Springer. 125–230.
- Kaiser, L., and Stafiniak, L. 2011. First-order logic with counting for general game playing. In *Proc. of AAAI'11*.
- Love, N.; Hinrichs, T.; Haley, D.; Schkufza, E.; and Genesereth, M. 2008. General game playing: Game description language specification. Technical report.
- Rajlich, V. 1973. Relational structures and dynamics of certain discrete systems. In *Proc. of MFCS'73*, 285–292.
- Toss. <http://toss.sourceforge.net>.