High Speed Forward Chaining for General Game Playing

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

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

Abstract

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

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

1 Introduction

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

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

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

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

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

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

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

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

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

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

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

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

2 General Game Playing and Forward Chaining

2.1 Game Description Language

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

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

2.2 GDL Rule as Engine

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

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

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

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

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

2.3 Executing a Rule

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

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

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

3 Knowledge Base

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

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

[1] Refer to Definition 4.

[2] Refer to Definition 3.

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

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

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

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

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

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

4 Relations

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

4.1 Rule, Inputs, and Conditions

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

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

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

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

[3]Dictionary of terms converting them to Int32.

4.2 Grounding

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

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

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

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

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

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

```
(⇐  (legal ?0 (mark ?1 ?2))
    (true (cell ?1 ?2 b))
    (true (control ?0)))
```

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

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

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

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

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

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

4.3 Relation ID

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

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

- $\Delta^i_Q \subseteq L$ is the ordered set of ground terms forming the domain of the i^{th} argument of the relation Q.

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

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

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

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

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

5 Processing Rules

5.1 Failing Inputs and Conditions

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

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

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

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

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

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

```
(⇐   (row ?m ?x)
     (true (cell ?m 1 ?x))
     (true (cell ?m 2 ?x))
     (true (cell ?m 3 ?x)))
```

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

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

Example 6. In Connect4 the relation does/4 only ever has one ground instance in the knowledge base, but can have 14 variations, so

P_{exists}((does red (drop 1))) $= 0.0714$.

5.2 Processing Time

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

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

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

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

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

5.3 Optimisation

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

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

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

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

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

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

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

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

- *Specifically, input selection and order matters.*

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

6 Reference Tables

6.1 Reference Tables for Inputs

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

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

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

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

[4]Refer to results in Table 1.

6.2 Reference Tables for Conditions

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

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

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

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

6.3 Reference Table for the Result

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

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

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

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

6.4 Processing a Rule

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

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

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

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

7 Experiments

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

7.1 Experimental Setup

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

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

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

7.2 Performance of the Proposed Approach

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

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

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

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

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

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

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

7.3 Hypothesis Ordering

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

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

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

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

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

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

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

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

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

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

8 Conclusion

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

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

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

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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