This is an extended version of the article published in ECAI'14
Fast Instantiation of GGP Game Descriptions Using Prolog with Tabling.

Efficient Grounding of Game Descriptions with Tabling

Jean-Noël Vittaut and Jean Méhat

LIASD - University of Paris 8, France
jnv@ai.univ-paris8.fr, jm@ai.univ-paris8.fr

Abstract. We present a method to instantiate game descriptions used in General Game Playing with the tabling engine of a Prolog interpreter. Instantiation is a crucial step for speeding up the interpretation of the game descriptions and increasing the playing strength of general players.
Our method allows us to ground almost all of the game descriptions present on the GGP servers in a time that is compatible with the common time settings of the GGP competition. It instantiates descriptions more rapidly than previous published methods.

1 Introduction

General Game Playing (GGP) aims at conceiving programs capable of playing a large variety of games without knowing the rules in advance. The Game Description Language (GDL) [8] has been used to communicate the rules of the game to be played at the beginning of a match in the General Game Playing competition since 2005.

Fast interpretation of GDL is important because it can significantly improve the strength of a player. Björnsson and Schiffel [1] [13] have compared the speed of several GDL reasoners[1] and they show that the reasoners are at least two to three orders of magnitude slower than hard coded versions of games. The two fastest reasoners they tested use a Prolog interpreter.

An approach to speed up a reasoner is to ground the rules, binding all variables with atoms. This instantiation of the rules can lead to better performance because it saves the time used to bind variables during unification and it eases the building of Propositional Nets [4]. Kissmann and Edelkamp [6] have shown that instantiation can allow from about 4 to 250 times more node expansions in a Monte-Carlo search on the tested games. Instantiation is also useful in the domain of action planning [7].

We use the tabling engine built in a Prolog interpreter. Tabling consists in storing answers for subgoals and reusing them whenever the same subgoal is called again. It was first implemented in the XSB programming language [9]. At the cost of a modification of the unification process, it avoids redundant sub-computations and deals with

[1] Flux Player, Cadia Player, Java Eclipse, Java Prover, GGPBase Prover, C++ Reasoner

infinite loops. We use here the tabling as implemented in the YAP Prolog interpreter because of its performance, its availability and our familiarity with this interpreter [10], [11], [12].

This paper is structured as follows: firstly we describe the Game Description Language; then we describe our method of instantiation of GDL programs which makes use of the Prolog tabling engine and we compare the performance of our method against other approaches.

2 The Game Description Language

The Game Description Language (GDL) allows to describe combinatorial perfect information games. It has also been extended to handle incomplete and imperfect information games (GDL-II). It uses first order logic and is similar to Datalog with negation as failure. Its syntax consists of Lisp S-expressions. A game is described with a set of facts and rules; a few keywords are reserved for logic and game-specific features (see table 1); variables begin with a question mark.

Table 1. GDL Keywords

Logical operators	
`<=`	clause declaration
`or`	disjunction
and	implicit in the premisses of a rule
`not`	negation
`distinct`	evaluates to true only if the two terms differ
Static predicates	
`role`	defines the names of the players
`init`	defines initial state of the game
`input`	defines a superset of possible moves
`base`	defines a superset of the game state components
Dynamic predicates	
`terminal`	true if the game state is terminal
`goal`	player's rewards
`legal`	legal moves in the current game state
`next`	transition to the next game state
`does`	player's moves
`true`	defines the game state components
Numerical atoms	
integers from 0 to 100 defined for `goal`	

We distinguish the *dynamic* predicates depending on the state of the game from the remaining *static* ones, the instantiated values of which are independent from the state

of the game and can be computed once and for all upon receiving the game description. GDL missing arithmetic, game descriptions usually contain the description of arithmetic operations on the numbers they need: it usually leads to a large number of static rules.

The predicates `true` and `does` are always dynamic because they trivially depend on the state of the game. The dynamic property is recursively extended to all predicates using at least one dynamic predicate even if it appears within a negation or a disjunction.

The other predicates mentioned as dynamic in table 1 are marked as such to keep them in the grounded rules even in the rare cases where they are static e.g. when the goal of a player is independent from the final position of the game.

By extension, a term is dynamic or static, depending on the predicate it is formed on. Note that terms of the form `(or` T_1 $T_2 \dots T_n$`)` or `(not` T_1`)` are: dynamic if they contain at least one dynamic term T_i; static if all of the T_i are static terms. Likewise a rule is dynamic if its conclusion is dynamic.

3 Instantiation of GDL rules

The instantiation of GDL rules is done in successive steps. We firstly present an overview of the whole process and then give some details on these steps and justify their usefulness.

3.1 Overview

The instantiation starts with a cleaning step where `or` is removed from any rule. Then we compute `base` and `input` facts if they are not provided in the description.

Next we rewrite each rule, removing negated terms, renaming `true` and `does` as `base` and `input` and adding a side effect. This side effect stores, in the instantiated game description, a grounded version of the initial rule with static terms removed.

In the Prolog engine we use, tabling is enabled at the predicate level. Then, to force the tabling engine to process each and every rule, we change the predicate name of every rule conclusion so we can use tabling at the rule level. To keep our program correct, we add a new rule the role of which is allowing to call the new predicate with its old name.

Then we add a series of rules the conclusion of which is always the `ground` atom and premises are queries asking for all the answers related to the new predicate introduced previously.

Finally, we call the Prolog interpreter with the `ground` goal. The grounded description is stored into a data structure shared between Prolog and the driver program by the Prolog interpreter solving the `ground` goal. This instantiated description is only made of grounded terms, called fluents, and logic connectors.

3.2 Eliminating `(or` T_1 $T_2 \dots T_n$`)` terms

The `or` operator has been deprecated in GDL since 2007 [3]. However, it is used in old game descriptions and players need to support it to play these games. As it is easy to rewrite a game description into a game description without `or` or use the built in

Prolog *or* operator, most players support this feature. Even in the 2013 official GGP competition, the game description of Eight Puzzle used it.

Removing any instance of the `or` operator ensures that we obtain at the end a grounded program in a disjunctive form. It also simplifies the next steps in case there is a disjunction between static and dynamic terms.

If a rule contains a term T of the form `(or` T_1 $T_2 \dots T_n$`)`, we simply duplicate the rule replacing T with T_i. We proceed recursively on the new rules which could still contain some term using `or`. Table 2 shows an example of this transformation from the Connect Five game description.

Table 2. A rule containing an `or` with 4 sub-terms is rewritten as 4 different rules.

Rule with `or`	Rules without `or`

`(<= (conn5 ?r)`
`  (or (col ?r) (row ?r)`
`  (diag1 ?r) (diag2 ?r)))`
	`(<= (conn5 ?r) (col ?r))`
	`(<= (conn5 ?r) (row ?r))`
	`(<= (conn5 ?r) (diag1 ?r))`
	`(<= (conn5 ?r) (diag2 ?r))`

3.3 Adding `input` and `base` predicates

The `base` predicate is used to enumerate all the terms that can be used in any reachable game state. Similarly, `input` allows one to pre-compute all the moves that can become legal in the course of any match of the described game. These predicates are a recent addition to GDL and we suppose they were introduced to facilitate the instantiation of game descriptions.

However, older GDL game descriptions do not provide `input` and `base` but are still in use on the servers running permanent tournaments that we use as a test bed for GGP competitions. The set of game descriptions including these predicates is small and does not contain many games that are commonly used for testing and performance comparison purposes.

Moreover, different descriptions of these predicates can lead to dramatic differences between grounded game descriptions, for instance the `input` predicate of the Breakthrough game description is defined more lazily on the Tiltyard server than on the Stanford server. It leads to a grounded description that contains 20 times more rules.

For these reasons, we do compute them when the game description does not provide them. Separating their computation from the strictly speaking grounding phase allows us to distinguish their respective computation times. It also allows us to discard undesirable rational tree terms which are infinite terms that this method can generate.

This step is detailed in section 4 in which we propose a method sharing many steps with the instantiation method we are currently describing.

Once computed, the `input` and `base` fluents are added to the description so there will be no difference with the case where the predicates are provided with the GDL description.

3.4 Eliminating `not`, renaming `true` and `does`

From each rule R, we construct a new rule $g(R)$ by removing every `(not` T`)` term where T is a dynamic term; renaming every `(true` T`)` and `(does` T_1 T_2`)` term respectively with `(base` T`)` and `(input` T_1 T_2`)`.

Removing the `not` operator in dynamic predicates allows us to compute any possible instantiation without risking an elimination by the negated term. It is a safe operation since GDL guarantees that any negated term always has to be fully instantiated. Consequently, the elimination cannot lead to a situation where one of the variables remains not instantiated. It is also necessary since the tabling engine we use cannot handle recursion through a negation. A drawback is that the process will produce useless grounded rules, since the `not` operator is never checked: these useless rules would never prove anything when used by a reasoner working with the instantiated description.

Replacing `(true` T`)` and `(does` T_1 T_2`)` by `(base` T`)` and `(input` T_1 T_2`)` allows us to ground all the rules in one pass, without computing `base` and `input` if they are already provided by the description.

Table 3 contains an example of this step on some rules of the Connect Five game description.

Table 3. Computation of $g(R)$: the negations are eliminated and `does` are replaced by `input`

R	$g(R)$
`(<= (goal x 50)` `  (not line_of_5))`	`(<= (goal x 50))`

`(<= (legal ?r noop)`
`  (role ?r)`
`  (not (true (ctrl ?r))))`
	`(<= (legal ?r noop)`
	`  (role ?r))`

`(<= (next (cell ?x ?y ?r))` `  (does ?r (mark ?x ?y)))`	`(<= (next (cell ?x ?y ?r))` `  (input ?r mark ?x ?y))`

3.5 Removing static terms

Terms formed on static predicates do not need to appear in the instantiated rules since their truth is known regardless of the state of the game: if true they can be removed; if false the entire rule can be discarded; conversely if they appear within a `not`, they can be removed if false and the rule can be discarded if true.

Consequently, we compute a rule $s(R)$ from the initial rule R by removing any static term or its negation from R.

This step could be skipped with no effect on the correctness of the method but without it, we would have to either post-process the instantiated rules to eliminate any true static term or include all of the static terms from the game description.

3.6 Adding the side effect and introducing a new symbol

The rules $g(R)$ and $s(R)$ are combined to produce the two new rules that will be part of our final grounded description. Given a rule $g(R)$ of the form:

$$(\texttt{<=}\ \ (p\ \ U_1 \ldots U_p)\ \ \ T_1 \ldots T_n)$$

we derive the two new rules:

$$(\texttt{<=}\ \ (p\# \ \ U_1 \ldots U_p)\ \ \ T_1 \ldots T_n\ \ \ (\texttt{store}\ \ s(R)))$$
$$(\texttt{<=}\ \ (p\ \ U_1 \ldots U_p)\ \ \ (p\# \ \ U_1 \ldots U_p))$$

where p is the original predicate symbol of the conclusion of $g(R)$. The `store` predicate has the side effect of storing the $s(R)$ instantiated rule in a data structure shared between the Prolog interpreter and the driver program; it always evaluates as true. $p\#$ is a new unique symbol, different for each processed rule. It is necessary to prevent the tabling engine from tabling rules with side effects because it would lead to missed instantiations: the rule including the side effect is not tabled while the second is.

These two rules are logically equivalent to the rule $g(R)$ since the side effect always evaluates as true.

An example of this step on rules from the Connect Five game description is shown in table 4

Table 4. Each original rule is transformed into two new rules: one with a new conclusion symbol and a side effect; the other with the original conclusion

Initial rule	Derived rules

`(<= (goal x 50)`
`  (not line_of_5))`
	`(<= (goal# x 50)`
	`  (store (<= (goal x 50)`
	`  (not line_of_5))))`
	`(<= (goal x 50)`
	`  (goal# x 50))`

`(<= (legal ?r noop)`
`  (role ?r)`
`  (not (true (ctrl ?r))))`
	`(<= (legal# ?r noop)`
	`  (role ?r)`
	`  (store (<= (legal ?r noop)`
	`  (not (true (ctrl ?r))))`
	`(<= (legal ?r noop)`
	`  (legal# ?r noop))`

`(<= (next (cell ?x ?y ?r))`
`  (does ?r (mark ?x ?y)))`
	`(<= (next# (cell ?x ?y ?r))`
	`  (input ?r (mark ?x ?y))`
	`  (store (<= (next (cell ?x ?y ?r))`
	`  (does ?r (mark ?x ?y)))))`
	`(<= (next (cell ?x ?y ?r))`
	`  (next# (cell ?x ?y ?r)))`

3.7 Tabling predicates and creating the instantiation query

Finally, to generate all instantiations in one Prolog query, we add a new predicate `ground`, the goal of which is to query all the rules with side effects. Therefore, a rule like

$$\texttt{(<= ground (}p\texttt{\#}\ U_1\ U_2\ \ldots U_n\texttt{))}$$

is added for each new symbol $p\#$ introduced in the previous step.

We set up the Prolog interpreter to table all predicates with the only exception of the new predicate symbols introduced in the previous subsection and the `ground` predicate which does not need to be tabled. By querying all the solutions to the `ground` goal, the `store` predicate inserts instantiated rules into the data structure shared between the Prolog interpreter and the driver program. In table 5 we show the result of the instantiation of one rule of Connect Five.

Table 5. One of the rules of Connect Five is instantiated in two rules

Initial rule	Grounded rules

`(<= (legal ?r noop)`
`  (role ?r)`
`  (not (true (ctrl ?r))))`
	`(<= (legal x noop)`
	`  (not (true (ctrl x))))`
	`(<= (legal o noop)`
	`  (not (true (ctrl o))))`

4 Computing `input` and `base`

Our instantiating method requires we generate the `input` and `base` predicates when not provided in the GDL description. We describe two ways of computing them: an *iterative* method which is equivalent to the one used by Kissmann and Edelkamp [6]; and our method using tabling which can be performed in *one step*.

4.1 Iterative method

We compute two sets B and I that contain all the fluents that can occur in a game state or as a legal move. We initialize B with the facts defined via `init` in the game description; I is initially empty:

$$I = \emptyset$$
$$B = \{\texttt{(true}\ T\texttt{)}\ \text{s.t.}\ \texttt{(init}\ T\texttt{)}\ \text{is true}\}$$

We temporarily redefine the `not` operator as always true. We then iterate, generating legal move fluents, adding the new ones to I and the new game state fluents to B until reaching a fixed point:

$$I = I \cup \{\texttt{(does}\ T_1\ T_2\texttt{)}\ \text{s.t.}\ \texttt{(legal}\ T_1\ T_2\texttt{)}\ \text{is true}\}$$
$$B = B \cup \{\texttt{(true}\ T\texttt{)}\ \text{s.t.}\ \texttt{(next}\ T\texttt{)}\ \text{is true}\}$$

We use tabling for all the predicates at each iteration and flush the tables at every update of I and B.

The performance of this method strongly depends on the number of iterations required to reach the fixed point. It often happens that only a few elements are added to B at each iteration. It is, for instance, the case in many games where the state of the game contains a `step` term which simply counts how many moves have been played to prevent infinite matches (see figure 1).

```
(<= (next (step 1)) (true (step 0)))
(<= (next (step 2)) (true (step 1)))
...
(<= (next (step 100)) (true (step 99)))
```

Fig. 1. A `step` counter is commonly used in many descriptions to prevent infinite matches. This fragment of a game description allows it to increment until a maximum figure of 100.

When this kind of counter is present in the game description, the method must be iterated the number of times that the counter needs to be incremented before reaching its final value.

To alleviate this specific problem, we first use the original GDL description to simulate a fake match from the initial position; we repeatedly compute the next state of the game that can be reached without playing a move. We halt this process when the reached state is empty or when a game state has already been seen. All the fluents that appeared in any game state are used to seed the B set along with the `init` predicates.

More generally, the aforementioned procedure processes any `next` rule not depending on the `does` predicate to compute fluents in order to initialize the B set. It provides an amelioration for many game descriptions however, it would not be difficult to conceive game rules capable of defeating this procedure.

4.2 One step method

We process the original GDL description with the same transformations we detailed in section 3, the only difference being that we do not add the side effect to the rules.

We also enable tabling for the same predicates that we mentioned in section 3.7 and add the three following rules to seed the B set with the initial state and add new fluents to B and new legal moves to I:

```
(<= (base ?x) (init ?x))
(<= (base ?x) (next ?x))
(<= (input ?r ?m) (legal ?r ?m))
```

Then by querying the Prolog interpreter with goals `(base ?x)` and `(input ?r ?m)`, we obtain all the fluents of these predicates enabling the instantiation of the whole game description more efficiently.

5 Exprerimental results

We collected the 246 different game descriptions that were active in February 2014 on the Dresden server[2].

We firstly measured the time necessary to generate `input` and `base` on the vast majority of game descriptions that do not include them using the *iterative* and *one step* methods.

Then we measured the time necessary to instantiate the game description enriched with the `input` and `base` fluents computed in the previous step except for the six game descriptions that already include them: for these, we used the predicates of the original game description. The time measured takes into account the translation of GDL terms from the Prolog interpreter into the driver program representation.

The experiments were run on one core of an Intel Xeon E5-4610 2.40GHz with 520Gb RAM. This amount of memory was more than enough to compute the instantiations. We measured that our method needs about 500Mb to compute one million instantiated rules. We used YAP 6.2.2 Prolog interpreter [2] as a library for our driver program written in C++.

5.1 Computing `input` and `base`

The `input` and `base` were successfully computed for the vast majority of the 240 game descriptions that did not already contain them, with the exception of two games for the *one step* method (`othello_comp2007` and `othellosuicide`) and 13 games for the *iterative* one. The two failures of the *one step* method were caused by a crash in the Prolog interpreter, whereas the *iterative* one was halted after 30 minutes as it had not yet converged. However, at least one method succeeded for each game description.

Figure 2 compares the two methods for the 225 games of the Dresden collection successfully processed by both. The x-axis represents the time used by the *one step* method and the y-axis the time used by the *iterative* one. The diagonal represents the location where the two methods take the same amount of time. A game plotted above the diagonal means that the *one step* method takes less time than the *iterative* one. Except for the 10 games plotted below the diagonal, the *one step* method is always faster than the *iterative* one. We also observed that the iterative method is only competitive when the number of iterations remains low.

In figure 3 we plotted the percentage of games of which `input` and `base` fluents have been computed in less than the time budget represented in the x-axis with a logarithmic scale. It shows that a large majority of computations take less than one second.

With the *one step* method, 45% of the games had the fluents computed in less than 100ms, 84% in less than one second and 98% in less than one minute. With the *iterative* method, none of the games had the fluents computed in less than 100ms, 75% in less than one second and 90% in less than one minute.

[2] The Dresden server is available at http://ggpserver.general-game-playing.de

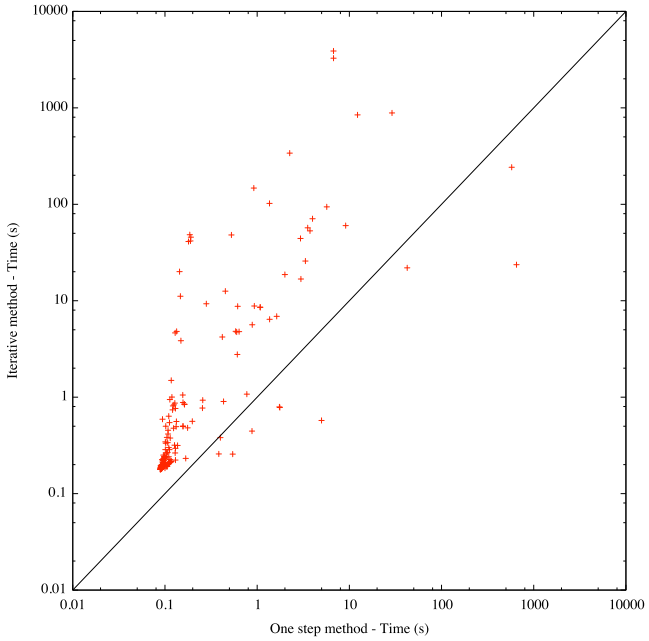

Fig. 2. Comparison of the computation of the `input` and `base` predicates between the *one step* and the *iterative* method.

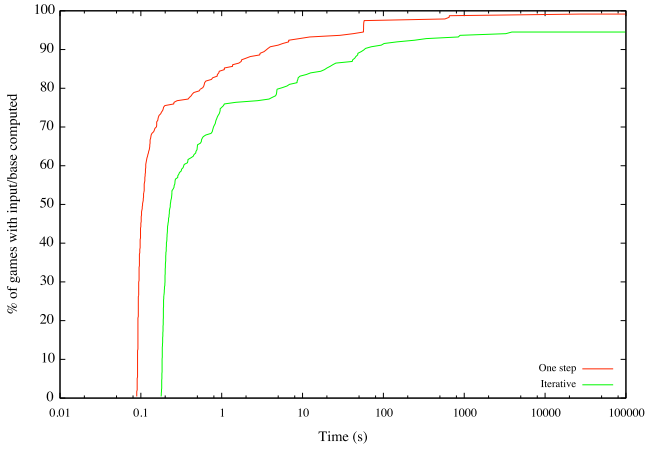

Fig. 3. Percentage of game descriptions of which `input` and `base` can be computed within the time budget on x-axis.

5.2 Instantiation of the rules

We tested the instantiation of the rules on all of the 246 games of the Dresden collection. Six of them already contained the `input` and `base` predicates. For the remaining 240, we added the fluents computed either by the *one step* or the *iterative* method. The processing of three games (`racer`, `ruledepthquadratic` and `laikLee_hex`) was halted after 30 minutes of computation.

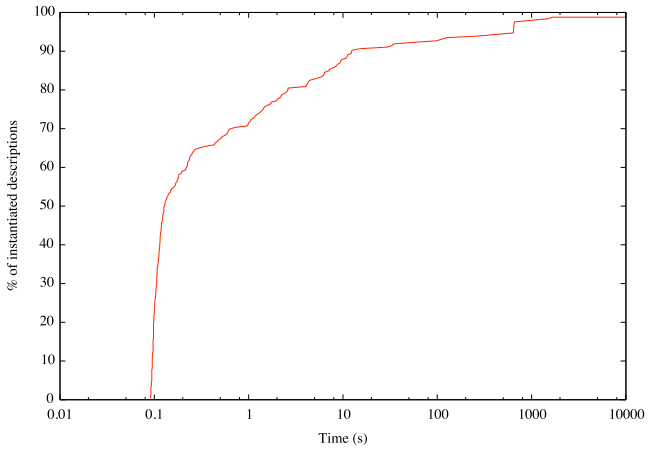

Fig. 4. Percentage of instantiated game descriptions that were grounded within the time budget in the x-axis.

In figure 4 we plotted the time performance of our grounding method where the time of the step computing `input` and `base` is not taken into account. We represented the percentage of games that can be instantiated within the time budget represented in the x-axis. 24% of the games were instantiated in less than 100ms, 72% in less than one second and 94% in less than one minute.

The remaining 6% that were grounded in more than one minute are `battlebrushes`, `merrills`, `amazons`, `racer4`, `farmers`, the two instances of `battlesnakes`, and 8 of the 13 instances of `vacuumcleaner`. All of these game instantiated descriptions contained more than 10^7 rules and facts.

Figure 5 demonstrates that the time to ground increases almost linearly with the size of the grounded game description when it is greater than 10^4. We also observed that a significant part of the time is used to translate the fluents from the Prolog internal representation into the GDL representation in the shared data structure.

6 Comparison with other works

It is somewhat difficult to compare our method with existing grounders, given that their measure of performance is usually mixed with the time used for building the Proposi-

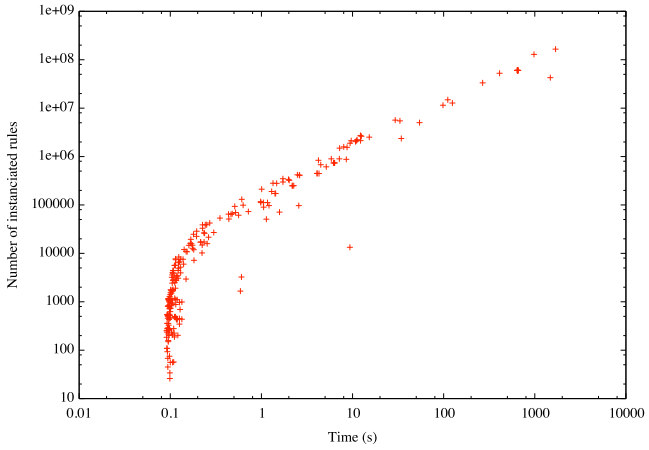

Fig. 5. The number of generated rules as a function of instantiation time for the 243 successfully instantiated games.

tional Net. We examine here the results available from [6] and the time we measured with the *GGPBase flattener*.

6.1 The *GGPBase flattener*

The *GGPBase flattener* is a freely distributed GDL grounder[3]. We compared the time to instantiate a few game descriptions that were of increasing difficulty for our grounder. We used a different machine that was more convenient to run the GGPBase flattener. The results are presented in table 6. The time needed by the flattener seems to increase at least quadratically with the size of the grounded program whereas the time needed by our method has been established to increasing linearly.

The GGPBase flattener has primarily an educational purpose and its performance is not its main goal.

Table 6. Comparison of our method with GGPBase-flattener on an Intel Core 2 Duo 1.86GHz with 2Gb RAM.

	Time to instantiate (seconds)	
Game	GGPBase	Our method
`connectfour`	0.844	0.560
`CephalopodMicro`	19.8	1.14
`breakthrough`	115	5.46
`chinesecheckers4`	Out of memory	14.9

[3] The set of GGPBase Java libraries is distributed at http://www.ggp.org/developers/players.html

6.2 The Kissmann and Edelkamp approach

Kissmann and Edelkamp presented two approaches of grounding in [6]. They were able to instantiate 96 of 171 game descriptions in less than one minute with their Prolog-based approach, and 90 of 171 with their method using dependency graphs which are proportions that we attain in less than one second.

Their article lacks precise figures but similar results are presented in Kissmann PhD thesis for the 124 game descriptions their method was able to successfully process [5]. Their experiments were carried out on an Intel i7-920 2.67GHz with 24Gb RAM with a different Prolog interpreter (SWI-Prolog). They method also includes a computation of mutually exclusive fluents.

The comparison of the percentage of game descriptions instantiated within a computational budget is given in figure 6 for the best result of their two approaches and our method applied to the same 124 game descriptions. We observe that our implementation needs a setup time of approximtely 0.2s. Our method appears to be of two orders of magnitude faster when the instantiation time becomes significant.

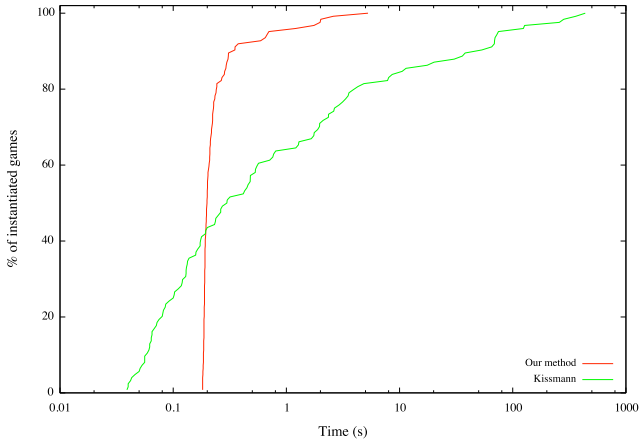

Fig. 6. Comparison of the percentages of instantiated game descriptions that were grounded within the time budget in the x-axis for the 124 game descriptions successfully processed in [5, p. 129–130].

7 Conclusion

We have demonstrated that it is possible to ground almost all the game descriptions found on the GGP servers in a time span compatible with the current GGP competition time settings. This relies on the use of tabling in a Prolog interpreter.

This result should be considered in relation to the study of [1] and [13] in which Prolog-based GDL reasoners greatly outperform other approaches: the Prolog interpreters benefit from decades of optimization from their maintainers.

We have also established that the new predicates `input` and `base` introduced in the 2013 competition, probably with the aim of helping programs to ground game descriptions and generate Propositional Nets, can be considered as superfluous. The few game descriptions in which they can be useful have a grounded size that is so large that building alternative representations such as Propositional Nets is problematic. Tweaking the Game Description Language for specific tasks is somewhat dubious since the description language should be as agnostic as possible in relation to methods that could be used by players.

As a future work, we are interested in finding mutually exclusive terms and rules in the game description that could lead to more concise instantiations and facilitate the generation of Propositional Nets.

References

1. Björnsson, Y., Schiffel, S.: Comparison of GDL reasoners. In: Proceedings of the IJCAI-13 Workshop on General Game Playing (GIGA'13) (2013)
2. Costa, V.S., Rocha, R., Damas, L.: The YAP prolog system. Theory and Practice of Logic Programming 12(1-2), 5–34 (2012)
3. Finnsson, H.: CADIA-Player: A General Game Playing Agent. Master's thesis, Reykjavk University - School of Computer Science (2007)
4. Genesereth, M., Thielscher, M.: General Game Playing (2013), available at http://logic.stanford.edu/ggp/chapters/cover.html
5. Kissmann, P.: Symbolic search in planning and general game playing. Ph.D. thesis, Universität Bremen (2012)
6. Kissmann, P., Edelkamp, S.: Instantiating general games using prolog or dependency graphs. In: Dillmann, R., Beyerer, J., Hanebeck, U., Schultz, T. (eds.) KI 2010: Advances in Artificial Intelligence, Lecture Notes in Computer Science, vol. 6359, pp. 255–262. Springer Berlin Heidelberg (2010)
7. Koehler, J., Hoffmann, J.: Handling of inertia in a planning system. Tech. rep. (1999)
8. Love, N., Hinrichs, T., Haley, D., Schkufza, E., Genesereth, M.: General game playing: Game description language specification. Tech. rep. (2008), most recent version should be available at http://games.stanford.edu/
9. Ramakrishnan, I., Rao, P., Sagonas, K., Swift, T., Warren, D.S.: Efficient access mechanisms for tabled logic programs. The Journal of Logic Programming 38(1), 31–54 (1999)
10. Rocha, R., Silva, F., Costa, V.S.: A tabling engine for the YAP prolog system. In: Proceedings of the 2000 APPIA-GULP-PRODE Joint Conference on Declarative Programming (AGP 2000), La Habana, Cuba (December 2000) (2000)
11. Rocha, R., Silva, F., Costa, V.S.: Dynamic mixed-strategy evaluation of tabled logic programs. In: Logic Programming, pp. 250–264. Springer (2005)
12. Rocha, R., Silva, F., Santos Costa, V.: Yaptab: A tabling engine designed to support parallelism. In: Conference on Tabulation in Parsing and Deduction. pp. 77–87 (2000)
13. Schiffel, S., Björnsson, Y.: Efficiency of gdl reasoners. Computational Intelligence and AI in Games, IEEE Transactions on PP(99), 1–1 (2014)