ECAI 2014
T. Schaub et al. (Eds.)

doi:10.3233/978-1-61499-419-0-1121

Fast Instantiation of GGP Game Descriptions Using Prolog with Tabling

Jean-Noël Vittaut [1] and **Jean Méhat** [2]

Abstract. We present a method to instantiate game descriptions used in General Game Playing with a Prolog interpreter using tabling. Instantiation is a crucial step for speeding up the interpretation of 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 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) [3] 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. 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 [3].

Fast interpretation of GDL is important because it can significantly improve the strength of a player. An approach to speed up a reasoner is to ground the rules and build a Propositional Net [1].

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. At the cost of a modification of the unification process, it avoids redundant sub-computations and deals with 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 [4], [5].

2 INSTANTIATION OF GDL RULES

Cleaning step The instantiation starts with a cleaning step where the explicit `or` operator, which has been deprecated in GDL since 2007, is removed by rewriting all the rules into a disjuctive form.

Adding `input` and `base` predicates The `input` and `base` predicates are used to enumerate all the moves and facts that may be used in any reachable game state. We add them to the game descriptions when they are missing.

These predicates are a recent addition to GDL and we suppose they were introduced to facilitate the instantiation of game descriptions. Actually, 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.

We compute them using either an *iterative method* equivalent to the one used by Kissmann et al. [2] or a *one step method* equivalent to the one we are describing for the instantiation. The two methods perform similarly with a slight advantage for the second one that uses tabling.

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

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 `not` 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. A drawback is that the process will produce useless grounded rules, since the `not` operator is never checked. This transformation allows us to ground all the rules in one pass.

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.

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 grounding program. Given a rule $g(R)$ of the form `(<=` `(`p $U_1 \dots U_p$`)` $T_1 \dots T_n$`)` we derive the two new rules:

`(<=` p`#` `(`$U_1 \dots U_p$`)` $T_1 \dots T_n$ `(store` $s(R)$`))`
`(<=` `(`p $U_1 \dots U_p$`)` `(`p`#` $U_1 \dots U_p$`))`

where p is the original predicate symbol of the conclusion of $g(R)$. The `store` predicate has the side effect of storing the $s(R)$ instanti-

[1] LIASD - University of Paris 8, France, email: jnv@ai.univ-paris8.fr
[2] LIASD - University of Paris 8, France, email: jm@ai.univ-paris8.fr

ated 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 $g(R)$.

Instantiation By querying all the solutions to the tabled predicates, the `store` predicate inserts instantiated rules into the shared data structure.

3 EXPERIMENTAL RESULTS

We tested the instantiation of the rules on all of the 246 games that were active in February 2014 on the Dresden server [3]. The experiments were run on one core of an Intel Xeon E5-4610 2.40GHz with 520GB of RAM. We measured that our method needs about 500MB to compute one million instantiated rules. We used YAP 6.2.2 Prolog interpreter as a library for our driver program written in C++.

In figure 1 we plotted the time performance of our grounding method. The time of the step computing `input` and `base` is not taken into account but we measured that it takes less than half the time of the whole instantiation process, demonstrating that our method does not need these predicates to be included in game descriptions. 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 contained more than 10^7 rules and facts. The processing of three games was halted after 30 minutes of computation.

Figure 2 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.

We compared our grounder with the *GGPBase flattener*, a freely distributed GDL grounder[4]. The time needed by the flattener seemed 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.

We also compared our method with the two approaches presented by Kissmann and Edelkamp in [2]. They were able to instantiate about 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. Our method appears to be of two orders of magnitude faster when the instantiation time becomes significant.

4 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.

We have also established that the new predicates `input` and `base` can be considered as superfluous. The few game descriptions in which they can be helpful 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.

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

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

REFERENCES

[1] Michael Genesereth and Michael Thielscher, *General Game Playing*, 2013. Available at http://logic.stanford.edu/ggp/chapters/cover.html.

[2] Peter Kissmann and Stefan Edelkamp, 'Instantiating general games using prolog or dependency graphs', in *KI 2010: Advances in Artificial Intelligence*, eds., Rüdiger Dillmann, Jürgen Beyerer, UweD. Hanebeck, and Tanja Schultz, volume 6359 of *Lecture Notes in Computer Science*, 255–262, Springer Berlin Heidelberg, (2010).

[3] Nathaniel Love, Timothy Hinrichs, David Haley, Eric Schkufza, and Michael Genesereth, 'General game playing: Game description language specification', Technical report, (2008). Most recent version should be available at http://games.stanford.edu/.

[4] Ricardo Rocha, Fernando Silva, and Vıtor Santos Costa, '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).

[5] Ricardo Rocha, Fernando Silva, and Vítor Santos Costa, 'Dynamic mixed-strategy evaluation of tabled logic programs', in *Logic Programming*, 250–264, Springer, (2005).

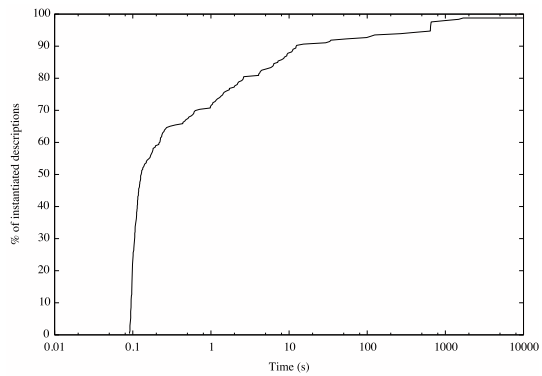

Figure 1. Percentage of instantiated game descriptions that were grounded within the time budget in the x-axis.

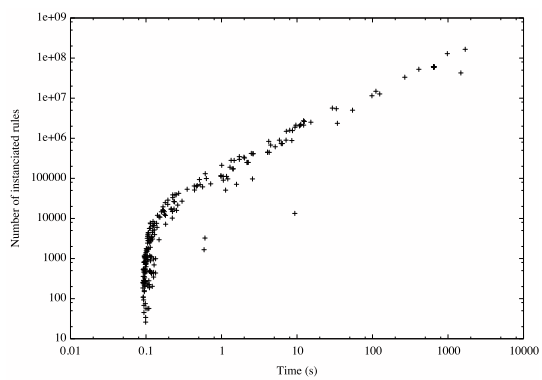

Figure 2. The number of generated rules as a function of instantiation time for the 243 successfully instantiated games.