Grounding GDL Game Descriptions [*]

Stephan Schiffel
School of Computer Science / CADIA
Reykjavik University, Iceland
stephans@ru.is

Abstract

Many state-of-the-art general game playing systems rely on a ground (propositional) representation of the game rules. We propose a theoretically well-founded approach using efficient off-the-shelf systems for grounding game descriptions given in the game description language (GDL).

1 Introduction

Games in General Game Playing are generally described in the game description language (GDL) [Love *et al.*, 2008]. While allowing to describe a large class of games and being theoretically well-founded, reasoning with GDL is generally slow compared to game specific representations [Schiffel and Björnsson, 2014]. This limits the speed of search, both for heuristic search methods, such as Minimax, as well as for simulation-based approaches, such as Monte Carlo Tree Search. Thus, an important aspect of General Game Playing is to find a better representation of the rules of a game that facilitates both fast search in the game tree as well as efficient meta-gaming analysis. Propositional networks [Schkufza *et al.*, 2008] and binary decision diagrams (BDDs) [Edelkamp and Kissmann, 2011] have been proposed for faster reasoning with the game rules. Both approaches, require that game descriptions be grounded, that is, translated into a propositional representation. Other meta-gaming approaches could also benefit from having a propositional description of the game rules as input. Some examples are finding symmetries in games [Schiffel, 2010], discovering heuristics for games (e.g., [Michulke and Schiffel, 2013]), proving game properties [Schiffel and Thielscher, 2009a; Haufe *et al.*, 2012] and factoring games [Cox *et al.*, 2009; Cerexhe *et al.*, 2014].

The GGP-Base framework [Schreiber and Landau, 2016], which is the basis for a number of general game players, contains code for generating a propositional network [Schkufza *et al.*, 2008] representing the game rules. This code requires computing ground instances of all rules in the game description. However, the code seems ad-hoc and it is not obvious whether and for which class of game descriptions it maintains the semantics of the game rules. With this paper, we propose to transform the game description into an answer set program [Baral, 2003] and use the grounder of a state-of-the-art answer set solving system to compute a propositional representation of the game. This has the advantage of using a highly optimized and well-tested system, as well as being theoretically well-founded. Our system also turns out to be able to handle more games than the GGP-Base framework and being significantly faster for many games.

2 Related Work

In [Kissmann and Edelkamp, 2010], the authors report two methods for grounding GDL, one using Prolog and another using dependency graphs. Both methods have some deficiencies and the authors only manage to ground 96 out of the 171 tested games. While the authors do not report on the size of the grounded game descriptions, they report on one of their methods to produce game descriptions that are unnecessarily big.

In [Haufe *et al.*, 2012], The authors use Answer Set Programming (ASP) [Baral, 2003] to prove properties of games by transforming a GDL description into an answer set program and adding constraints that encode the properties to be proven. The system they have implemented uses the Potassco ASP solver [Gebser *et al.*, 2011] which relies on grounding the answer set program, and thus, indirectly the game description.

3 Game Description Language (GDL)

The game description language [Love *et al.*, 2008; Schiffel and Thielscher, 2009b] is a first-order-logic based language that can be seen as an extension of Datalog permitting negations and function symbols. Thus, a game description in GDL is a logic program. The game specific semantics of GDL stems from the use of certain special relations, such as for describing the initial game state (`init`), detecting (`terminal`) and scoring (`goal`) terminal states, for generating legal moves (`legal`) and successor states (`next`). A game state is represented by the set of terms that are true in the state (e.g., `cell(1,1,b)`) and the special relations `true`(f) and `does`(r, m) can be used to refer to the truth of f being in the current state and role r doing move m in the current state

[*]This work was supported by the Icelandic Centre for Research (RANNIS).

```
role(xplayer).
role(oplayer).

init(cell(1, 1, b)).
init(cell(1, 2, b)).
...
init(cell(3, 2, b)).
init(cell(3, 3, b)).
init(control(xplayer)).

legal(W, mark(X, Y)) :-
  true(cell(X, Y, b)),
  true(control(W)).
legal(oplayer, noop) :-
    true(control(xplayer)).
...
next(cell(M, N, x)) :-
  does(xplayer, mark(M, N)),
  true(cell(M, N, b)).
next(control(oplayer)) :-
    true(control(xplayer)).
...
row(M,X) :-
  true(cell(M, 1, X)),
  true(cell(M, 2, X)),
  true(cell(M, 3, X)).
...
line(X) :- row(M, X).
line(X) :- column(M, X).
line(X) :- diagonal(X).
...
goal(xplayer, 100) :- line(x).
goal(xplayer, 0) :- line(o).
...
terminal :- line(x).
```

Figure 1: A partial GDL game description for the game Tictactoe (reserved GDL keywords are marked in bold)

transition. Figure 1 shows a partial GDL description for the game Tictactoe.

GDL allows to describe a wide range of deterministic perfect-information simultaneous-move games with arbitrary number of adversaries. Turn-based games are modeled by having the players that do not have a turn return a move with no effect (e.g., `noop` in Figure 1).

To ensure an unambiguous declarative interpretation, valid GDL descriptions need to fulfill a number of restrictions:

Definition 1. *The* dependency graph *for a set G of clauses is a directed, labeled graph whose nodes are the predicate symbols that occur in G and where there is a* positive *edge* $p \xrightarrow{+} q$ *if G contains a clause* $p(\overline{s}) \Leftarrow \ldots \wedge q(\overline{t}) \wedge \ldots$, *and a* negative *edge* $p \xrightarrow{-} q$ *if G contains a clause* $p(\overline{s}) \Leftarrow \ldots \wedge \neg q(\overline{t}) \wedge \ldots$

To constitute a valid GDL specification, *a set of clauses G and its dependency graph* Γ *must satisfy the following.*

1. *There are no cycles involving a negative edge in* Γ *(this is also known as being* stratified *[Apt* et al.*, 1987; van Gelder, 1989]);*
2. *Each variable in a clause occurs in at least one positive atom in the body (this is also known as being* allowed *[Lloyd and Topor, 1986]);*
3. *If* p *and* q *occur in a cycle in* Γ *and G contains a clause*
$$p(s_1, \ldots, s_m) \Leftarrow b_1(\overline{t}_1) \wedge \ldots \wedge q(v_1, \ldots, v_k) \wedge \ldots \wedge b_n(\overline{t}_n)$$
then for every $i \in \{1, \ldots, k\}$,
   - v_i *is variable-free, or*
   - v_i *is one of* $s_1, \ldots, s_m$, *or*
   - v_i *occurs in some* $\overline{t}_j$ $(1 \leq j \leq n)$ *such that* b_j *does not occur in a cycle with* p *in* Γ.

■

4 Restrictions

As mentioned in [Haufe *et al.*, 2012] (Section 3.2), there is no finite grounding of a GDL description in general. While the restrictions from Definition 1 ensure that reasoning about single states or state transitions is finite, the restrictions are not strong enough to ensure finiteness or decidability of reasoning about the game in general, such as, whether the game will terminate or is winnable for some player.

In fact, without further restrictions, GDL as defined in [Love *et al.*, 2008] or [Schiffel and Thielscher, 2009b] is Turing complete [Saffidine, 2014]. Thus, some restrictions to the language are necessary in order to be able to ground a game description and only game descriptions that adhere to these restrictions can be grounded. The restriction used in [Haufe *et al.*, 2012] and termed *bounded GDL* in [Saffidine, 2014] is the following:

Definition 2. *Let G be a GDL specification. Let G′ be G extended with the following three rules:*

```
    true(F)    :- init(F).
    true(F)    :- next(F).
    does(R,M)  :- legal(R,M).
```

G is in the bounded GDL *fragment of GDL descriptions, if, and only if, G′ satisfies the recursion restriction.* ■

As discussed in [Saffidine, 2014], this restriction makes bounded GDL decidable and therefore truly less expressive than (unbounded) GDL. However, this is of little practical consequence as all of the game descriptions currently available in GDL belong to the bounded fragment.

5 Grounding

To obtain a ground version of a game description, we transform it into an answer set program P, ground P using very optimized grounder for answer set programs and extract the ground version of the game rules from the grounded answer set program.

Specifically, the program P that we create consists of

- the game description itself,
- a state generator,
- an action generator, and
- rules that encode all possible state terms and moves in the game.

The following definitions are based on [Haufe *et al.*, 2012], but simplified for our purpose.

Definition 3. *A* state generator *for a valid GDL specification G is an answer set program P^{gen} such that*

- *The only atoms in P^{gen} are of the form* true(f)*, where $f \in \Sigma$, or auxiliary atoms that do not occur elsewhere; and*
- *for every reachable state S of G, P^{gen} has an answer set $\mathcal{A}$ such that for all $f \in \Sigma$:* true$(f) \in \mathcal{A}$ *iff $f \in S$.*

■

We use the following state generator

```
{true(F):base(F)}.
```

where, intuitively, *base*(f) encodes all possible terms f that might appear in a state of the game.

Definition 4. *Let $A(S)$ denote the set of all legal joint moves in S, that is,*

$$A(S) \overset{\text{def}}{=} \{A : R \mapsto \Sigma | l(r, A(r), S)\}$$

An action generator *for a valid GDL specification G is an answer set program P^{legal} such that*

- *The only atoms in P^{legal} are of the form* does(r, m)*, where $r \in R$ and $m \in \Sigma$, or auxiliary atoms that do not occur elsewhere;*
- *for every reachable (non-terminal) state S of G and every joint move $A \in A(S)$, P^{legal} has an answer set $\mathcal{A}$ such that for all $r \in R$:* does$(r, A(r)) \in \mathcal{A}$*; and*
- *for every reachable (terminal) state $S \in T$ of G, P^{legal} has an answer set.*

■

We use the following action generator

```
1={does(R, M):input(R, M)} :- role(R).
```

where, intuitively, *input*(r, m) encodes all possible moves m of role r in the game. Thus, our action generator does admit answer sets that might not be legal joint moves for a specific state. However, this is not a problem, since we are not interested in the answer sets, but only the grounded answer set program.

For several years, games in the international general game playing competition contain definitions of **base** and **input** predicates as used above. However, there is no formal definition of the semantics of those predicates in GDL and many older game descriptions do not have those predicates. We argue for the following definition:

Definition 5. *A game description G is said to have* well defined base and input definitions *if, and only if,*

- *for every reachable state S of G, for every $f \in S$, $G \vdash$* base(f)*; and*
- *for every reachable state S of G with $S \notin T$, role $r \in R$ and move $m \in \Sigma$, if $l(r, m, S)$ then $G \vdash$* input(r, m)*.*

■

Instead of putting the burden of writing well defined base and input definitions, we propose to generate them from the remaining rules. The idea for this is that the possible instances of *base*(f) should comprise all possible instances of *init*(f) and all possible instances *next*(f) for all possible state transitions. Similarily, the possible instances of *input*(r, m) must contain all instances of *legal*(r, m) in any reachable non-terminal state.

The idea is to compute a *static* version P^{base} of the rules that define next states and legal moves of the players. Here, static means a relaxation of the rules that is independent of *true* and *does*, defined as follows.

Definition 6. *Let G be a GDL specification. We call a predicate p in G* static *iff $p \notin \{\texttt{init}, \texttt{true}, \texttt{next}, \texttt{legal}, \texttt{does}\}$ and p does neither depend on* `true` *nor* `does` *in the dependency graph of G.*

Furthermore, let p^{static} be a predicate symbol which represents a unique name for the static version of predicate p. By definition

$$\begin{aligned}
\texttt{init}^{static} &= \texttt{base}\\
\texttt{true}^{static} &= \texttt{base}\\
\texttt{next}^{static} &= \texttt{base}\\
\texttt{does}^{static} &= \texttt{input}\\
\texttt{legal}^{static} &= \texttt{input}\\
p^{static} &= p, \text{ if } p \text{ is static}
\end{aligned}$$

For each rule $p(\vec{X}) :- \ B \in G$ such that $p \in \{\texttt{init}, \texttt{next}, \texttt{legal}\}$ or either one of `init`, `next`, `legal` *depends on p in the dependency graph of G with positive edges, P^{base} contains the rule*

$$p^{static}(\vec{X}) :- \ B^{static}.$$

where B^{static} comprises the following literals:

$$\{q^{static}(\vec{Y}) : q(\vec{Y}) \in B\} \cup$$
$$\{\texttt{not}\, q(\vec{Y}) : \texttt{not}\, q(\vec{Y}) \in B \wedge q \textit{ is static }\}$$

■

As an example, the following rules form P^{base} as generated for the Tictactoe game (Figure 1):

```
base(cell(1, 1, b)).
...
base(cell(3, 3, b)).
base(control(xplayer)).
base(cell(M, N, x)) :-
    input(xplayer, mark(M, N)),
    base(cell(M, N, b)).
base(cell(M, N, o)) :-
    input(oplayer, mark(M, N)),
    base(cell(M, N, b)).
base(cell(M, N, C)) :-
    base(cell(M, N, C)), C!=b.
base(control(xplayer)) :-
    base(control(oplayer)).
base(control(oplayer)) :-
    base(control(xplayer)).
base(cell(M, N, b)) :-
    base(cell(M, N, b)),
    input(R, mark(X, Y)), M!=X.
```

```
base(cell(M, N, b)) :-
    base(cell(M, N, b)),
    input(R, mark(X, Y)), N!=Y.

input(W, mark(X, Y)) :-
    base(control(W)), base(cell(X, Y, b)).
input(xplayer, noop) :-
    base(control(oplayer)).
input(oplayer, noop) :-
    base(control(xplayer)).
```

The answer set program P that we generate from a game description G is defined as $P = G \cup P^{base} \cup$

```
  {true(F):base(F)}.
  1={does(R, M):input(R, M)} :- role(R).
```

As can easily be seen, our definition of P^{base} (and thus P) fulfills the restrictions for a valid GDL description (Definition 1) if the original game description G belongs to the bounded fragment of GDL (Definition 2). The reason is that in P^{base} we introduce recursions involving $\texttt{true}^{static}$ and $\texttt{next}^{static}$ (which are both `base`) and therefor also between $\texttt{legal}^{static}$ and $\texttt{does}^{static}$ (both `input`). Thus, all bounded GDL programs can be grounded using this method in principle. GDL descriptions not fulfilling the restrictions for bounded GDL, can lead to an infinite ground representation.

That said, grounding bounded GDL can still lead to an exponential blowup in the size of the representation which can make grounding infeasible. Especially games containing rules with many variables suffer from this problem.

Optimizations Before grounding the answer set program P, we apply optimizations to it similar to the ones described in [Haufe *et al.*, 2012] (Section 6.2). That is, we try to reduce the resulting grounding by removing existential variables and removing unnecessary rules, as illustrated in the following paragraphs.

As an example, consider the rule `p(X,Z) :- q(X,Y), r(Y), s(Z)..` The variable `Y` in the body is existentially quantified (does not appear in the head). We replace this rule by

```
   p(X,Z) :- qr(X), s(Z).
   qr(X)  :- q(X,Y), r(Y).
```

where `qr` is a new predicate symbol and obtain two rules with two variables each instead of one rule with three variables. This reduces the number of ground rules that are generated (unless the domains of the variables are singletons).

Some rules in the game descriptions are unnecessary and can be removed. For example, Tic-Tac-Toe (see Figure 1) contains the rules for `line(X)`. In those rules `X` can be replaced with any of $\{x, o, b\}$, however only `line(x)` and `line(o)` appear in the body of another rule. Thus, the ground rules that would be generated for `line(b)` are irrelevant as are the ground instances of `row`, `column` and `diagonal` where `X` is replaced with `b`. We prevent these unnecessary rules from being generated in the first place, by instantiating the X in the rules for `line(X)` with `x` and `o` and handing these partially instantiated rules to the grounder.

6 Experiments

We ran experiments on 231 games from the GGP server [Schiffel, 2016]. For each game we ran our grounder and recorded

- the time it took to generate the answer set program P;
- the total runtime for grounding (including the time for generating P);
- the size of the resulting ground description in terms of number of resulting clauses;
- the number of components of a propositional network created from the those clauses without optimizations.

For comparison, we used the GGP-Base framework [Schreiber and Landau, 2016] to generate a propositional network (propnet) using the OptimizingPropNetFactory class. Generating a propnet includes grounding the game description as a first step and we measured only the time for this step without the remaining time that is spent on optimizing the propositional network. However, these two are somewhat intertwined such that a complete separation is not possible. GGP-Base makes use of **`base`** and **`input`** predicates. Since most games on the GGP server do not contain **`base`** and **`input`** definitions, we added the base and input definitions that were generated by our own grounder to the game rules that were given as input to the propnet generation. For GGP-Base we recorded the runtime and the number of resulting components, where each component represents a conjunction, disjunction, negation or proposition in the (grounded) rules. Thus, this number is roughly comparable to the number of clauses (including facts) in the grounded game description.

The ASP-based grounder can ground 226 of the 231 tested games within the time limit of 1h and memory limit of 4GB. The median runtime was 1.4s (average 4.5s), which includes the time for starting an external process for the grounder and reading the resulting grounded game description. For comparison, the GGP-Base grounder can ground 218 of the tested games with a median runtime of 2.4s (average 5.9s). Since no external process needs to be started, this runtime does not include any process communication overhead. There was no game that could be grounded using GGP-Base but not using ASP. Most of the games that could be handled by the ASP-based grounder but not by GGP-Base feature heavy use of recursive rules. Neither system could ground laikLee_hex, merrills, mummymaze1p, ruledepthquadratic or small_dominion. All of those games feature recursive rules, except for mummymaze1p, which could be grounded by the ASP system resulting in about 5 million clauses, but processing the ground clauses and generating the propnet took too much time. The games farmers, god, Goldrush, kalaha_2009, quad_5x5, SC_TestOnly, sudoku_simple and uf20-01.cnf.SAT could be grounded by the ASP-based grounder, while the GGP-Base grounder exceeds the run-time limit. Of those games, only farmers and kalaha_2009 can be considered complex games taking 25.4s and 21.7s to ground respectively and resulting in more than 100000 components. The other games could all be grounded by the ASP-based grounder in under 5s resulting in no more than 13000 components.

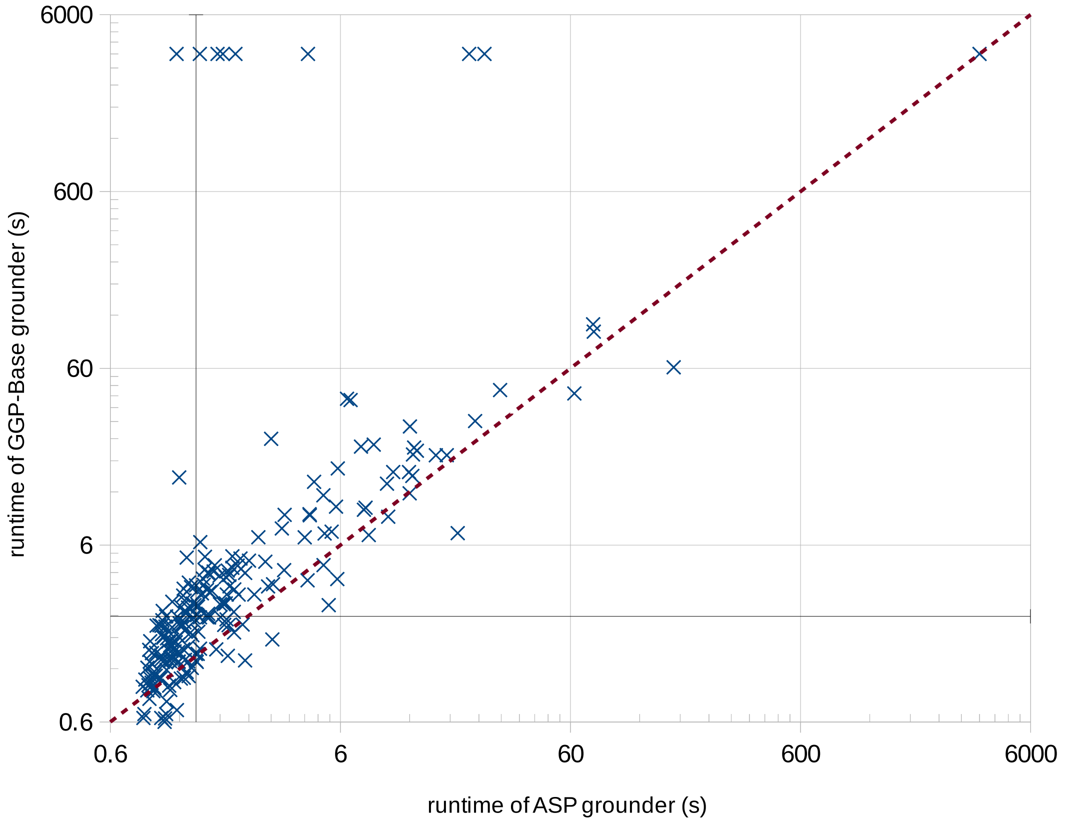

Figure 2: Runtime for grounding game descriptions with GGP-Base vs. our ASP-based grounder. Points above the diagonal denote games where the GGP-Base grounder takes longer than the ASP-based grounder. The runtime limit was set to 1 hour. Thus, points at x=3600s denote games where the ASP grounder failed to ground the game within the time limit (and conversely for the GGP-Base grounder). The horizontal and vertical lines show the median runtime of GGP-Base and ASP-based grounder, respectively.

Figure 2 shows the runtimes of the GGP-Base vs. ASP-based grounder for all 231 games on a logarithmic scale. We can see that most games can be grounded in less than 1 minute with both grounders. In 173 cases the ASP-based grounder is faster than the GGP-Base grounder (some of them within the margin of error). On average, the ASP-based grounder is 20.4% faster then the GGP-Base grounder.

We compared the size of the grounded description with both systems in Figure 3. As can be seen, there is little difference between both systems, but the GGP-Base system creates significantly larger groundings in few selected games (smallest, logistics, mastermind, crossers3, othello-comp2007, othellosuicide, racer, racer4, battlebrushes). The number of clauses of the grounded game descriptions range from 26 (from troublemaker01) to 2038583 (for battlesnakes1509) with a median of 2444. The number of components in the generated propositional networks is similar (between 46 and 2715284). The median number of generated components is 2455 for the the ASP-based grounder vs. 2518 for the GGP-Base grounder.

In Figure 4, we plotted the size of the ground representation (propositional network) compared to the size of the original GDL rules for each of the games. We used the smaller of the two groundings for each game and measured the size of the rules by taking the sum of the number of literals of all rules. As can be seen in the graph, the propositional network is generally some orders of magnitude larger than the GDL rules, except for some edge cases where the rules are essentially already grounded. However, we can not see a general trend indicating an exponential blowup in size. That is, although this blowup is theoretically possible, it seems to happen rarely in the games we looked at. In fact, the best fit to the data (excluding the abnormal cases) seems to be a power law which puts the propnet size at slightly more than a quadratic function of the size of the GDL rules.

7 Conclusion

Grounding game descriptions using a state-of-the art answer set programming system is a viable alternative to the GDL specific approach implemented in the GGP-Base framework. The system we presented is able to handle more games and is typically faster despite the overhead of transforming GDL

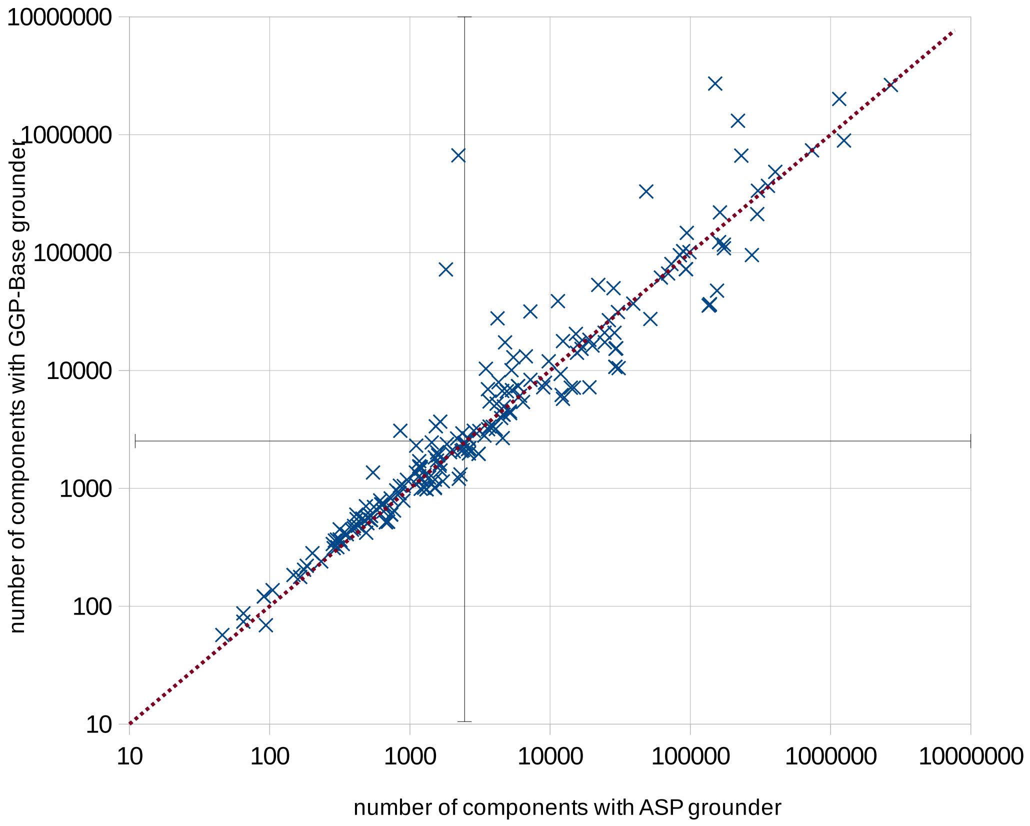

Figure 3: Number of components of a propositional network created from the grounding resulting from GGP-Base vs. our ASP-based grounder. Points above the diagonal denote games where the GGP-Base grounder creates larger propnets. The horizontal and vertical lines show the median numbers of components for GGP-Base and ASP-based grounder, respectively.

into a different format and starting and communicating with a separate process. Furthermore, our grounding of a game description is well-founded theoretically by the transformation into answer set programs. This allows to optimize the descriptions further without changing their semantics. In the future, we plan to look into further optimizations of the grounding to allow grounding of more complex game descriptions. Additionally, these optimizations will likely reduce the size of the grounded descriptions which generally leads to faster reasoning with the grounded game descriptions, for example, in the form of propositional networks. However, even with those optimization there will likely be games where the potential exponential blowup will prevent grounding from being feasible. In those cases it is necessary to fall back on reasoners that do not require a propositional representation (e.g., Prolog).

References

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

[Baral, 2003] Chitta Baral. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press, New York, NY, USA, 2003.

[Cerexhe *et al.*, 2014] Timothy Cerexhe, David Rajaratnam, Abdallah Saffidine, and Michael Thielscher. A systematic solution to the (de-)composition problem in general game playing. In *Proceedings of the European Conference on Artificial Intelligence (ECAI)*, 2014.

[Cox *et al.*, 2009] Evan Cox, Eric Schkufza, Ryan Madsen, and Michael Genesereth. Factoring general games using propositional automata. In *Proceedings of the IJCAI-09 Workshop on General Game Playing (GIGA'09)*, pages 13–20, 2009.

[Edelkamp and Kissmann, 2011] Stefan Edelkamp and Peter Kissmann. On the complexity of bdds for state space search: A case study in connect four. In *Proceedings of the AAAI Conference on Artificial Intelligence*. AAAI Press, 2011.

[Gebser *et al.*, 2011] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Marius Schneider. Potassco: The Potsdam answer set solving collection. *AI Communications*, 24(2):107–124, 2011.

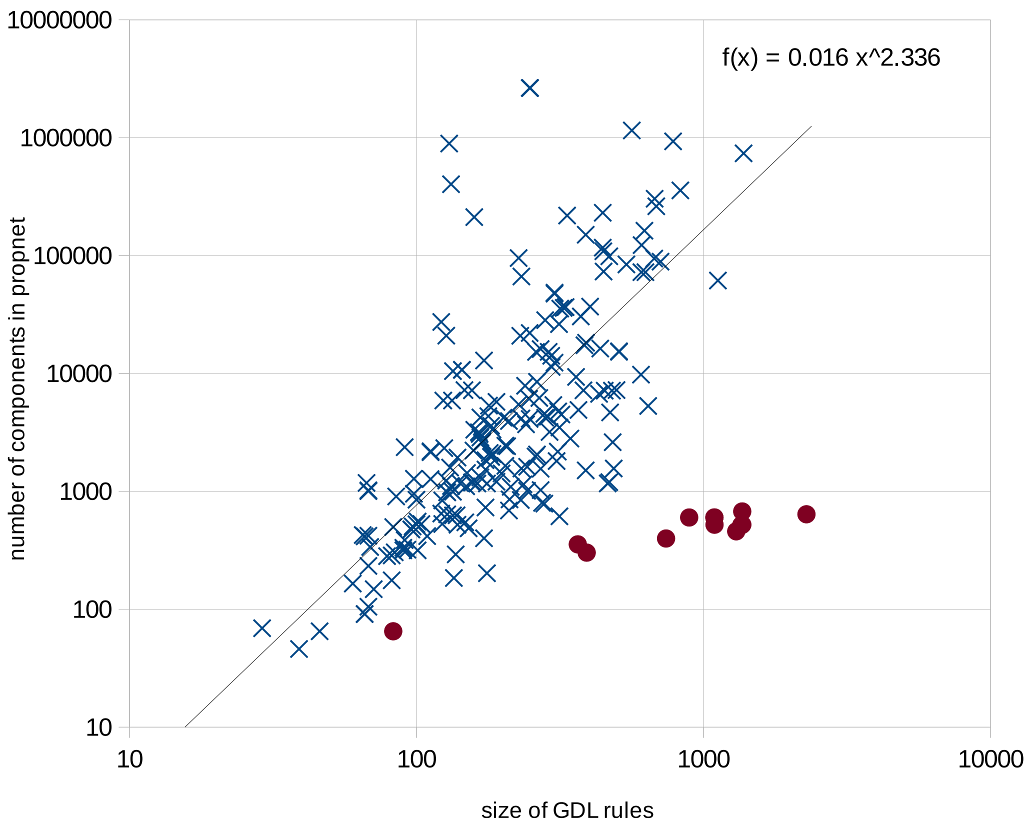

Figure 4: Number of components of the propositional network compared to the size of the original GDL description (without base and input rules) measured in number of literals for all 226 games that could be grounded. The dots denote games that are abnormal in that the GDL rules are larger than the propnet. All of these games turned out to be either test case games made to test certain aspects of GDL reasoners (as opposed to general game players) or games that are essentially already ground. The line shows the best matching regression.

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

[Kissmann and Edelkamp, 2010] Peter Kissmann and Stefan Edelkamp. Instantiating general games using prolog or dependency graphs. In *German Conference on Artificial Intelligence*, pages 255–262, 2010.

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

[Love *et al.*, 2008] Nathaniel Love, Timothy Hinrichs, David Haley, Eric Schkufza, and Michael Genesereth. General game playing: Game description language specification. Technical report, Stanford University, 2008.

[Michulke and Schiffel, 2013] Daniel Michulke and Stephan Schiffel. Admissible distance heuristics for general games. In *Agents and Artificial Intelligence*, volume 358, pages 188–203. Springer Berlin Heidelberg, 2013.

[Saffidine, 2014] Abdallah Saffidine. The game description language is turing complete. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(4):320–324, Dec 2014.

[Schiffel and Björnsson, 2014] Stephan Schiffel and Yngvi Björnsson. Efficiency of GDL reasoners. *Computational Intelligence and AI in Games, IEEE Transactions on*, 6(4):343–354, 2014.

[Schiffel and Thielscher, 2009a] Stephan Schiffel and Michael Thielscher. Automated theorem proving for general game playing. In *Proceedings of IJCAI'09*, pages 911–916, 2009.

[Schiffel and Thielscher, 2009b] Stephan Schiffel and Michael Thielscher. A multiagent semantics for the game description language. In *Agents and Artificial Intelligence*. Springer, 2009.

[Schiffel, 2010] Stephan Schiffel. Symmetry detection in general game playing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, pages 980–985. AAAI Press, 2010.

[Schiffel, 2016] Stephan Schiffel. GGPServer. `http://ggpserver.general-game-playing.de/`, 2016.

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

[Schreiber and Landau, 2016] Sam Schreiber and Alex Landau. The general game playing base package. `https://github.com/ggp-org/ggp-base`, 2016.

[van Gelder, 1989] Allen van Gelder. The alternating fixpoint of logic programs with negation. In *Proceedings of the 8th Symposium on Principles of Database Systems*, pages 1–10. ACM SIGACT-SIGMOD, 1989.