Instantiating General Games using Prolog or Dependency Graphs

Peter Kissmann and Stefan Edelkamp

TZI Universität Bremen, Germany
kissmann,edelkamp@tzi.de

Abstract. This paper proposes two ways to instantiate general games specified in the game description language GDL to enhance exploration efficiencies of existing players. One uses Prolog's inference mechanism to find supersets of reachable atoms and moves; the other one utilizes dependency graphs, a datastructure that can calculate the dependencies of the arguments of predicates by evaluating the various formulas from the game's description.

1 Introduction

General game playing (GGP) is concerned with the playing of games whose rules are not known beforehand. Thus, in contrast to classical game playing it is not possible to write a highly specialized game player, such as DEEP BLUE [3] for Chess, or CHINOOK [15] for American Checkers. Instead, the players are supposed to perform well on a much larger variety of games.

Most of the successful players of the last years (e. g., CADIAPLAYER [5] or ARY [14]) use the Monte-Carlo based approach UCT [11] to play general games. So far, they still use the uninstantiated input specified in the game description language GDL [13] and use Prolog to infer knowledge about possible moves and successor states.

Initial experiments with pure Monte-Carlo runs on a number of games indicate that using Prolog on the uninstantiated input usually leads to a significantly worse runtime behavior in the number of expanded nodes per second compared to the performance on instantiated input (see Table 1). Thus, transforming the input to instantiated form, i. e., one without any variables, should be of high priority.

Moreover, in the domain of action planning almost all recent planners (including, e. g., FF [7], GRAPHPLAN [1], and SATPLAN [8]) infer such an instantiated input. The typical input comes in uninstantiated form, so there exists the same problem as in general game playing. Therefore, we expect that the next generation of efficient general game players will avoid unification for binding variables during play via a static analysis prior to the search that yields an instantiated version of the input.

To come up with a superset for all reachable atoms, moves and axioms (as needed for instantiating the problem) we use either a fixpoint computation via unification by implementing an inference interface to the logical programming language Prolog or the knowledge we gain by evaluating the dependency graphs for the several structures in the game's description.

Table 1. Number of expanded nodes using pure Monte-Carlo search with Prolog and with instantiated input. The timeout was set to 10 seconds.

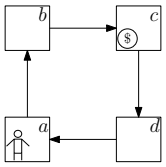
Game	Exp_{Prolog}	$\text{Exp}_{Inst.}$	Factor	Game	Exp_{Prolog}	$\text{Exp}_{Inst.}$	Factor
asteroidsserial	59,364	219,575	3.70	lightsout	28,800	7,230,080	251.04
beatmania	28,680	3,129,300	109.11	pancakes6	154,219	2,092,308	13.57
chomp	22,020	1,526,445	69.32	peg_bugfixed	19,951	1,966,075	98.55
connectfour	44,449	2,020,006	45.45	sheep_and_wolf	20,448	882,738	43.17
hanoi	84,785	7,927,847	93.51	tictactoe	65,864	5,654,553	85.85

This paper, which provides an extension and improvement to a precursing workshop paper [9], is structured as follows: In Section 2, we give a brief introduction to general game playing and the game description language. In Section 3, we describe the instantiation process. In Section 4, we present our experimental results. Finally, we draw some conclusions in Section 5 and show ideas for further improving the instantiation process.

2 The Game Description Language GDL

In recent years the game description language GDL [13] has become the language of choice for modeling general games. It is a logic-based language, which describes games in an uninstantiated form, i. e., the game description typically incorporates variables.

In the following we will explain the important keywords of GDL and give examples according to the description of the game Maze, which is a single-player game containing four cells with the robot starting in a and a stack of gold lying in c. The goal is to move the gold to a. If this goal is not reached, the game ends after at most nine moves. The cells are connected as shown in the figure to the right.



All the formulas consist of a body (an arbitrary formula, which might be empty) and a head, which is a positive atom.

role: Specifies the names of the players defined in the game.

```
(role robot)
```

init: Specifies the initial state.

```
(init (cell a)) (init (gold c)) (init (step 1))
```

true: Denotes the atoms true in the current state (appears only in bodies of formulas).
state axioms: Help to shorten the description. The operator <= denotes the implication, so that some of the axioms can be considered as derived predicates.

```
(adjacent a b) (adjacent b c) (adjacent c d) (adjacent d a)
(<= timeout (true (step 10)))
```

variables: Variables are denoted with prefix `?`.
legal: Describes the moves possible in a specific state for one of the players. For each reachable non-terminal state and each player at least one move has to be possible.

```
(<= (legal robot grab) (true (cell ?x)) (true (gold ?x)))
(<= (legal robot move))
```

does: Specifies the moves chosen by the players (appears only in the formulas' bodies).
next: Determines the successor state, mostly dependent on the moves chosen by all the players. The frame is modeled explicitly in GDL, i. e., any state atom not satisfied by any of the `next` formulas is supposed to be false.

```
(<= (next (cell ?y))
    (does robot move) (true (cell ?x)) (adjacent ?x ?y))
```

terminal: Provides a description of the terminal states. The game ends once one of these states is reached.

```
(<= terminal timeout)
(<= terminal (true (gold a)))
```

distinct: States that the two arguments must differ (appears only in formulas' bodies).
goal: Specifies the rewards (within $[0, \ldots, 100]$) for the players for each possible terminal state. Higher rewards are to be prefered.

```
(<= (goal robot 100) (true (gold a)))
(<= (goal robot 0) (true (gold ?x)) (distinct ?x a))
```

The game starts at the initial state. Each player has to check which moves it can perform and select one of these. Once all players have chosen a move, the successor state is calculated and the process starts over. When a terminal state is reached, the game ends and the players get the corresponding rewards.

3 Instantiation Process

There are several cases where it might be important to have the games' descriptions in an instantiated form. One is our general game solver [10], which uses binary decision diagrams (BDDs) [2] to represent the states. To create a BDD, it is important to minimize the number of variables needed for representing any state. This number depends on the total number of reachable state atoms (it might be reduced if groups of mutually exclusive atoms can be found). The complete instantiation process works similar to the ideas in [4,6] (see Algorithm 1). Apart from some more subtle differences, GDL differs from PDDL mainly by modeling the frame explicitly and by splitting the moves into two sets of formulas, namely the `legal` formulas, which act as a move's precondition, and the `next` formulas, which specify the effects, but the moves for which they are relevant are denoted by the `does` terms and thus can appear deep inside the body's formula. Thus, the various steps cannot be performed in the same way as in planning. In the following, we will present the critical steps in more detail.

3.1 Calculating Supersets of Reachable State Atoms, Moves and Axioms

Here, the two approaches (using Prolog and dependency graphs) differ.

Algorithm 1: Instantiation

1 Parse the GDL input.
2 Create the disjunctive normal form of the bodies of all formulas.
3 Calculate the supersets of all reachable atoms, moves and axioms (see Section 3.1).
4 Instantiate all formulas (see Section 3.2).
5 Find groups of mutually exclusive atoms (see Section 3.3).
6 Remove the axioms (by applying them in topological order).
7 Generate the instantiated GDL output.

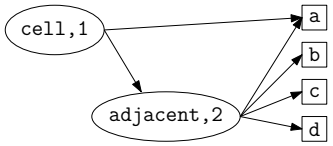
Prolog We generate a Prolog description of the game but remove all negated atoms (in disjunctive normal form, any negation appears only right in front of an atom).

We repeat the following two steps, until no new moves or state atoms are found. The knowledge base contains all atoms and moves reached so far. Given these, we check for possible moves by querying Prolog for `legal(P, M)`[1]. For each returned player-move pair we add a corresponding move into the knowledge base by using Prolog's assert mechanism (`assert(does(player, move))`). Now, we determine the atoms true in the successor states by querying `next(Atom)` and insert them into the knowledge base as well (`assert(true(atom))`).

When a fixpoint is reached, the knowledge base contains a superset of all reachable state atoms and moves. To also come up with a superset of reachable axioms, we iterate through all the axioms and send queries to the Prolog program (by calling `ax(V1, ..., Vn)` with `ax` representing an axiom with n parameters).

Dependency Graph We found out that in several games Prolog is rather slow as it finds a large amount of duplicates. In these cases, our second approach using a dependency graph often is superior to the Prolog approach.

We calculate the dependencies of any formula, i. e., for each head we connect its arguments to the arguments of predicates in the body containing the same variables (see the figure to the right displaying a part of the dependencies for the game Maze[2], esp. those for the `next` formula and the axioms in Section 2), similar to [16].



When resolving these dependencies, we come up with possible assignments for each of the parameters of each predicate. Unfortunately, we lose all information about the interaction between the different parameters, e. g., in a chess-game where a rook can move only along rows or columns, we may lose this information, so that a rook's move might be assumed to be possible from any cell to any other cell (depending on the quality of the game's description). To remedy this we perform several post-processing steps. We start by calculating all the possible state atoms, moves and axioms that the dependency graph suggests. Then we evaluate each formula for a given head to check if it might be applicable. This way, we can erase several of the unreachable state atoms, moves and axioms, so that the next stepwill take significantly less time.

[1] For Prolog we denote variables by starting with a capital letter.
[2] Note, that we use `pred,n` to denote the nth argument of the predicate `pred`.

```
(<= (next (cell b))    (<= (next (cell c))
    (does robot move)      (does robot move)
    (true (cell a))        (true (cell b))
    (adjacent a b))        (adjacent b c))

(<= (next (cell d))    (<= (next (cell a))
    (does robot move)      (true (cell d))
    (true (cell c))        (does robot move)
    (adjacent c d))        (adjacent d a))
```

Fig. 1. Instantiated `next` formula from the Maze example.

3.2 Instantiating Formulas

Once the supersets of all reachable state atoms, moves and axioms are found we can instantiate all the given formulas. A naïve idea would be to test each possible instantiation of each predicate of each formula and keep only those that are not conflicting (due to multiple assignments to variables), but this results in a big overhead concerning memory and runtime.

As we already calculated the disjunctive normal form and split those formulas whose base operator is a disjunction, all formulas incorporate only conjunctions. For these, we determine the number of different variables within the formula and create a matrix. The columns of this matrix represent the different variables, the rows their possible assignments. To get those we consider each predicate appearing in the conjunction and find, for each predicate, the set of reachable instantiated atoms (state atoms, moves or axioms), no matter how the other predicates might be instantiated.

To come up with the full instantiations we combine the assignments of the variables. For each predicate of the conjunction we need to check if the chosen instantiation is indeed possible. In case of some variables being `distinct` we additionally check if they are still distinct after instantiation (if they are not, we forget about this instantiation). When we are done, we have generated a superset of all reachable instantiations of the formulas.

In the dependency graph approach, we found that these supersets still are too large in several cases. Thus, we perform another post-processing step. Right after having calculated all the instantiated formulas, we perform a reachability analysis using these formulas but omit the negations, which is similar to what we did using Prolog with the uninstantiated formulas. In contrast to that approach, here we do not suffer from predicates being satisfied by several formulas and thus a large number of duplicates, so that the overhead in the runtime is small compared to the previous calculations.

The `next` formula from Section 2 would be represented by the matrix displayed to the right resulting in the instantiated formulas shown in Figure 1.

	?y	?x
(next(cell ?y))	a	
	b	
	c	
	d	
(true(cell ?x))		a
		b
		c
		d
(adjacent ?x ?y)	b	a
	c	b
	d	c
	a	d

3.3 Finding Mutex Groups

In principle, at this point the instantiation is finished. To be prepared for a better evaluation of the games, we next calculate groups of mutually exclusive state atoms. A group of n mutually exclusive atoms can be encoded using $\lceil \log n \rceil$ bits.

To calculate these mutexes we proceed similar to the ideas of [12,16]. For each predicate they try to find its input and output parameters. In most cases the first denote some position on a game board, whereas the latter denote the tokens placed there. For one predicate, all instantiated atoms with the same input parameters but different output parameters are mutually exclusive. To calculate these mutex groups we start by creating several hypotheses for each predicate specifying which of the parameters might be input parameters and check these in a number of states. Finally, we retain that hypothesis that holds in all visited states and produces the smallest state encoding. Using this, for a predicate we group those instantiated atoms together that share the same input parameters. We add an additional atom to each group denoting that there might be states where none of the group's atoms hold. Only for predicates with one parameter we check, if they appear in each visited state. If that is the case, we omit the additional atom.

To find the input and output parameters we write a Prolog program representing the game. With this we perform several simulation steps, where we calculate the `legal` moves for each player and choose one randomly. After each step we check, if the hypotheses hold by evaluating the corresponding predicates true in the current state. For these, we analyze their supposed input parameters by sorting and scanning them. If we find some duplicates, we know for certain that this set of parameters does not form a set of input parameters and discard the corresponding hypothesis. In the worst case, we have to check a number of hypotheses exponential in the number of parameters for each predicate, but often checking the initial state already discards lots of these hypotheses. Also, most games have only predicates with three or four parameters, which would result in at most three or six hypotheses, respectively.

In our experiments we observed no significant slowdown. Also, we found that in many cases already the analysis of the initial state yielded the final hypotheses. Only the games where the game board is not completely specified beforehand, e. g., in ConnectFour or TicTacToe, we need a number of steps to find the final hypotheses. Opposed to [12,16], for us two or three steps might not be enough. Especially in ConnectFour some more steps are required to reliably get correct mutexes. So, we chose to perform 50 steps, but still there is no guarantee that the final hypotheses are correct.

4 Experimental Results

We have implemented the instantiator in C++ using SWI-Prolog[3] and performed some experiments on our machine (one core of an Intel i7 920 with 2.67 GHz and 12 GB RAM). Using a timeout of one minute, which is close to the startup time during a competition, with this implementation we can instantiate 96 of the 171 enabled games from the website of Dresden's GGP server[4] using the Prolog based approach and 90

[3] `http://www.swi-prolog.org`

[4] `http://euklid.inf.tu-dresden.de:8180/ggpserver`

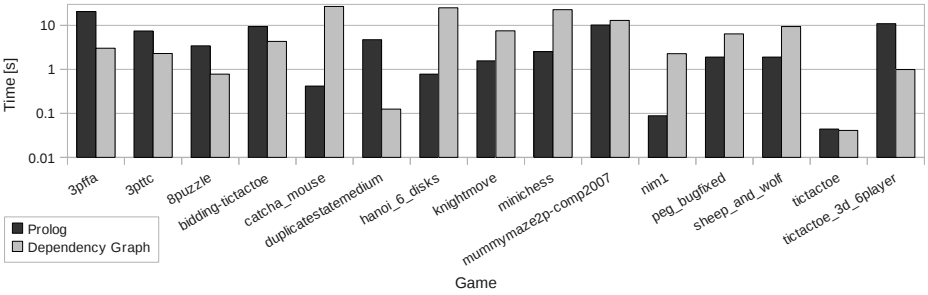


Fig. 2. Results for some of the instantiated games.

games using the dependency graphs. Of all the games instantiated, the Prolog based approach instantiated 11 games the dependency graph based one cannot, while the latter can instantiated 4 we cannot when we use Prolog.

A number of the games on the server (20) uses GDL features we do not yet support, so that we cannot instantiate them for now; mostly these are encapsulated formulas. For the remaining games, we often run out of time (typically in the step of finding the supersets, or the one where we instantiate the formulas) or out of memory (either during the instantiation step or the removal of the axioms).

Note that, compared to our workshop paper [9], we used another output. While the GDDL output we previously generated is better for our solver and in most cases also our player, here we output instantiated GDL (or KIF), so that our output can also be processed by all existing players. Also, especially for multi-player games, the generation of GDDL often dominates the runtime. So, we decided to generate the simpler output to better find the differences between the two instantiation approaches.

For a large number of the games the runtimes are similar with both approaches. For some games such as `3pffa`, `duplicatestate`, or `tictactoe_3d_6player` we found that the Prolog based approach is a lot slower, because the inference mechanism returns too many duplicates, while for others such as `catcha_mouse`, `hanoi`, `minichess`, `peg`, or `sheep_and_wolf` the dependency graph based approach initially generates supersets that are a lot larger than the final ones, so that the instantiation of the formulas is more time-consuming and also the post-processing steps take a while. Some results can be found in Figure 2.

5 Conclusion and Future Work

In this paper we presented a way to instantiate general games specified in GDL, where we followed two different approaches for finding the supersets of reachable state atoms, moves and axioms, i. e., Prolog and dependency graphs.

The instantiator has been used to instantiate several games, though there are some for which the process either takes too much time or the memory runs out.

Especially for games containing lots of duplicates (or different ways to satisfy a formula), dependency graphs can outperform Prolog, while Prolog is faster if the supersets

found by the dependency graph are too large. Thus, one might try to find out automatically which algorithm to use for instantiation – or to run both instantiations in parallel on two cores. Alternatively, if it is possible, decreasing the sizes of the supersets found by the dependency graph earlier should positively influence that algorithm's runtime.

Another aspect for future work might be the parallelization of the instantiation process. At least for some steps, e. g., the calculation of the disjunctive normal form (step 2) as well as the instantiation of the formulas (step 4), a parallel calculation is possible.

Acknowledgments Thanks to DFG for support in project ED 74/11-1 and to the anonymous reviewers for their helpful comments.

References

1. A. L. Blum and M. L. Furst. Fast planning through planning graph analysis. In *IJCAI*, pages 1636–1642, 1995.
2. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.
3. M. Campbell, A. J. Hoane, Jr., and F.-H. Hsu. Deep Blue. *Artificial Intelligence*, 134(1–2):57–83, 2002.
4. S. Edelkamp and M. Helmert. Exhibiting knowledge in planning problems to minimize state encoding length. In *ECP*, volume 1809 of *LNCS*, pages 135–147. Springer, 1999.
5. H. Finnsson and Y. Björnsson. Simulation-based approach to general game playing. In *AAAI*, pages 259–264, 2008.
6. M. Helmert. *Understanding Planning Tasks: Domain Complexity and Heuristic Decomposition*, volume 4929 of *LNAI*. Springer, 2008.
7. J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *JAIR*, 14:253–302, 2001.
8. H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *AAAI*, pages 1194–1201, 1996.
9. P. Kissmann and S. Edelkamp. Instantiating general games. In *IJCAI-Workshop on General Game Playing*, pages 43–50, 2009.
10. P. Kissmann and S. Edelkamp. Layer-abstraction for symbolically solving general two-player games. In *SoCS*, 2010.
11. L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo planning. In *ECML*, volume 4212 of *LNCS*, pages 282–293, 2006.
12. G. Kuhlmann, K. Dresner, and P. Stone. Automatic heuristic construction in a complete general game player. In *AAAI*, pages 1457–1462, 2006.
13. N. C. Love, T. L. Hinrichs, and M. R. Genesereth. General game playing: Game description language specification. Technical Report LG-2006-01, Stanford Logic Group, April 2006.
14. J. Méhat and T. Cazenave. Ary, a general game playing program. In *13th Board Game Studies Colloquium*, 2010.
15. J. Schaeffer. *One Jump Ahead: Computer Perfection at Checkers*. Springer, 2009.
16. S. Schiffel and M. Thielscher. Fluxplayer: A successful general game player. In *AAAI*, pages 1191–1196, 2007.