Symbolic Exploration for General Game Playing in PDDL

PhD Student: Peter Kissmann* and **Phd Advisor: Stefan Edelkamp**
Computer Science Department
University of Dortmund, Germany
{peter.kissmann, stefan.edelkamp}@cs.uni-dortmund.de

Introduction

In artificial intelligence (AI), two- and multiplayer games have been of some interest, but the best AI algorithms always had considerable knowledge of the game they were designed for (Schaeffer 2000).

In *general game playing* (Love, Hinrichs, & Genesereth 2006) strategies are computed domain independently without knowing which game is played. Best policies result in perfect play. The opponents attempt to maximize their gain.

Game trees are often depth bounded and values at the leaf nodes of the trees are computed by a static evaluation function. On the other hand, retrograde analysis (Schaeffer *et al.* 2005) calculates databases of classified positions in backward direction, starting from won and lost ones. These endgame databases can be used in conjunction with game playing programs to eventually *solve* the game by computing the game theoretical status of the initial position.

The purpose of this paper is to close the gap between general game playing and domain independent action planning. Moreover, we illustrate how symbolic planning technology can be applied.

Description Languages

We briefly review the origins of the two formalisms.

GDL The game description language (GDL) (Love, Hinrichs, & Genesereth 2006) is designed for defining complete information games. GDL is a Datalog-inspired language for finite games with discrete outcomes for each player. Broadly speaking, every game specification describes the states of the game, the legal moves, and the conditions that constitute a victory for the players. This definition of games is similar to the traditional definition in game theory (Rapoport 1966) with a couple of exceptions. In this version, a game is a graph rather than a tree. This makes it possible to describe games more compactly, and it makes it easier for players to play games efficiently. Another important distinction between GDL and classical definitions from game theory is that states of the game are described succinctly, using logical propositions instead of explicit trees or graphs.

PDDL and GDDL The planning domain definition language (PDDL) is the standard language for the encoding of planning domains. The original version of the language was developed by McDermott (2000). We use PDDL 2.2, level 1 (Hoffmann & Edelkamp 2005) as the basic language.

Since all games in GDL are finite, it is possible, in principle, to describe games in the form of lists. Unfortunately, such explicit representations are not practical. Therefore, we translate GDL into GDDL (for *game domain description language*)[1], which matches the syntax of PDDL 2.2, level 1, with the exception that the evaluation of goals is handled differently. The only additional construct is:

```
(:gain <parameters> <number> <body>)
```

with `number` being a bounded integer in $\{0, \ldots, 100\}$, `body` a goal description and `parameters` a typed list. When a terminal state is reached, each player receives a certain gain (the higher the better) depending on the current state. This is described by the above gain construct. To allow existing PDDL parsers (like *Adl2Strips* by Hoffmann) to handle the extended input, specialized actions embed the gain in their name.

Symbolic Exploration

Symbolic planning is based on checking the satisfiability of formulas (Kautz & Selman 1996). We refer to symbolic exploration only in the context of using BDDs (Bryant 1985). While invented in model checking, BDDs contribute to many successful AI planning systems (Cimatti, Roveri, & Traverso 1998; Edelkamp & Helmert 2001; Jensen 2003). Compared to the space requirements of explicit-state planners, symbolic planning systems save space by exploiting a shared representation of states. This has a drastic impact on the design of available algorithms, as not all algorithms adapt to the exploration of state sets.

Symbolic search executes a functional exploration of the problem graph. This functional representation of states and actions then allows to compute the functional representation of a set of successors, or the *image*. As a byproduct, the functional representation of the set of predecessors, or the *preimage*, can also be efficiently determined.

The automated inference of a minimized state encoding

*After some research on the externalization of the multiple sequence alignment problem (Kissmann 2007), the author now concentrates on general game playing.

[1]For an overview of GDDL and a selection of models, see http://ls5-web.cs.uni-dortmund.de/∼edelkamp/pddl-games.

for a propositional planning problem in PDDL refers to work by Edelkamp & Helmert (1999). The automated translation of planning problems into BDD representations is provided in Edelkamp & Helmert (2001).

One novelty is the compilation of derived predicates. A planning instance is compiled into an equivalent description without derived predicates as follows. On the fully instantiated level derived predicates define a partial order, which can be sorted topologically. This allows to substitute the derived predicates in preconditions of actions, termination criteria, or bodies of other derived predicates one after the other.

Reachability Not all positions that can be expressed in the domain language are actually reachable from the initial state. Essentially, a reachability analysis corresponds to a symbolic breadth-first search traversal. Starting with the initial state, successor states are generated until all of them are present. As in GDL the game ends once a terminal state is reached, only non-goal states from the search frontier are expanded. When no new state is generated, i.e., all states reachable from the initial state are created, the algorithm terminates. Its pseudo-code can be found in Edelkamp (2002).

Classification of Single-Player Games For single-player games we can partition all reachable states, i.e., we can determine the maximal gain that the player can achieve in a specific state. Furthermore, we can give a strategy for each state that leads to the goal that achieves the calculated gain.

The player aims at maximizing the received gain. Thus we first calculate all states that lead to a gain of 100 by starting with the reachable goal states that achieve a gain of 100 and calculating all their predecessors. Afterwards we iteratively calculate the states leading to lesser gains. During this we remove those states that can also establish higher gains to determine the maximal possible gain for each state.

When all buckets are properly filled, the user might want to know which gain a given state might lead to in optimal play. To do this, the BDD for the state of interest will be constructed and, starting at bucket 100, the conjunction with the BDD in each bucket will be determined. If the conjunction is not *false*, the state will lead to the corresponding gain.

Classification of Two-Player Games Two-player games with perfect information are classified iteratively. Edelkamp (2002) presented a symbolic algorithm for the retrograde analysis of turn-taking two-player zero-sum games. In GDL the alteration of the players' moves is determined by the existence of a predicate *control-player* for both players.

We present a new algorithm that is more complex but much more general in that it works well for any gains from the set of $\{0, \ldots, 100\}$ for each player. Thus the only restriction we impose is alternation of the players. To achieve this we generate a 101×101-matrix of all possible gain combinations for the two players. The entry at bucket (i, j) is initialized with the conjunction of the BDD representing gain i for player 1 and the one representing gain j for player 2. Once there are no new predecessors in any bucket, the algorithm ends. Also important is the step number k, which is initialized to 0.

We start by calculating the predecessors in which one player had control, i.e., he could perform the corresponding moves This we repeat, alternating with those in which the other player had control. After each calculation of predecessors we might get duplicates in some buckets as we calculate the predecessors of all buckets, which might overlap. These duplicates will be deleted and we restart and reset k to 0.

To maximize the gain for one player, we retain only those duplicate states that achieve highest gain for him. If then there are still duplicates we delete all except for the ones that achieve the least gain for the opponent. So after securing maximal gain for the current player, we minimize the gain for his opponent.

In order to prevent the newly deleted states from being created in the corresponding buckets after restart, we need to store them in a matrix of lists of forbidden states. One bucket of this matrix contains a list of BDDs representing the forbidden states for the corresponding bucket along with the k of the calculation in which they were forbidden. When creating the predecessors of a certain set of states, we remove those that are in the same bucket within the forbidden matrix. The restart is necessary to delete the predecessors of the now forbidden states. If we delete states in a certain step, we remove all those from the list of forbidden states that were generated with a smaller k. As we delete the predecessors of the newly forbidden states, it might be that these were responsible for the deletion of some states. So when the now forbidden states and their predecessors are no longer generated, those deleted states can be created again.

If there were no duplicates, we increment k and continue, until no new states were generated. Once the algorithm stops, we simply check in which bucket the initial state resides. This then represents the gain for both players.

An extended description of this algorithm can be found in Edelkamp & Kissmann (2007).

Experimental Results

We implemented the above algorithms of our game based planner in Java and performed the experiments on an AMD Opteron processor with 2.3 GHz and 4 GB RAM. To use BDDs we apply JavaBDD[2], which provides a native interface to the classical C++ library CUDD[3].

We transfered about 20 single- and two-player games from GDL to GDDL. Some of the games cannot be solved due to their size: either they cannot be instantiated (e.g., connectFour and nineMenMorris) due to the need for too much memory, or they cannot be transformed to BDDs (e.g., queens and endgame). For the latter the main problem is the number of derived actions: when trying to delete them the new domain description becomes too big to fit into main memory. Most of the games that could be instantiated and transformed to BDDs are shown in Table 1 along with the results of the reachability analysis (n and s represent the number of BDD nodes and the number of reachable states).

Peg (or *Solitaire*) is the most complex one of all the transfered single-player games: a total of more than 375 million states is reachable, while 3.5 million nodes suffice to rep-

[2] http://javabdd.sourceforge.net

[3] http://vlsi.colorado.edu/~fabio/CUDD/

Game	n	s	Game	n	s
8-Puz.	155,722	6,980,353	clob (3x4)	4,878	13,343
15-Puz.	2,055,704	9,251,016	clob (4x5)	471,456	26,787,440
blocks	110	42	minichess	3,151	4,573
hanoi	932	5,504	nim40	18	80
peg	3,501,604	375,110,246	tictactoe	625	5,478

Table 1: Single- and two-player games.

Pegs remaining	Gain	n	s
1 (in the middle)	100	1,835,093	26,856,243
1 (somewhere else)	99	70	4
2	90	7,321,698	134,095,586
3	80	7,022,261	79,376,060
4	70	6,803,498	83,951,479
5	60	3,589,371	25,734,167
6	50	2,309,661	14,453,178
7	40	1,266,697	6,315,974
8	30	651,352	2,578,583
9	20	338,281	1,111,851
10	10	166,229	431,138
> 10	0	94,094	205,983

Table 2: Partitioning of peg (n is the number of nodes, s the number of states in the corresponding bucket).

resent them. The classification is shown in Table 2. The reachability analysis took about 27 minutes, while the total running time was more than 7.5 hours.

Most of the two-player zero-sum games that can be solved by the old classification algorithm can also be solved by the new one. This has a worse runtime which can be seen even with small games: in our implementation of *Nim* we have only one row of the specified number of matches and each player may take one up to three matches at his turn. For the situation of 40 matches, the exponential blowup becomes apparent: While the old algorithm takes only four seconds to classify it, the new one takes 80 minutes.

Of all the two-player games that could be instantiated and transformed to BDDs, *Clobber*[4] is the most interesting one, as with the bigger instances its state space becomes huge. Here the effect of BDDs can be seen again: In the case of a 4×5 board, a total of nearly 26.8 million states is reachable while less than half a million nodes suffice to represent them. The classification takes about nine hours with the old algorithm; the new one did not finish after 20 days.

The new algorithm can handle the general gains provided with the GDL. So we can not only determine who will win a game, but also what gain can be achieved. For the 3×4 instance of Clobber we give gains according to Table 3. The older classification algorithm takes about 13, the new one nearly 200 seconds (with only gains 0 and 100 for each player). The classification with the depicted gains runs even longer than an hour. In the end the initial state resides in bucket $(0, 40)$, i.e., in optimal play the final situation is both players having two tokens left on the board, a result the old algorithm could not provide.

[4]http://homepages.gac.edu/∼wolfe/games/clobber/

number of tokens (white:black)	gain white	gain black	n	s
4 : 4	0	70	330	38
4 : 3	70	0	305	49
3 : 3	0	55	3,268	1,544
3 : 2	55	0	3,205	1,509
2 : 2	0	40	7,090	6,581
2 : 1	40	0	3,955	1,853
1 : 1	0	25	3,733	1,769

Table 3: Classification for 3×4-Clobber.

Conclusion and Future Work

With the game domain description language, game playing has eventually approached classical AI planning. In this paper we presented set-based exploration algorithms to solve general games with limited memory.

In the future we will extend our algorithm to games with more than two players and try to find a way to support games with simultaneous moves. We will also try to improve the algorithm that instantiates the domains and problems so that we may be able to instantiate more complex games. Another goal is an automated translation from GDL to GDDL.

References

Bryant, R. E. 1985. Symbolic manipulation of boolean functions using a graphical representation. In *ACM/IEEE Design Automation Conference (DAC)*, 688–694.

Cimatti, A.; Roveri, M.; and Traverso, P. 1998. Automatic OBDD-based generation of universal plans in non-deterministic domains. In *AAAI*, 875–881.

Edelkamp, S., and Helmert, M. 1999. Exhibiting knowledge in planning problems to minimize state encoding length. In *ECP*.

Edelkamp, S., and Helmert, M. 2001. Mips: The model-checking integrated planning system. *AI Magazine* 22(3):67–72.

Edelkamp, S., and Kissmann, P. 2007. Symbolic exploration for generalized game playing in PDDL. In *ICAPS-Workshop on Planning in Games*.

Edelkamp, S. 2002. Symbolic exploration in two-player games: Preliminary results. In *AIPS-Workshop on Model Checking*.

Hoffmann, J., and Edelkamp, S. 2005. The deterministic part of IPC-4: An overview. *JAIR* 24:519–579.

Jensen, R. 2003. *Efficient BDD-based planning for non-deterministic, fault-tolerant, and adversarial domains*. Ph.D. Dissertation, Carnegie-Mellon University.

Kautz, H., and Selman, B. 1996. Pushing the envelope: Planning, propositional logic, and stochastic search. In *AAAI*, 1194–1201.

Kissmann, P. 2007. Externalisierung des Sequenzenalignierungsproblems. Diploma Thesis, University of Dortmund.

Love, N. C.; Hinrichs, T. L.; and Genesereth, M. R. 2006. General game playing: Game description language specification. Technical Report LG-2006-01, Stanford Logic Group.

McDermott, D. 2000. The 1998 AI Planning Competition. *AI Magazine* 2(2):35–55.

Rapoport, A. 1966. *Two-Person Game Theory*. University of Michigan Press.

Schaeffer, J.; Björnsson, Y.; Burch, N.; Kishimoto, A.; and Müller, M. 2005. Solving checkers. In *IJCAI*, 292–297.

Schaeffer, J. 2000. The games computers (and people) play. In Zelkowitz, M., ed., *Advances in Computers*, volume 50, 189–266. Academic Press.