Symbolic Exploration for General Game Playing in PDDL

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

Abstract

This paper studies the application and extension of planning technology for general game playing.

First, we study the transformation of the general game playing language GDL into PDDL with the help of domain constants and derived predicates (level 1 of PDDL2.2) such that the PDDL input can be instantiated using existing static analysis tools. We discuss the differences in the conjunctive representation of the transition relation (in GDL) and its disjunctive form (in PDDL).

Next, we present symbolic exploration algorithms for one- and two-player games to fully characterize optimal play. So far, the algorithms assume alternating moves by the players, but extend existing classification algorithms for zero-sum games to the more general cost model imposed in GDL.

We evaluate the approach on a range of PDDL benchmarks matching the ones in the game playing community and show current limits and possibilities.

Introduction

In the field of artificial intelligence (AI), two- and multi-player 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. In other words, the AI designer does not know anything about the rules. Best policies result in perfect play. The opponent(s) attempt to maximize their individual outcome.

Games can be represented by game trees, which are often depth-bounded. The values at the leaf nodes of the trees are computed by a static evaluation function. 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 by illustrating how close their input languages actually are. Moreover, we illustrate how state-of-the-art symbolic planning technology can be applied to general game playing.

The paper is structured as follows. First, we introduce GDL and PDDL and the mapping between the two. In the next section, we describe existing and novel symbolic algorithms to solve single- and two-player games. While we have no restrictions for the single-player games, the two-player games need to be alternating. We provide results on a large subset of general game playing domains.

Description Languages

We briefly review the origins of the two formalisms.

GDL The game description language (GDL) (Love, Hinrichs, & Genesereth 2006) is designed for use in defining complete information games. It is a subset of first order logic, using syntax from the knowledge interchange format (KIF) language[1]. 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. Since 2005 there have been annual competitions. The 2005 competition was won by Jim Clune with *Cluneplayer*, the 2006 winners were Stephan Schiffel and Michael Thielscher with *Fluxplayer*. While their aim was to win the competition in match play with tuned algorithms, we concentrate on solving general games. This might take some time, often even much longer than the startup- and move-times in a competition. But to the best of the authors' knowledge no solver for general games exists.

PDDL The planning domain definition language (PDDL) is the standard language for the encoding of planning domains. The original version of the language was developed

[1]http://logic.stanford.edu/kif/dpans.html

Copyright © 2007, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

by McDermott (2000). Since then there have been competitions every second year. Fahiem Bacchus (2001) selected a subset of the original language as the language for the 2000 competition. In PDDL2.1, Long & Fox (2003) then extended the competition language to allow for numerical variables and concurrent execution of actions. PDDL2.1 is divided into different levels. Level 1 is propositional planning, level 2 adds numbers, level 3 adds durational actions. On top of PDDL2.1, two constructs were added in PDDL2.2 (Hoffmann & Edelkamp 2005) including game domain axioms in form of derived predicates.

Translation and Extension

Since all games in GDL are finite, it is possible, in principle, to describe games in the form of trees. Unfortunately, such explicit representations are not practical. Therefore, we propose a translation from GDL into GDDL (for *game domain description language*) [2] as an extension of PDDL. GDDL matches the syntax of PDDL2.2, level 1, with the exception that the evaluation of goals is handled differently. One additional construct is needed:

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

With `number` being a bounded integer in $\{0, \ldots, 100\}$, `body` being a goal description, and `parameters` being a typed list. To allow existing PDDL parsers (like *Adl2Strips* by Hoffmann) to handle the extended input, specialized actions embed the gain in their name. Currently all games are translated by hand. In Schiffel & Thielscher (2006) some aspects of such a translation are considered.

As an example, we consider the well-known two-player game TicTacToe. Figure 1 illustrates that the translation of GDL into GDDL of the initial and terminal positions is one-to-one. Using derived predicates with domain constants and existential quantification, game domain axioms translate straight-forwardly from GDL to GDDL (see Figure 2).

The core difference between GDL and GDDL is that the specification of moves is conjunctive for GDL, while in GDDL the moves are specified in a disjunctive form. Figure 3 displays the difference for the TicTacToe game.

Symbolic Exploration

Symbolic planning is based on checking the satisfiability of formulas (Kautz & Selman 1996). Here, we refer to symbolic exploration only in the context of using Binary Decision Diagrams (BDDs) (Bryant 1986). These contribute to many successful AI planning systems (Cimatti, Roveri, & Traverso 1998; Jensen 2003; Edelkamp & Helmert 2001). Compared to the space requirements of explicit-state planners, symbolic planning systems save space by exploiting a shared representation of state sets. 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

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

```
(init (cell 1 1 b))            (:init
...                              (cell r1 c1 b)
(init (cell 3 3 b))              ...
(init (control xplayer))         (cell r3 c3 b)
                                 (control xplayer))

(<= terminal (line x))         (:goal
(<= terminal (line o))           (or
(<= terminal (not open))           (line x) (line o)
                                   (not (open))))

(<= (goal xplayer 100)         (:gain ?player - role 100
  (line x))                      (and
                                   (= ?player xplayer)
                                   (line x)))

(<= (goal xplayer 50)          (:gain ?player - role 50
  (not (line x))                 (and
  (not (line o))                   (= ?player xplayer)
  (not open))                      (not (line x))
                                   (not (line o))
                                   (not (open))))

(<= (goal xplayer 0)           (:gain ?player - role 0
  (line o))                      (and
                                   (= ?player xplayer)
                                   (line o)))
```

Figure 1: Encoding initial and terminal game positions and the gains for the xplayer of TicTacToe in GDL (left) and GDDL (right).

Explicit-State Concept	Symbolic Concept
Search Frontier	Function *front*(S)
Expanded States	Function *reach*(S)
Initial State(s)	Function *init*(S)
Goal	Function *goal*(S)
Action a	Relation $t_a(S, S')$
Action Set	Relation $t(S, S')$

Table 1: Concepts in explicit-state and symbolic search.

of a set of successors, or the *image*, in a specialized operation. As a byproduct, the functional representation of the set of predecessors, or the *preimage*, can also be efficiently determined. Table 1 relates the concepts for explicit-state and symbolic search for variable sets S and S'. Individual relations $t_a(S, S')$ maintain $t(S, S') = \bigvee_a t_a(S, S')$ in a partitioned form.

Throughout this paper we use two sets of variables, S and S', to denote the precondition and effect variables, respectively. During forward search it is necessary to replace the precondition variables by the effect variables, which is achieved by the procedure *replace*. This way we take some states that were the effects of a transition and make them the preconditions for the next step. For backward search this replacement is the other way around.

The automated inference of a minimized state encoding of a propositional planning problem in PDDL refers to work of 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 instan-

```
(<= (row ?m ?x)
    (true (cell ?m 1 ?x))
    (true (cell ?m 2 ?x))
    (true (cell ?m 3 ?x)))
...
(<= (diagonal ?x)
  (true (cell 1 1 ?x))
  (true (cell 2 2 ?x))
  (true (cell 3 3 ?x)))
...
(<= (line ?x)
  (row ?m ?x))

...
(<= (line ?x)
  (diagonal ?x))
(<= open
  (true (cell ?m ?n b)))
```

```
(:derived (row ?m - row ?x - tok)
  (and (cell ?m c1 ?x)
       (cell ?m c2 ?x)
       (cell ?m c3 ?x)))
...
(:derived (diagonal ?x - tok)
  (and (cell r1 c1 ?x)
       (cell r2 c2 ?x)
       (cell r3 c3 ?x)))
...
(:derived (line ?x - tok)
  (exists (?m - row)
    (row ?m ?x)))
...
(:derived (line ?x - tok)
  (diagonal ?x))
(:derived (open)
  (exists (?m - row ?n - col)
    (cell ?m ?n b)))
```

Figure 2: Encoding game domain axioms of TicTacToe in GDL (left) and GDDL (right).

```
(<= (next (cell ?m ?n x))
  (does xplayer (mark ?m ?n))
  (true (cell ?m ?n b)))
(<= (next (cell ?m ?n ?w))
  (true (cell ?m ?n ?w))
  (distinct ?w b))
(<= (next (cell ?m ?n b))
  (does ?w (mark ?j ?k))
  (true (cell ?m ?n b))
  (or (distinct ?m ?j)
      (distinct ?n ?k)))
(<= (next (control oplayer))
  (true (control xplayer)))
(<= (legal ?w (mark ?x ?y))
  (true (cell ?x ?y b))
  (true (control ?w)))
```

```
(:action mark
:parameters
  (?player - role
   ?x - row ?y - col
   ?t - tok ?nplayer - role)
:precondition
(and
  (cell ?x ?y b)
  (control ?player)
  (= ?player xplayer)
  (= ?t x)
  (= ?nplayer oplayer)
:effect
(and
  (not (cell ?x ?y b))
  (cell ?x ?y ?t)
  (not (control ?player))
  (control ?nplayer)))
```

Figure 3: Encoding moves of the xplayer of TicTacToe in GDL (left) and GDDL(right).

tiated 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

In general, 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 that successively takes the set *front* of all positions in the current iteration and applies the transition relation to find the set of all *to* positions in the next iteration.

Starting with the initial state, successor states are generated until a fixpoint is reached. As in GDL the game ends once a terminal state is reached, only non-goal states from *from* are expanded. The union of all new positions is stored

1 reach ← init;
2 front ← reach;
3 **repeat**
4 to ← front ∧ replace(¬goal, $S' \rightarrow S$);
5 to ← relProd(t, to, S');
6 to ← replace(to, $S' \rightarrow S$);
7 front ← to ∧ ¬reach;
8 reach ← reach ∨ front;
9 **until** *front* = *false* ;
10 **return** *reach*

Algorithm 1: reachable

in the set *reach*. When no new state is generated, i.e., all states reachable from the initial state are created, the algorithm terminates. The pseudo-code is shown in Algorithm 1.

The forward calculation of the successors is done by the function $\mathit{relProd}\,(t, to, S') = \exists S\,(t\,(S, S') \wedge to\,(S))$. As the transition relation t is the disjunction of all actions t_a, we can calculate the *relProd* of each action and calculate the disjunction of all images afterwards. The search frontier is updated to the successors without the states that have already been reached. Thus the successors are determined only once for each state.

When the search frontier runs empty, *reach* contains all states that are reachable from the initial state.

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 are able to give a strategy for each state that leads to the goal that achieves the calculated gain.

In contrast to reachability analysis, the direction of the search process is *backwards*. Fortunately, backward search causes no problem, since the representation of all moves has already been defined as a relation.

The player aims at miximizing the received gain. Thus, we first calculate all states that lead to a maximal gain of 100 by starting at the reachable goal states that achieve a gain of 100 a 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.

The complete procedure is shown in Algorithm 2. First of all, we determine all reachable states as presented in Algorithm 1. Next we look only at those gains for which a BDD is present. The BDD *gain*$[i][j]$ represents the situation needed for player i to get gain j. Of all the states described by the BDD we take only those that are reachable goal states.

Then we calculate the predecessors of the states in the search frontier and remove those states that are already in a bucket of higher value. The new states are stored in the search frontier *front* and also added to the set of all reachable states leading to the specified gain (*gainGoal*). Once the search frontier runs empty, all states for this bucket have been determined and we can continue with the next.

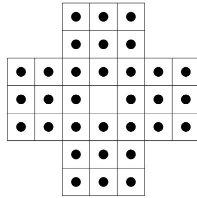
When all buckets are properly filled, the user might want to know which gain a given state might lead to in optimal

```
1  reach ← reachable();
2  partition ← false;
3  for i ← 100 to 0 do
4    if gain[1][i] ≠ false then
5      gainGoal ← gain[1][i] ∧ reach ∧ goal;
6      front ← gainGoal;
7      repeat
8        to ← replace(from, S→S′);
9        to ← reach ∧ relProd(t, to, S) ∧ ¬partition;
10       front ← to ∧ ¬gainGoal;
11       gainGoal ← gainGoal ∨ front;
12     until front = false ;
13     states[i] ← gainGoal;
14     partition ← partition ∨ states[i];
15   endif
16 endfor
```

Algorithm 2: partitionStatesSinglePlayer

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.

As an example we choose the well-known single-player game *Peg* (also known as *Solitaire*), whose initial state is shown. On the board 32 pegs are located at the depicted locations. The player can move one peg by jumping over an occupied field onto an empty one. This jump may only be performed in horizontal or vertical direction. The peg then moves to the formerly empty field, leaving the field it started its jump empty. The peg that was jumped over is deleted from the board leaving its field empty, too. The game ends when no more jumps are possible. As with each jump one peg is removed, this situation arises after 31 jumps at the latest. The main goal is to remove all pegs except for one. This then should be located in the very middle (i.e., we aim at the inverse of the initial state). This situation receives 100 points. We also give certain points for other final states: 99 points for one remaining peg that is not in the middle, 90, ..., 10 points for $2, \dots, 10$ pegs remaining, respectively, and 0 points for more than 10 pegs still on the board.



Classification of Two-Player Games

Two-player games with perfect information are classified iteratively. As in the case for the partitioning of single-player games, we also use backward search.

Even for zero-sum games the partitioning of the states is involved. Assuming optimal play and starting with all won positions of one player all previous winning positions have to be computed. A position is lost if all moves lead to an intermediate winning position in which the other player can force a move back to a lost one.

In single-player games we could remove the states also leading to higher gains from the buckets representing lower gains. But in two-player games it might not be clear, what really is higher or lower. For example a state might lead to

```
1  reach ← reachable();
2  foreach player do
3    selfLose ← reach ∧ gain[player][0] ∧ replace(goal,
     S′→S);
4    from ← selfLose;
5    opponentWin ← selfLose;
6    repeat
7      to ← replace(from, S→S′);
8      to ← opponentMove ∧ relProd(t, to, S) ∧ reach;
9      opponentWin ← opponentWin ∨ to;
10     to ← replace(opponentWin, S→S′);
11     to ← selfMove ∧ appAll(t, to, S) ∧ reach;
12     new ← to ∧ ¬selfLose;
13     from ← new;
14     selfLose ← selfLose ∨ new;
15   until new = false ;
16 endfch
```

Algorithm 3: classify (original version)

gain 100 for player 1 and 0 for player 2, but also to 0 for player 1 and 100 for player 2. For each player it is simple to say which of these buckets means the higher gain, but as both try to maximize their gain, it is not trivial to determine to which gains the state really leads.

We present two algorithms that overcome this problem. Both perform symbolic retrograde analysis for turn-taking games. These games are identified by the existence of a predicate *control-player* for each player. The first algorithm assumes that we have a two-player zero-sum game with possible gains of 100 and 0 for each player and maybe some draw states, giving gain 50 for both players. If player 1 receives gain 100, player 2 has to receive 0 and the other way around. Algorithm 3 shows this algorithm, which was originally posted in Edelkamp (2002).

First of all, we calculate all the reachable states. Next we construct four sets: The own lost states *selfLose* and the states won for the opponent *opponentWin* for each player. The own lost states and the opponent's winning states are initialized with the BDDs representing the reachable goal states in which the player receives a gain of 0. Now we construct the predecessors of the lost states. Here the move from the predecessor states to the current ones has to be made by the opponent. These predecessors are added to the won states of the opponent. Starting from those won states we further build their predecessors. The corresponding move has to be made by the current player. These new states are added to the lost states. If there are no new states at this point, the calculation for the current player ends.

Once the algorithm ends for both players, we can simply check in which set the initial state resides. If it is in one of the won states, the corresponding player can assure a victory; if it is in one of the lost states, the player surely looses. Otherwise the optimal outcome is a draw.

The simplest way to understand this algorithm is to have a look at the game-tree. We start the analysis at the leaves and propagate the gains towards the root. If one node v of the opponent has at least one successor u that is won for it, then v is also won for it, as it can assure victory by choosing

```
1  reach ← reachable();
2  forbidden ← ∅;
3  restart ← true;
4  repeat
5    if restart then
6      initialize matrix;
7      newBDDs ← |{(i, j) | matrix[i][j] ≠ false}|;
8      restart ← false;
9      step ← 0;
10   endif
11   foreach player do
12     matrix ← constructPredecessors(matrix, player,
       forbidden, newBDDs, reach);
13     forbidden ← deleteDuplicates(matrix, player,
       forbidden, step);
14     if duplicateDeleted then
15       restart ← true;
16       continue ;
17     endif
18     step ← step + 1;
19   endfch
20 until newBDDs = 0 ;
```

Algorithm 4: classify (new version)

the appropriate action. If one node v of the current player has no successor that is won for it, there is no way it can win along this path, so v also is lost. The latter is represented by the expression *appAll* $(t, to, S) = \forall S' \, (t\,(S, S') \Rightarrow to\,(S'))$, which can be partitioned and traced back to *relProd*:

$$\begin{aligned} & appAll\,(t, to, S) = \forall S' \,(t \Rightarrow to) \\ = \;& \neg\neg\forall S' \,((t_1 \vee \ldots \vee t_n) \Rightarrow to) \\ = \;& \neg\exists S' \,((t_1 \vee \ldots \vee t_n) \wedge \neg to) \\ = \;& \neg\exists S' \,(t_1 \wedge \neg to) \wedge \ldots \wedge \neg\exists S' \,(t_n \wedge \neg to) \\ = \;& \neg relProd\,(t_1, \neg to, S) \wedge \ldots \wedge \neg relProd\,(t_n, \neg to, S) \end{aligned}$$

Algorithm 4 is a bit more complex. Here we perform only one search after which we find the classification of the game. This algorithm is much more general in that it works well for the general gains, not only zero-sum games. To the best of our knowledge, no algorithm for general turn-taking two-player games with general gains has been published before.

We generate a 101×101-matrix of all possible gain combinations for the two players. The entry at bucket (i, j) is initialized as the conjunct of the BDD representing gain i for player 1 and the one representing gain j for player 2. We also need the number of BDDs (*newBDDs*) for which we still find new predecessors. Once there are no new predecessors in any bucket, the algorithm ends. Initially this is the number of BDDs in the matrix that are not *false*. Also important is the step number, which is initialized to 0.

We start by calculating the predecessors in which one player had control, i.e., it 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. Another possibility to create duplicates is that we reach some state that already is in one bucket along a different path. All these duplicates will be deleted and we perform a restart.

```
1  for i, j ← 0 to 100 do
2    if matrix[i][j] ≠ false then
3      to ← replace(matrix[i][j], S→S');
4      foreach item ∈ forbidden[i][j] do
5        matrix[i][j] ← relProd(t, to, S) ∧ selfMove ∧
         reach ∧ ¬(item.bdd);
6      endfch
7      if unchangedForTwoSteps(matrix[i][j]) then
8        newBDDs ← newBDDs - 1;
9      endif
10   endif
11 endfor
```

Algorithm 5: constructPredecessors

```
1  duplicateDeleted ← false;
2  total ← false;
3  for i ← 100 to 0 do
4    for j ← 0 to 100 do
5      if j ≠ 0 then
6        total ← total ∨ matrix[i][j − 1];
7      endif
8      if (total ∧ matrix[i][j]) ≠ false then
9        forbidden[i][j] ← forbidden[i][j] ∪
         {(step, total ∧ matrix[i][j])};
10       duplicateDeleted ← true;
11     endif
12   endfor
13 endfor
14 if duplicateDeleted then
15   for i, j ← 0 to 100 do
16     foreach item ∈ forbidden[i][j] do
17       if item.step < step then
18         forbidden[i][j] ← forbidden[i][j] \ item;
19       endif
20     endfch
21   endfor
22 endif
```

Algorithm 6: deleteDuplicates

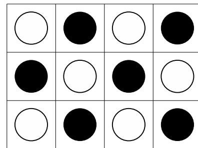
To maximize the gain for one player, we retain only those duplicate states that achieve highest gain for it. If then there are still duplicates, we delete all except for the ones that achieve the least gain for the opponent. This way we also, after securing maximal gain for the current player, minimize the gain for its opponent. Algorithm 6 shows this deletion of duplicates[3].

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 *forbidden*. One bucket of this matrix contains a list of BDDs representing the forbidden states for the corresponding bucket

[3]Here we show only the situation when the predecessors for the first player were created. In case of the second player, the indices of the matrix and the forbidden-matrix will be interchanged.

along with the step 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 (Algorithm 5, line 5). 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 step (Algorithm 6, from line 14). 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 the step number and continue, until no new states were generated. To check this, we have the number of newly created BDDs *newBDDs*, which will be updated each time we are finished with the calculation of the predecessors: If there were no new BDDs in a certain bucket for two steps, *newBDDs* will be decremented. Once the algorithm stops, we simply check in which bucket the initial state resides. This then represents the gain for both players.



Our running example, the game *Clobber*[4], was invented by M. H. Albert, J. P. Grossman and R. Nowakowski in 2001. It can be played on a board of any dimension though originally it was supposed to be played on a checkers board. The two players, white and black, have tokens all over the board. Usually these are alternating, i.e., a white token is always next to a black one. Such an initial position for Clobber on a 3×4 board is shown. Each player, starting with white, can move one of its tokens only to a horizontally or vertically adjacent field. To do this, the target field has to be occupied by a token of the opponent. This then is taken from the board and its position is occupied by the moved token, leaving its starting field empty. The game ends once no more moves are possible. During each move one token is taken from the board, thus the game ends after $n-1$ steps at the latest, assuming n is the total number or tokens. The game is won for the player performing the final move.

To demonstrate the working of the algorithm, we use the game Clobber on a 1×6 board with alternating colors starting with a white token in the leftmost place. We encode the states as follows. We have a row of six symbols: b for a black token, w for a white one and - for an empty field. This is followed by the player to move, W or B. Thus the initial state is wbwbwb W. The reachability analysis shows that there are 28 reachable states. Of these, 9 are goal states lost for the white player and 2 are goal states lost for the black player. These are depicted in Table 2.

The columns show the gain for the black player, the rows that for the white one. As only two buckets are occupied, we henceforth only depict those buckets: $(0, 100)$, which is gain 0 for the black and gain 100 for the white player, and

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

	0	100
0		b---w- W b-ww-b W -b---w W --w-b- W -w-b-- W -ww-bb W w-bb-w W ww-bb- W ww--bb W
100	b--w-w B w-w--b B	

Table 2: Goal states of 1×6-Clobber.

$(0, 100)$	$(100, 0)$	
b--w-w B	b---w- W	-ww-bb W
b--wwb W	b-ww-b W	w-bb-w W
b-wb-w W	-b---w W	ww-bb- W
w-bw-b W	--w-b- W	ww--bb W
w-w--b B	-w-b-- W	

Table 3: Example of 1×6-Clobber, step 1.

$(100, 0)$, in which the gains are the other way around.

First of all we calculate all predecessors with the white player having control. Only the two states lost for the black player have such predecessors. These are generated and added to the matrix, cf. Table 3.

As there are no duplicate states, we continue. Now we calculate the predecessors with the black player having control. Here new elements are created in both buckets, cf. Table 4.

The states wbwb-w and wbww-b are in both buckets. As the last move was done by the black player, we remove the duplicates from the $(0, 100)$-bucket. Now the algorithm starts over, but these states may not be created in the $(0, 100)$-bucket until we restart after a higher step. By doing this, we assure that their predecessors are not created. If we only remove the duplicate states, their predecessors might remain in the bucket and cause problems.

After restarting, we create the same states as before with the exception that in step 2, the states wbwb-w and wbww-b will not be created in bucket $(0, 100)$. In step 3 (cf. Table 5), the initial state wbwbwb is created in both buckets. As this was a move by the white player, we remove the duplicate from bucket $(100, 0)$ and restart. This time, we may not create the state wbwbwb in bucket $(100, 0)$. The two other states that were forbidden since the last restart may be generated again, as it might be possible that the newly forbidden state prevents the duplicates in bucket $(100, 0)$ from being created. However, after step 2 they are once again removed from bucket $(0, 100)$ and the algorithm restarts again, this time forbidding all three states in the corresponding buckets.

During the next iteration, we get to step 4, where the state wb-wwb, which was already present in bucket $(0, 100)$ is again created in bucket $(100, 0)$, as depicted in Table 6. The player to make the move is the black one, so we remove the state from bucket $(0, 100)$ and restart from the beginning; this time only forbidding the newly deleted state.

In the next iteration, we once again find states wbwb-w

$(0, 100)$	$(100, 0)$	
`b--w-w` B	`b---w-` W	`-ww-bb` W
`b--wwb` W	`b-ww-b` W	`wb--w-` B
`b-wb-w` W	`-b---w` W	`wbwb-w` B
`wb-wwb` B	`--w-b-` W	`wbww-b` B
`wbwb-w` B	`--w-wb` B	`w-bb-w` W
`wbww-b` B	`-wb--w` B	`ww-bb-` W
`w-bw-b` W	`-w-b--` W	`ww-bwb` B
	`-w-wb-` B	`ww--bb` W
	`-wwbwb` B	

Table 4: Example of 1×6-Clobber, step 2.

$(0, 100)$	$(100, 0)$	
`b--w-w` B	`b---w-` W	`-ww-bb` W
`b--wwb` W	`b-ww-b` W	`wbb--w` W
`b-wb-w` W	`-b---w` W	`wb--w-` B
`wb-wwb` B	`--w-b-` W	`wb-wb-` W
`wbwbwb` W	`--w-wb` B	`wbwb-w` B
`w-bw-b` W	`-wb--w` B	`wbwbwb` W
`w-w--b` B	`-wb-wb` W	`wbww-b` B
	`-w-b--` W	`w-bb-w` W
	`-w-wb-` B	`ww-bb-` W
	`-wwbb-` W	`ww-bwb` B
	`-wwbwb` B	`ww--bb` W

Table 5: Example of 1×6-Clobber, step 3.

and `wbww-b` to be withdrawn from bucket $(0, 100)$ after step 2. After that, the final iteration starts, during which none of the states `wbwb-w`, `wbww-b` and `wb-wwb` may be created in bucket $(0, 100)$. So after step 1, no new states are found for that bucket, and we no longer consider it during the calculation of predecessors (if we would restart again, we once again had to consider that bucket). Hence, the initial state `wbwbwb` will not be created in that bucket and thus not be deleted from the $(100, 0)$-bucket. In the end, this initial state is located in the latter one (cf. Table 7), so we are sure that the game is won for the black player.

Experimental Results

We developed a game based planner in Java implementing the above algorithms. Experiments were performed on an AMD Opteron processor with 2.3 GHz and 4 GB RAM. To use BDDs we apply JavaBDD[5], which provides a native interface to the classical C++ library CUDD[6]. We transfered about 20 single- and two-player games from GDL to GDDL.

Most single-player games and about half of the two-player games can be solved using our algorithms. The problem of the other games is their size: Either they could not be instantiated (e.g., pancake, connectFour, and nineMenMorris) due to the need for too much memory, or they could not be transformed to BDDs (e.g., queens and endgame). For the latter the main problem is the number of derived predicates: 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 8 along with the results of the reachability analysis.

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

[6]http://vlsi.colorado.edu/∼fabio/CUDD/

$(0, 100)$	$(100, 0)$	
`b--w-w` B	`b---w-` W	`-ww-bb` W
`b--wwb` W	`b-ww-b` W	`wbb--w` W
`b-wb-w` W	`-b---w` W	`wb--w-` B
`wb-wwb` B	`--w-b-` W	`wb-wb-` W
`wbwbwb` W	`--w-wb` B	`wb-wwb` B
`w-bw-b` W	`-wb--w` B	`wbwb-w` B
`w-w--b` B	`-wb-wb` W	`wbww-b` B
	`-w-b--` W	`w-bb-w` W
	`-w-wb-` B	`ww-bb-` W
	`-wwbb-` W	`ww-bwb` B
	`-wwbwb` B	`ww--bb` W

Table 6: Example of 1×6-Clobber, step 4.

$(0, 100)$	$(100, 0)$	
`b--w-w` B	`b---w-` W	`wbb--w` W
`b--wwb` W	`b-ww-b` W	`wb--w-` B
`b-wb-w` W	`-b---w` W	`wb-wb-` W
`w-bw-b` W	`--w-b-` W	`wb-wwb` B
`w-w--b` B	`--w-wb` B	`wbwb-w` B
	`-wb--w` B	`wbwbwb` W
	`-wb-wb` W	`wbww-b` B
	`-w-b--` W	`w-bb-w` W
	`-w-wb-` B	`ww-bb-` W
	`-wwbb-` W	`ww-bwb` B
	`-wwbwb` B	`ww--bb` W
	`-ww-bb` W	

Table 7: Example of 1×6-Clobber, final situation.

Peg is the most complex one of all the transfered games: A total of more than 375 million states is reachable, while 3.5 million nodes suffice to represent them all. The acquired results of Algorithm 2 are shown in Table 9. The reachability analysis took about 30 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. But this has a worse runtime, which can be seen even with small games. The classical Tic-Tac-Toe game takes less than a second to be fully classified by the old algorithm. The new one takes nearly 12 seconds. A similar situation arises with the game Nim. In our implementation of this we have only one row of the specified number of matches and each player may take one up to three matches at its turn. For the situation of 40 matches, the blowup becomes apparent: While the old algorithm takes only four seconds to classify it, the new one takes 1 hour, 20 minutes.

Of all the two-player games that could be instantiated and transformed to BDDs, Clobber 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 the 4×5 board, a total of more than 26.5 million states are reachable, but less than half a million nodes suffice to represent them all. The classification takes nearly nine hours with the original algorithm, while 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

Game	n	s	Game	n	s
8-Puz.	155,722	6,980,353	clobber3x4	4,878	13,343
15-Puz.	2,055,704	9,251,016	clobber4x5	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 8: Single- and two-player games (n and s represent the number of BDD nodes and the number of reachable states, respectively).

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

a game, but also what gain can be achieved. For the 3×4 instance of Clobber we give gains according to Table 10. The zero-sum classification takes about 13 seconds with the old algorithm and nearly 3 minutes, 20 seconds with the new one. The classification with the depicted gains runs even longer than an hour. In the end, the initial state resides in the bucket $(0, 40)$, which means that the final situation is both players having two tokens left on the board, something the old algorithm could not tell.

Conclusion

With the game domain description language, game playing has eventually approached classical AI planning. The paper has shown how to bridge the gap of general game playing and domain-independent action planning by illustrating how close their respective input languages GDL and PDDL actually are. Moreover, set-based exploration algorithms have been presented to solve general games with limited memory. As the presented approach for solving two-player games with costs is general, it might be extended to solve games with more than two players. To adapt the algorithm one has to increase the dimension of the cost (and forbidden) matrix and extend the loop to the number of players.

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 10: Gains for each player at 3×4-Clobber and the number of nodes and states in each bucket after classification.

References

Bacchus, F. 2001. The AIPS'00 planning competition. *AI Magazine* 22(3):47–56.

Bryant, R. E. 1986. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers* 35(8):677–691.

Cimatti, A.; Roveri, M.; and Traverso, P. 1998. Automatic OBDD-based generation of universal plans in non-deterministic domains. In *Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference (AAAI '98 / IAAI '98)*, 875–881. AAAI Press.

Edelkamp, S., and Helmert, M. 1999. Exhibiting knowledge in planning problems to minimize state encoding length. In Fox, M., and Biundo, S., eds., *Recent Advances in AI Planning. Fifth European Conference on Planning (ECP '99)*, volume 1809 of *Lecture Notes in Artificial Intelligence*, 135–147. Springer-Verlag.

Edelkamp, S., and Helmert, M. 2001. The model checking integrated planning system (MIPS). *AI Magazine* 22(3):67–71.

Edelkamp, S. 2002. Symbolic exploration in two-player games: Preliminary results. In *International Conference on AI Planning & Scheduling (AIPS '02), Workshop on Model Checking*, 40–48.

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

Jensen, R. M. 2003. *Efficient BDD-Based Planning for Non-Deterministic, Fault-Tolerant, and Adversarial Domains*. Ph.D. Dissertation, Carnegie Mellon University.

Kautz, H. A., and Selman, B. 1996. Pushing the envelope: Planning, propositional logic and stochastic search. In *Thirteenth National Conference on Artificial Intelligence and Eighth Innovative Applications of Artificial Intelligence Conference (AAAI '96 / IAAI '96)*, 1194–1201. Menlo Park: AAAI Press / MIT Press.

Long, D., and Fox, M. 2003. The 3rd international planning competition: Results and analysis. *Journal of Artificial Intelligence Research (JAIR)* 20:1–59.

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. V. 2000. The 1998 AI planning systems competition. *AI Magazine* 21(2):35–55.

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

Schaeffer, J.; Björnsson, Y.; Burch, N.; Kishimoto, A.; Müller, M.; Lake, R.; Lu, P.; and Sutphen, S. 2005. Solving checkers. In Kaelbling, L. P., and Saffiotti, A., eds., *Nineteenth International Joint Conference on Artificial Intelligence (IJCAI '05)*, 292–297. Professional Book Center.

Schaeffer, J. 2000. The games computers (and people) play. In *Seventeenth National Conference on Artificial Intelligence and Twelfth Innovative Applications of Artificial Intelligence Conference (AAAI '00 / IAAI '00)*, 1179. AAAI Press / MIT Press.

Schiffel, S., and Thielscher, M. 2006. Reconciling situation calculus and fluent calculus. In *Twenty-First National Conference on Artificial Intelligence and Eighteenth Innovative Applications of Artificial Intelligence Conference (AAAI '06 / IAAI '06)*. AAAI Press.