Layer-Abstraction for Symbolically Solving General Two-Player Games

Peter Kissmann and **Stefan Edelkamp**
TZI Universität Bremen, Germany
{kissmann,edelkamp}@tzi.de

Abstract

In this paper, we propose a new algorithm for solving general two-player turn-taking games that performs symbolic search utilizing BDDs. It consists of two stages: First, it determines all BFS layers using forward search and omitting duplicate detection, next, the solving process itself operates in backward direction only within these BFS layers. Thus, all BDDs are partitioned according to the layers the states reside in.

We provide experimental results for selected games and compare to a previous approach. This comparison shows that in most cases the new algorithm outperforms the existing one and thus we can solve games that could not be solved using a general approach before.

Introduction

In recent years, general game playing has received an increasing amount of attention. This might be due the annual general game playing competition (Genesereth, Love, and Pell 2005) that is held at AAAI or IJCAI since 2005. In general game playing the agents are provided a description of a game according to certain rules and need to play it. In case of multi-player games, the agents often play against each other, while in case of single-player games one agent tries to find a way to reach a terminal state where it can achieve the best reward possible. The authors of the agents do not know, which games will be played, so no domain specific knowledge can be inserted.

General single-player games match classical action planning problems (Fikes and Nilsson 1971) as for both the player (or the planner) intends to find a sequence of moves (or actions) that transforms the initial state to one of the terminal states. While nowadays in planning action costs as well as rewards for achieving soft goals can be combined, in general game playing the players only get rewards for achieving goals: for each possible terminal state the player is awarded points ranging from 0 (worst) to 100 (best).

General game playing also allows to express multi-player games. Problems from the non-deterministic planning extension of classical planning can be translated to a two-player game (with the planner being the player and the environment that controls the non-determinism its opponent) (Jensen, Veloso, and Bowling 2001; Bercher and Mattmüller 2008). General game playing supports any number of participants, thus it still is a generalization of action planning.

In this paper, we are interested in two-player turn-taking games, i. e., in games, where only one player may decide, which move to take in each state. The other one can only perform a noop, which does not change anything about the current game state. We also can handle games that are not strictly alternating, so that one player might be active in several consecutive states.

Our goal is to strongly solve the games, i. e., we want to find the outcome for each player in any reachable state in case of optimal play. Using domain dependent solvers, this has often been done in the past. One of the last prominent results was by Schaeffer et al. (2007), who were able to solve American Checkers after more than ten years and proved that the optimal outcome is a draw. Of course, without domain specific knowledge, we cannot expect to come up with solutions for such complex games in general game playing.

In explicit representation, many general games are too complex to fit into RAM or even on a hard disk. So, to solve them we perform symbolic search that utilizes binary decision diagrams (BDDs) (Bryant 1986) as they decrease the memory consumption, if a good variable ordering is found.

The paper is structured as follows. First, we give brief introductions to general game playing and symbolic search. Next, we propose our new algorithm to solve general two-player turn-taking games. Then, we show some experimental results where we also compare to an existing algorithm, and, finally, we present a short discussion, draw some conclusions and point out possible future research avenues.

General Game Playing

General game playing is concerned with playing games that need to be finite, discrete, and deterministic and must contain full information for all the players. It is possible to model single- as well as multi-player games, which by default are games with simultaneous moves by all players. They can be made turn-taking by adding a predicate that denotes whose turn it is to choose the next move and by allowing the other players to perform only noops, i. e., moves that do not change the game's current state. To describe these games, the logic-based game description language GDL (Love, Hinrichs, and Genesereth 2006) is used.

A general game is a tuple $\mathcal{G} = \langle \mathcal{S}, p, \mathcal{M}, \mathcal{I}, \mathcal{T}, \mathcal{R} \rangle$ with $\mathcal{S}$ being the set of reachable states, p the number of players, $\mathcal{M} \subseteq \mathcal{S} \times \mathcal{S}$ the set of possible moves for each state, $\mathcal{I} \in$

```
(role xplayer) (role oplayer)  ; names of the players

(init (cell 1 1 b)) ... (init (cell 3 3 b))  ; all cells empty
(init (control xplayer))  ; xplayer is active

(<= (next (cell ?m ?n x))          (<= (next (cell ?m ?n o))  ; effect of marking a cell
    (does xplayer (mark ?m ?n)))       (does oplayer (mark ?m ?n)))
(<= (next (cell ?m ?n ?w))  ; part of the frame (marked cells remain marked)
    (true (cell ?m ?n ?w)) (distinct ?w b))
(<= (next (cell ?m ?n b))  ; part of the frame (untouched empty cells remain empty)
    (does ?w (mark ?j ?k)) (true (cell ?m ?n b))
    (or (distinct ?m ?j) (distinct ?n ?k)))
(<= (next (control xplayer))   (<= (next (control oplayer)); change of the active player
    (true (control oplayer)))      (true (control xplayer)))

(<= (legal ?w (mark ?x ?y))  ; possible move (empty cell can be marked)
    (true (cell ?x ?y b)) (true (control ?w)))
(<= (legal xplayer noop)        (<= (legal oplayer noop)  ; if opponent active, no move
    (true (control oplayer)))       (true (control xplayer)))

; axioms (utility functions) for reducing the complexity of the description
(<= (row ?m ?x)
    (true (cell ?m 1 ?x)) (true (cell ?m 2 ?x)) (true (cell ?m 3 ?x)))
(<= (column ?n ?x)
    (true (cell 1 ?n ?x)) (true (cell 2 ?n ?x)) (true (cell 3 ?n ?x)))
(<= (diagonal ?x)
    (true (cell 1 1 ?x)) (true (cell 2 2 ?x)) (true (cell 3 3 ?x)))
(<= (diagonal ?x)
    (true (cell 1 3 ?x)) (true (cell 2 2 ?x)) (true (cell 3 1 ?x)))
(<= (line ?x) (row ?m ?x))  (<= (line ?x) (column ?m ?x))  (<= (line ?x) (diagonal ?x))

(<= (goal xplayer 100) (line x))  ; rewards for xplayer (oplayer analogously )
(<= (goal xplayer 50) (not (line x)) (not (line o)))
(<= (goal xplayer 0) (line o))

; terminal states
(<= terminal (line x))  (<= terminal (line o))  (<= terminal (not (true(cell ?m ?n b))))
```

Figure 1: GDL description of the game Tic-Tac-Toe.

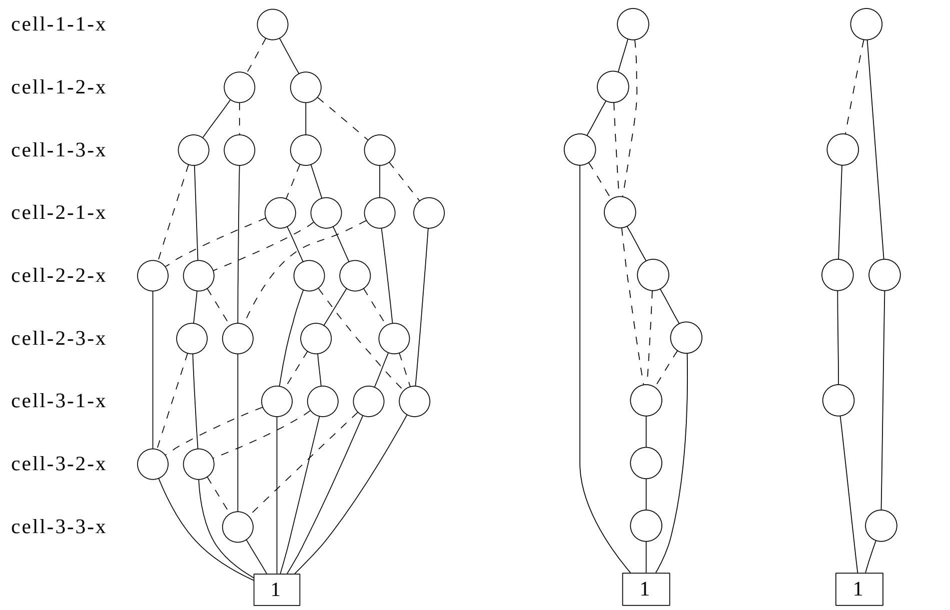
$\mathcal{S}$ the initial state, $\mathcal{T} \subseteq \mathcal{S}$ the set of terminal states, and $\mathcal{R} : \mathcal{T} \mapsto \{0, \ldots, 100\}^p$ the reward for each player in all terminal states. General games are defined implicitly, i. e., only the initial state is provided and we can calculate the set of reachable states $\mathcal{S}$ using the applicable moves. For turn-taking games there are subsets $\mathcal{S}_i \subseteq \mathcal{S}$ of states where player $i \in \{1, \ldots, p\}$ is active as well as subsets $\mathcal{M}_i \subseteq \mathcal{M}$ denoting those moves, where player $i \in \{1, \ldots, p\}$ is the only one to choose a move other than a noop.

Figure 1 shows the description of the game Tic-Tac-Toe. The players are denoted by the `role` keyword; the initial state $\mathcal{I}$ by the `init` keyword, the terminal state $\mathcal{T}$ by the `terminal` keyword and the rewards $\mathcal{R}$ by the `goal` keyword. The moves $\mathcal{M}$ are splitted into two parts, the `legal` formulas describing the preconditions necessary for a player to perform the corresponding moves, and the `next` formulas, which determine the successor state.

Playing a general game always starts at its initial state. All players choose one applicable move in the current state. These moves are combined and using the rules for this combined move, a successor state is generated. This goes on, until a terminal state is reached, where the game ends and the players receive their rewards according to $\mathcal{R}$.

This paper does not address playing general games but solving them strongly, i. e., we want to find the outcome of each reachable state in case of optimal play of both players. With this information, we can design a perfect player, or we can check played games for bad moves, which might give insight to weaknesses of certain agents. For some games we are not able to find a solution in reasonable time. However, we might use what was calculated so far as an endgame database for a player, e. g., one that utilizes UCT search (Kocsis and Szepesvári 2006), which is used in many successful players (e. g., in CADIAPLAYER (Finnsson and Björnsson 2008), the world champion of 2007 and 2008, as well as in Méhat's ARY, the current world champion).

Unfortunately, except for our precursing work, we are not aware of any other research in this area. Thus, in this paper we will compare to the better of our previous approaches (Edelkamp and Kissmann 2008b). Some general game players, e. g., Schiffel and Thielscher's FLUXPLAYER (2007), might also be able to solve simple games, but as they are designed for playing, it would be unfair to compare to these when solving is concerned.



(a) Column for xplayer. (b) Row for xplayer. (c) Diagonal for xplayer.

Figure 2: BDDs for the three utility functions of Tic-Tac-Toe used in the terminal states. Each node corresponds to a predicate (denoted on the left); solid edges mean that predicate is true, dashed edges mean it is false. The bottom-most node represents the 1-sink, i. e., all paths leading from the top-most node to this sink represent satisfied assignments. The 0-sink has been omitted for better readability.

Symbolic Search

When we speak of symbolic search we mean state space search using binary decision diagrams (BDDs) (Bryant 1986). With these, we can perform a set-based search, i. e., we do not expand single states but sets of states.

BDDs typically have a fixed variable ordering and are reduced using two rules so that only a minimal number of internal BDD nodes is needed to represent a given formula / set of states. The resulting representation also is unique and no duplicates are present in any BDD.

BDDs enable us to completely search some state spaces that would not be possible in explicit search. E. g., in the game Connect Four more than 4.5×10^{12} states are reachable. We use 85 bits to encode each state (two bits for each cell and an additional one to denote the active player), so that in case of explicit search we would need about 43.5 TB to store all of them, while with BDDs 16 GB are sufficient. If we store only the current breadth-first search (BFS) layer and flush the previous one to a hard disk, the largest one even fits into 12 GB.

For symbolic search, we need BDDs to represent the initial state $\mathcal{I}$, the terminal states $\mathcal{T}$, the formula describing when the players get which reward $\mathcal{R}$, as well as the moves $\mathcal{M}$. Unfortunately, most games contain variables, so that we do not know the exact size of a state, but this information is mandatory for BDDs. Thus, we instantiate the games (Kissmann and Edelkamp 2009) and come up with a variable-free format, similar to what most successful action planners in recent years do before the actual planning starts (Helmert 2008). As all formulas are Boolean, generating BDDs of these is straight-forward. Figure 2 shows BDDs for some of the utility functions needed to evaluate the termination of Tic-Tac-Toe.

To decrease the number of BDD variables, we try to find groups of mutually exclusive predicates. For this we perform a simulation-based approach similar to Kuhlmann, Dresner, and Stone (2006) and Schiffel and Thielscher (2007) who identify the input and output parameters of each predicate. Often, input parameters denote the positions on a game board while the output parameters specify its content. Predicates sharing the same name and the same input but different output parameters can never be true at the same time and thus are mutually exclusive. If we find a group of n mutually exclusive predicates, we need only $\lceil \log n \rceil$ BDD variables to encode these.

After instantiation, we know the precise number of moves of all the players and can also generate the possible combinations of moves of all players, which results in $\mathcal{M}$. Each move $m \in \mathcal{M}$ can be represented by a BDD $trans_m$, so that the complete transition relation $trans$ is the disjunction of all these: $trans := \bigvee_{m \in \mathcal{M}} trans_m$.

To perform symbolic search, we need two sets of variables: one set, S, for the current states, the other one, S', for the successor states. To calculate the successors of a state set *from*, in symbolic search we use the $image$ operator:

$$image\,(from) := \exists S.\,(trans\,(S, S') \wedge from\,(S))\,.$$

As these successors are represented using only S', we need to swap them back to S.[1] This way, if we start at the initial

[1] We omit the explicit mention of this in the pseudo-codes to en-

state, each call of the image results in an entire BFS layer. So, BFS is simply the iteration of the image, until a fix-point is reached.

As the transition relation $trans$ is the disjunction of a number of moves, it is equivalent to generate the successors using one move after the other and afterwards calculate the disjunction of all these states:

$$image\,(from) := \bigvee_{m \in \mathcal{M}} \exists S.\,(trans_m\,(S, S') \wedge from\,(S))\,.$$

This way, we do not need to calculate a monolithic transition relation, which takes time and often results in a BDD too large to fit into RAM.

The inverse operation of the image is also possible. The $pre\text{-}image$ results in a BDD representing all the states that are predecessors of the given set of states $from$:

$$pre\text{-}image\,(from) := \exists S'.\,(trans\,(S, S') \wedge from\,(S'))\,.$$

With this, we can perform a BFS in backward direction as well.

Solving General Two-Player Turn-Taking Games

In this section we show an algorithm to solve general two-player turn-taking games symbolically with only images and pre-images, whereas in our existing approach (Edelkamp and Kissmann 2008b) we also use strong pre-images.

The existing approach works by using a 101×101 matrix M of BDDs. The BDD at $M\,[i, j]$ represents the states where player 1 can achieve a reward of i and player 2 a reward of j, $i, j \in \{0, \ldots, 100\}$. Initially, all terminal states are inserted in the corresponding buckets. Starting at these, the strong pre-image is used to calculate those preceding states whose successors are all within the matrix and thus already solved. These predecessors are then sorted into the matrix by using the pre-image from each of the buckets in a certain order.

The new algorithm works in two stages. First, we perform a symbolic BFS in forward direction (see Algorithm 1). Starting at the initial state, we calculate the successors of the current BFS layer by using the image operator. In contrast to the existing approach where a BFS was used to calculate the set of reachable states, here we retain only the BFS layers to partition the BDDs according to the layers the states reside in, hoping that the BDDs will keep smaller.

For the game Tic-Tac-Toe we start with the empty board. After one iteration through the loop, $curr$ contains all states with one x being placed on the board; after the next iteration all states with one x and one o being placed and so on.

Unfortunately, for the second step to work correctly we need to omit duplicate detection (except for the one that implicitly comes with using BDDs). The search will terminate nonetheless, as the games in general game playing are finite by definition, but it might be possible to expand states more than once, if they appear on different paths in different layers.

hance readability. Whenever we write of an image (or pre-image), we assume such a swapping to be performed immediately after the image (or pre-image) itself.

Algorithm 1: Calculate Reachable States ($reach$).

Input: General game description $\mathcal{G}$.
Output: Maximal reached BFS-layer.

$curr \leftarrow \mathcal{I}$;
$l \leftarrow 0$;
while $curr \neq \bot$ **do**
  store $curr$ as layer l on disk;
  $prev \leftarrow curr \wedge \neg\mathcal{T}$;
  $curr \leftarrow image\,(prev)$;
  $l \leftarrow l + 1$;
end while
return l;

A question that immediately arises is, when we will have to deal with such duplicate states. To answer this, we need to define a progress measure.

Definition 1 ((Incremental) Progress Measure)**.** *Let $\mathcal{G}$ be a general two-player turn-taking game and ψ be a mapping from states to numbers, i. e., $\psi : \mathcal{S} \mapsto \mathbb{N}$.*

If $\mathcal{G}$ is not necessarily alternating, ψ is a progress measure *if $\psi\,(s) < \psi\,(s')$ for all $(s, s') \in \mathcal{M}$. It is an* incremental progress measure*, if $\psi\,(s) = \psi\,(s') - 1$.*

Otherwise, ψ also is a progress measure*, if $\psi\,(s) = \psi\,(s') < \psi\,(s'')$ for all $(s, s') \in \mathcal{M}_1$ and $(s', s'') \in \mathcal{M}_2$. It is an* incremental progress measure*, if $\psi\,(s) = \psi\,(s') = \psi\,(s'') - 1$.*

Theorem 1 (Duplicate Avoidance)**.** *Whenever there is an incremental progress measure ψ for a general game $\mathcal{G}$, no duplicate arises across the layers found by Algorithm 1.*

Proof. We need to show this for the two cases:

If $\mathcal{G}$ is not necessarily alternating, we claim that all states within one layer have the same progress measurement but a different one from any state within another layer, which implies the theorem. This can be shown by induction: The first layer consists only of the initial state. Let $succ\,(s)$ be the set of successor states of s, i. e., $succ\,(s) = \{s' \mid (s, s') \in \mathcal{M}\}$. According to the induction hypothesis, all states in layer l have the same progress measurement. For all states s in layer l and successors $s' \in succ\,(s)$, $\psi\,(s') = \psi\,(s) + 1$. All successors $s' \in succ\,(s)$ are inserted into layer $l + 1$, so that all states within layer $l + 1$ have the same progress measurement. It is also greater than that of any of the states in previous layers, as it always increases between layers, so that it differs from the progress measurement of any state within another layer.

Otherwise, the states within any succeeding layers differ, as the predicate denoting the active player has changed. Thus, it remains to show that for all $s, s' \in \mathcal{S}$, $s_1 \in \mathcal{S}_1$ and $s_2 \in \mathcal{S}_2$, $\psi\,(s) = \psi\,(s')$ if s and s' reside in the same layer and $\psi\,(s_1) = \psi\,(s_2)$ if s_1 resides in layer l and s_2 resides in layer $l + 1$. For all other cases, we claim that the progress measurement of any two states does not match, which proves the theorem. The proof is very similar: The first layer consists only of the initial state. All successors of this state reside in the next layer and their progress measure equals, according to the definition of ψ. Let l be a

Algorithm 2: Solving General Two-Player Games

Input: General game description $\mathcal{G}$.
1 $l \leftarrow \mathit{reach}\,(\mathcal{G})$;
2 **while** $l \geq 0$ **do**
3   $\mathit{curr} \leftarrow$ load BFS layer l from disk;
4   $\mathit{currTerms} \leftarrow \mathit{curr} \wedge \mathcal{T}$;
5   $\mathit{curr} \leftarrow \mathit{curr} \wedge \neg \mathit{currTerms}$;
6   **for each** $i, j \in \{0, \ldots, 100\}$ **do**
7     $\mathit{terms}_{l,i,j} \leftarrow \mathit{currTerms} \wedge \mathcal{R}_{i,j}$;
8   store $\mathit{terms}_{l,i,j}$ on disk;
9     $\mathit{currTerms} \leftarrow \mathit{currTerms} \wedge \neg \mathit{terms}_{l,i,j}$;
10   **end for**
11   **for each** $i, j \in \{0, \ldots, 100\}$ **do in specific order**
12     $\mathit{succ}_1 \leftarrow$ load $\mathit{terms}_{l+1,i,j}$ from disk;
13     $\mathit{succ}_2 \leftarrow$ load $\mathit{rewards}_{l+1,i,j}$ from disk;
14     $\mathit{succ} \leftarrow \mathit{succ}_1 \vee \mathit{succ}_2$;
15     $\mathit{rewards}_{l,i,j} \leftarrow \mathit{curr} \wedge \mathit{pre\text{-}image}\,(\mathit{succ})$;
16   store $\mathit{rewards}_{l,i,j}$ on disk;
17     $\mathit{curr} \leftarrow \mathit{curr} \wedge \neg \mathit{rewards}_{l,i,j}$;
18   **end for**
19   $l \leftarrow l - 1$;
20 **end while**

layer that contains only states from $\mathcal{S}_1$. According to the induction hypothesis, all states in this layer have the same progress measurement. For all states s in layer l and successors $s' \in \mathit{succ}\,(s)$, $\psi\,(s) = \psi\,(s')$. All successors s' are inserted into layer $l+1$. For all states s' in layer $l+1$ and $s'' \in \mathit{succ}\,(s')$, $\psi\,(s'') = \psi\,(s') + 1$. All successors $s'' \in \mathit{succ}\,(s')$ are inserted in layer $l+2$, so that all states within layer $l+2$ have the same progress measurement. It is also greater than that of any of the states in previous layers, as it never decreases, so that it differs from the progress measurement of any state within different layers.

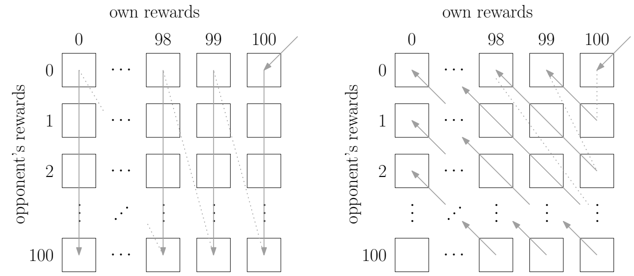
□

Note that in games that do not incorporate an incremental progress measure we need to expand each state at most d_{max} times, with d_{max} being the maximal distance from the initial state to one of the terminal states. This is due the fact that in such a case each state might reside in every layer.

Once all BFS layers are calculated we can start the second stage, the actual solving process, for which we perform a symbolic retrograde analysis (see Algorithm 2). We start at the last generated BFS layer l and move upwards layer by layer until we reach the initial state ($l = 0$).

For each layer we perform two solving steps. First, we calculate all the terminal states that are contained in this layer (line 4). For these we then determine the rewards that the players get and store them in the corresponding files (lines 6 to 10). As each player achieves exactly one reward for each possible terminal state, no specific order is needed in this step.

In the second step, we solve the non-terminal states. For this, we need to proceed through all possible reward combinations in a specific order (line 11). This order corresponds to an opponent model. The two most reasonable assumptions are that an agent either wants to maximize its own reward or to maximize the difference to the opponent's reward. The order, in which these reward combinations are processed, is indicated in Figure 3. For the experiments we assumed both players to be interested in maximizing the difference to the opponent's reward.



(a) Maximizing own reward. (b) Maximizing difference.

Figure 3: Order to traverse the reward combinations.

The solving of the non-terminal states is depicted in lines 11 to 18. We load the BDDs representing the states that are terminal states or solved non-terminal states in the successor layer for which the players can surely achieve the corresponding rewards. From the disjunction of these we calculate their predecessors (using the pre-image). These states achieve the same rewards (in case of optimal play according to the opponent model) and thus can be stored on disk and must be removed from the unsolved states to prevent them from being assigned other rewards as well.

For the game Tic-Tac-Toe we start in layer 9, where all cells are filled. All these states are terminal states, thus we can solve them immediately by checking the rewards, so that we partition this layer into two parts: Those states, where `xplayer` gets 100 points and `oplayer` 0 (the last move established a line of `x`s), and those with 50 points for each player (no line for any player).

In the next iteration, we reach those states where four `x`s and four `o`s reside on the board and the `xplayer` has control. First, we remove the states containing a line for one of the players (which can be only a line of `o`s, as it was the `oplayer`'s turn to perform the last move), as these are the terminal states, and solve them according to their rewards (for all these, the `xplayer` will get 0 points, while the `oplayer` gets 100.

Next, we check how to solve the remaining states. For it we start by loading the terminal states from layer 9 where the `xplayer` achieved 100 points, calculate their predecessors and verify, if any of these predecessors is present in the set of the remaining states. If that is the case, we can remove them and store them in a file that specifies that the `xplayer` achieves 100 points and the `oplayer` 0 points for these states as well. In the Tic-Tac-Toe example, these are all the states where the placement of another `x` finishes a line. The remaining states of this layer result in a draw.

Theorem 2 (Correctness)**.** *The presented algorithm is correct, i. e., it determines the game theoretical value wrt. the chosen opponent model.*

Proof. The correctness of the forward search comes immediately from the use of a BFS. We generate all reachable states, no matter if we remove duplicates or not. And as the games are finite by definition, we will find only finitely many layers.

For the second stage, we need to show that all states are correctly solved according to the opponent model. We show this using induction. We start at the states in the final layer, which we immediately can solve according to their corresponding rewards. When tracing back towards the initial state, the terminal states again are immediately solvable by their rewards. The most important observation is that due to the construction, non-terminal states have successors only within the next layer. All states within this layer are already solved. If we check if a state has a successor achieving a certain reward and look at the rewards in the order according to the opponent model, we can be certain that all states within the current layer can be solved correctly as well. □

Note that if we removed the duplicate states within different layers, we would reach states whose successors are not in the next layer but in some layer closer to the initial state and thus not yet solved, so we could not correctly solve such a state when reaching it for the first time.

Some games are not strictly alternating, i. e., a player might perform two or more consecutive moves, so that both players can be active in different states within the same BFS layer. To handle this, we split the second step of Algorithm 2 (lines 11 to 18) in two and perform this step once for the states where the first player was the active one and once for the second player. Note that both players go through the possible reward combinations in different orders, thus it is not possible to combine these two steps. Instead, we have to solve the states once for one player, store the result on disk, solve the remaining states for the other player, load the previous results, calculate the disjunction, and store the total result on disk. The order in which the two players are handled is irrelevant, as there is no state where both players are active.

Experimental Results

We performed experiments using several games from the website of the German general game playing server[2], which we instantiated automatically[3]. For Clobber (Albert et al. 2005) we specified rewards dependent on the number of pieces left on the board, so that we came up with general rewards, while the other games are all zero-sum games.

We implemented the presented algorithm in Java using JavaBDD[4], which provides a native interface to the CUDD package[5], a BDD library written in C++.

[2]http://euklid.inf.tu-dresden.de:8180/ggpserver/public/show_games.jsp

[3]For Connect Four we adapted the existing GDL description, as the instantiator's output was too large for the solver. For Clobber no GDL description exists, so that we created one from scratch.

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

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

Table 1: Results of Solving Two-Player Turn-Taking Games. All times in m:ss or h:mm:ss.

	Time		optimal
Game	(new)	(existing)	outcome
Catch a Mouse	0:24	1:44	100/0
Chomp	0:10	0:02	100/0
Clobber 3×4	0:01	0:01	0/40
Clobber 4×5	10:43	1:18:51	30/0
Connect 4 (5×6)	31:37	2:14:52	50/50
Cubi Cup 5	8:21:09	o.o.m.	100/0
Nim	0:10	0:01	100/0
Number Tic-Tac-Toe	1:41	4:54	100/0
Sheep and Wolf	0:23	0:57	0/100
Sum 15	0:01	0:00	50/50
Tic-Tac-Toe	0:01	0:00	50/50

General Observations

Our system contains a CPU with 2.67 GHz and 12 GB RAM. The runtime results for our new approach as well as the existing one are compared in Table 1. From this we can see that for the small games such as Tic-Tac-Toe or Sum 15 the new approach does not lose much, although all results are stored on the hard disk. Omitting this in the cases where all BDDs easily fit into RAM, however, would speed up the search.

For two slightly larger games, namely Chomp and Nim, the new approach is slower than the old one. This is due to the fact that for these games there is no incremental progress measure, so that we generate duplicate states in different layers and expand them several times. This results in more BFS layers (56 layers with 162,591 states opposed to 8 layers with 25,734 states for Chomp and 63 layers with 1,866,488 states opposed to 5 layers with 129,776 states for Nim), which in turn results in more effort during the solving stage.

For the larger games the new algorithm clearly outperforms the existing one. In all these games, an incremental progress measure can be found explicitly (e. g., a step counter in Catch a Mouse) or implicitly (e. g., the number of stones removed from the board in Clobber). Sheep and Wolf is the only game we solved, for which the second definition of the incremental progress measure is needed: Whenever the wolf moves, the progress does not increase, while the sheep can only move forwards. Thus, the sum of the rows of the sheep is a possible incremental progress measure. Due to the partitioning according to the layers, the BDDs stay smaller and the image thus can be calculated faster. We also save time as we do not need to calculate the strong pre-images.

As a result, we see that the new approach outperforms the existing one for games that are not too small and that incorporate an incremental progress measure.

Cubi Cup

In the game Cubi Cup[6], cubes are stacked with a corner up on top of each other on a three dimensional board. A new

[6]For a short description of the game by its authors see http://english.cubiteam.com.

cube may only be placed in positions where the three touching cubes on the bottom are already placed. If one player creates a state where these three neighbours have the same color, this is called a Cubi Cup. In this case, the next player has to place the cube in this position and remains active for another turn. The player to place the last cube wins – unless the three touching cubes produce a Cubi Cup of the opponent's color; in this case the game ends in a draw.

Due to the rule with one player needing to perform two moves in a row, it is clear that in one BFS layer both players might be active (in different states), so that we needed to use the proposed extension of the algorithm.

We are able to solve an instance of Cubi Cup with an edge length of 5 cubes.[7] Using the existing approach we were not able to solve this instance, as it needed too much memory: After nearly four hours of computation less than 25% of all states were solved but the program started swapping.

Connect Four

We also performed experiments for the game Connect Four. This was originally (weakly) solved in 1988 independently by James D. Allen and Victor Allis (Allis 1988) for a 7×6 board (7 columns and 6 rows). For this instance we were able to perform the complete reachability analyis achieving a total of $4{,}531{,}985{,}219{,}092$ reachable states, but unfortunately the BDDs get too large during the solving steps.

We noticed that the sizes for the BDDs representing the terminal states as well as those representing the rewards are very large. This is due to the fact that the two rules for the players having achieved a line are largely independent.

In the terminal state description we have a disjunction of the case that player 1 has achieved a line, player 2 has achieved a line, or neither has and the board is filled. So to find the terminal states of a layer, we first calculated the conjunctions with each of the BDDs representing only one part of the disjunction and afterwards calculated the disjunction of these. Similarly we could partition the reward BDDs.

In both cases, the intermediate BDDs were a lot smaller and the reachability calculation was sped up by a factor of about 4. Thus, at least for Connect Four not only partitioning the BDDs according to the BFS layers but also according to parts of the terminal and reward descriptions kept them smaller and thus calculation times lower. It remains yet to be seen if it is possible to automatically find such partitions of the BDDs for any given game.

With the machine described before, we were only able to completely solve Connect Four up to a size of 5×6; the 6×6 version needed a bit more than the available 12 GB of main memory. On a machine with 64 GB RAM and a 2.6 GHz CPU we were able to solve this instance as well. The new approach took $22{:}38{:}09$ to completely solve all $69{,}173{,}028{,}785$ reachable states, while the existing approach was aborted after about four days. We also tried to solve the original instance of 7×6, but for that even 64 GB of RAM were not sufficient.

[7] Unfortunately, we had to stop the solving several times and restart with the last not completely solved layer, as somehow the implementation for loading BDDs using JavaBDD and CUDD seems to contain a memory leak, which so far we could not locate. No such leak exists in the existing approach, as it does not load or store any BDDs.

Discussion

Unfortunately, BDDs are rather unpredictable. Their size greatly depends on the encoding of the states. Given a good variable ordering, they might save an exponential number of variables. Also, it is hard to predict the efficiency of a BDD. On the one hand, BDDs storing more states might be smaller than those storing fewer states. On the other hand, their efficiency is highly domain dependent: For some domains, such as Sokoban, BDDs are great, while for others, such as permutation games like the Sliding Tiles Puzzle they do not help much, no matter what variable ordering was chosen (Ball and Holte 2008; Edelkamp and Kissmann 2008a).

An interesting side-remark might be that this approach can in principle also be used for any turn-taking game. All we need is the way to pass through the p-dimensional matrix of (possible) reward combinations, which gives us an opponent model. Unfortunately, this is not found trivially. Especially, in general game playing the agent gets no information as to which other agents it plays against, so that learning such a model seems impossible so far. If we assume that we can get an opponent model, we are able to solve all turn-taking games under the assumption that the model holds. The result is then similar to that of the Max^n algorithm by Luckhardt and Irani (1986), and thus has the same shortcomings – namely, if one of the players does not play according to the model, the solution might be misleading.

Conclusions and Future Work

We presented a new algorithm for solving general two-player turn-taking games making use of the information of the forward BFS. This brings the advantage that we do not have to use any strong pre-images, as all the successors of a given layer are solved for sure once this layer is reached. We have shown that this algorithm can greatly outperform existing approaches.

One shortcoming is that the BFS is mandatory, while this was not the case for the existing algorithms. Furthermore, it does not perform any duplicate detection, so that in some games more BFS layers are generated and states are expanded multiple times.

One of the advantages is that we can stop the solving at any time and restart with the last partially solved layer later on. Also, we can use the information we find on the hard disk as an endgame database, e. g., in combination with a general game player that uses UCT (Kocsis and Szepesvári 2006) for finding good moves.

A future research avenue is to find a way to further partition the BDDs in a way that improves the approach. So far, we partitioned the BDDs only according to the BFS layers and in most cases this results in a great improvement. For games incorporating a step counter, we partitioned the BDDs according to the mutually exclusive variables representing it. Such an approach might also work for other mutually exclusive variables, but the question is, how to find the best ones automatically. Also, it is unclear, if such a partitioning generally helps to keep the BDDs smaller, so that it is important to find a good partitioning that will decrease the BDD sizes and speed up the pre-image calculations.

References

Albert, M. H.; Grossman, J. P.; Nowakowski, R. J.; and Wolfe, D. 2005. An introduction to Clobber. *INTEGERS: The Electronic Journal of Combinatorial Number Theory* 5(2).

Allis, V. 1988. A knowledge-based approach of connect-four. Master's thesis, Vrije Universiteit Amsterdam.

Ball, M., and Holte, R. C. 2008. The compression power of symbolic pattern databases. In *18th International Conference on Automated Planning and Scheduling (ICAPS)*, 2–11. AAAI Press.

Bercher, P., and Mattmüller, R. 2008. A planning graph heuristic for forward-chaining adversarial planning. In *18th European Conference on Artificial Intelligence (ECAI)*, volume 178, 921–922. IOS Press.

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

Edelkamp, S., and Kissmann, P. 2008a. Limits and possibilities of BDDs in state space search. In *23rd AAAI Conference on Artificial Intelligence (AAAI)*, 1452–1453. AAAI Press.

Edelkamp, S., and Kissmann, P. 2008b. Symbolic classification of general two-player games. In *31st Annual German Conference on Artificial Intelligence (KI)*, volume 5243 of *LNCS*, 185–192. Springer.

Fikes, R. E., and Nilsson, N. J. 1971. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence* 2(3–4):189–208.

Finnsson, H., and Björnsson, Y. 2008. Simulation-based approach to general game playing. In *23rd AAAI Conference on Artificial Intelligence (AAAI)*, 259–264. AAAI Press.

Genesereth, M. R.; Love, N.; and Pell, B. 2005. General game playing: Overview of the AAAI competition. *AI Magazine* 26(2):62–72.

Helmert, M. 2008. *Understanding Planning Tasks: Domain Complexity and Heuristic Decomposition*, volume 4929 of *LNCS*. Springer.

Jensen, R. M.; Veloso, M. M.; and Bowling, M. H. 2001. OBDD-based optimistic and strong cyclic adversarial planning. In *6th European Conference on Planning (ECP)*, 265–276.

Kissmann, P., and Edelkamp, S. 2009. Instantiating general games. In *IJCAI-Workshop on General Game Playing*, 43–50.

Kocsis, L., and Szepesvári, C. 2006. Bandit based Monte-Carlo planning. In *17th European Conference on Machine Learning (ECML)*, volume 4212 of *LNCS*, 282–293. Springer.

Kuhlmann, G.; Dresner, K.; and Stone, P. 2006. Automatic heuristic construction in a complete general game player. In *21st National Conference on Artificial Intelligence (AAAI)*, 1457–1462. AAAI Press.

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.

Luckhardt, C. A., and Irani, K. B. 1986. An algorithmic solution of N-person games. In *5th National Conference on Artificial Intelligence (AAAI)*, 158–162. Morgan Kaufmann.

Schaeffer, J.; Burch, N.; Björnsson, Y.; Kishimoto, A.; Müller, M.; Lake, R.; Lu, P.; and Sutphen, S. 2007. Checkers is solved. *Science* 317(5844):1518–1522.

Schiffel, S., and Thielscher, M. 2007. Fluxplayer: A successful general game player. In *22nd AAAI Conference on Artificial Intelligence (AAAI)*, 1191–1196. AAAI Press.