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 binary decision diagrams (BDDs). It consists of two stages: First, it determines all breadth-first search (BFS) layers using forward search and omitting duplicate detection, next, the solving process operates in backward direction only within these BFS layers thereby partitioning all BDDs 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 in terms of runtime and used memory so that it can solve games that could not be solved before with a general approach.

Introduction

In recent years general game playing has received an increasing amount of attention, especially due to 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 the agent tries to find a sequence of moves 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 in 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: in each possible terminal state the players are awarded points ranging from 0 (worst) to 100 (best).

Problems from the non-deterministic 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), while general game playing supports any number of participants, so that 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, in each state, only one player may decide which move to take. 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, which means finding the outcome for each player in any reachable state in case of optimal play. Using domain dependent solvers, several games have been solved, though often only in a weaker sense, so that the optimal outcome is known for the initial state and the states along the optimal paths, but not for all states. One of the latest prominent results was by Schaeffer et al. (2007), who were able to solve American Checkers after more than ten years of computation and proved that the optimal outcome is a draw. Of course, due to the domain independent scenario, 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, which utilizes binary decision diagrams (BDDs) (Bryant 1986) as they decrease the memory consumption, if a good variable ordering is found.

In this paper we will present a new approach to solve general two-player turn-taking games using an approach that uses the information of a breadth-first search (BFS) by operating only within the layers of reachable states. For most games storing only the current and successor layer in memory and the rest on the hard disk proves to be good for saving memory as well as runtime compared to our previous approaches (Edelkamp and Kissmann 2008b). Thanks to the smaller amount of used memory, we are now able to solve some more complex instances of games that the previous approach cannot handle.

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, and, finally, we present a short discussion, draw conclusions and point out possible future research avenues.

```
(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)) (does xplayer (mark ?m ?n)))  ; effects of marking a cell
(<= (next (cell ?m ?n o)) (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)) (true (control oplayer)))  ; change of the active player
(<= (next (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) (true (control oplayer)))  ; if opponent active, do nothing
(<= (legal oplayer noop) (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.

General Game Playing

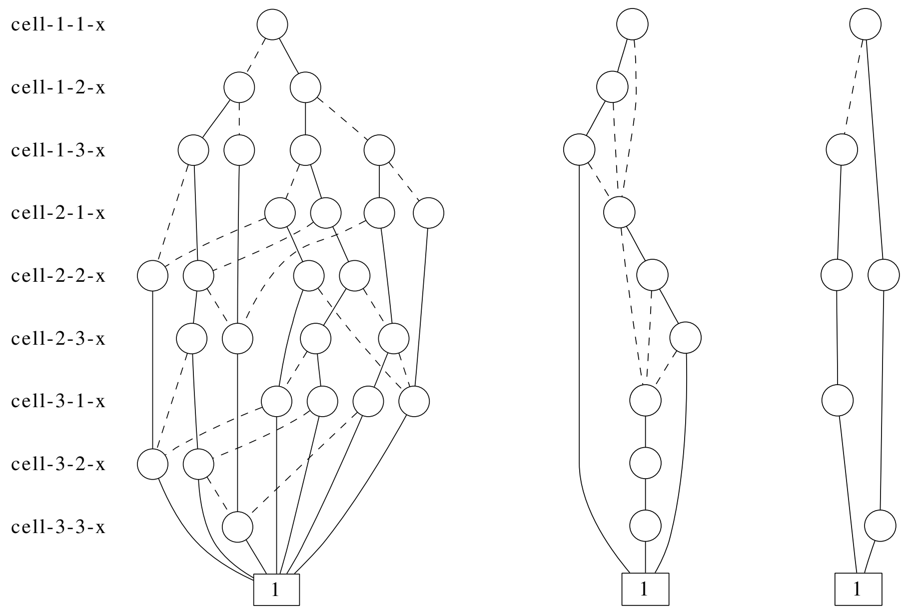
General game playing is concerned with playing games that need to be finite, discrete, and deterministic and must contain full information for all the players. It is possible to model single- as well as multi-player games, which by default are games with simultaneous moves by all players. They can be 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 $\mathcal{G} = \langle \mathcal{S}, \mathcal{P}, \mathcal{M}, \mathcal{I}, \mathcal{T}, \mathcal{R} \rangle$ with $\mathcal{S}$ being the set of reachable states, $\mathcal{P}$ the set of participating players, $\mathcal{M} \subseteq \mathcal{S} \times \mathcal{S}$ the set of possible moves for each state, $\mathcal{I} \in \mathcal{S}$ the initial state, $\mathcal{T} \subseteq \mathcal{S}$ the set of terminal states, and $\mathcal{R} : \mathcal{T} \times \mathcal{P} \mapsto \{0, \ldots, 100\}$ the reward for each player in all terminal states. General games are defined implicitly, i. e., only the initial state $\mathcal{I}$ 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 $i \in \{1, \ldots, |\mathcal{P}|\}$ is active as well as subsets $\mathcal{M}_i \subseteq \mathcal{M}$ denoting those moves, where player $i \in \{1, \ldots, |\mathcal{P}|\}$ 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 states $\mathcal{T}$ by the `terminal` keyword and the rewards $\mathcal{R}$ by the `goal` keyword. The moves $\mathcal{M}$ are split 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 $\mathcal{I}$. 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. 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



(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 Boolean variable (denoted on the left); solid edges mean that it 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.

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, we chose not to compare to those.

Symbolic Search

When we speak of symbolic search we mean state space search using 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 (elimination of nodes with both successors being identical and merging of nodes having the same successors), so that only a minimal number of BDD nodes is needed to represent a given formula / set of states. The resulting representation also is unique and all duplicates that might be present in a given set are captured by the BDD structure, so that each state is stored only once and the BDD is free of duplicates.

BDDs enable us to completely search some state spaces that would not be possible in explicit search. E. g., in the original 7×6 version of Connect Four $4{,}531{,}985{,}219{,}092$ states are reachable.[1] 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 43.8 TB to store all of them, while with BDDs 16 GB are sufficient. If we store only the current 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) as well as Schiffel and Thielscher (2007) who all identify the input and output parameters of each predicate. Often, input parameters denote the positions

[1] Recently, John Tromp arrived independently at the same result, see http://homepages.cwi.nl/~tromp/c4/c4.html.

on a game board while the output parameters specify its content. Predicates sharing the same name and the same input but different output parameters 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 $\mathcal{M}$, the possible combinations of moves of all players. Each move $m \in \mathcal{M}$ can be represented by a BDD $trans_m$, so that the complete transition relation $trans$ is their disjunction: $trans := \bigvee_{m \in \mathcal{M}} trans_m$.

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.[2] This way, if we start at the initial state, each call of the image results in the next BFS layer, so that the a complete BFS is 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-image* results in a BDD representing all the states that are predecessors of the given set of states *from*:

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

This allows us to perform BFS in backward direction.

An additional operator, the strong pre-image *spi*, which we needed in previous approaches, returns all those predecessor states of a given set of states *from* whose successors are within *from*. It is defined as

$$spi\,(from) := \forall S'.\,(trans\,(S, S') \rightarrow from\,(S'))$$

and can be derived from the pre-image:

$$spi\,(from) = \neg pre\text{-}image\,(\neg from)\,.$$

Solving General Two-Player Turn-Taking Games

Our existing approach for solving general two-player turn-taking games (Edelkamp and Kissmann 2008b) works by using a 101×101 matrix M of BDDs. The BDD at $M\,[i, j]$

[2] We omit the explicit mention of this in the pseudo-codes to enhance 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 ← I;
l ← 0;
while curr ≠ ⊥ do
    store curr as layer l on disk;
    prev ← curr ∧ ¬T;
    curr ← image (prev);
    l ← l + 1;
end while
return l − 1;
```

represents the states where player 1 can achieve a reward of i and player 2 a reward of j, with $i, j \in \{0, \ldots, 100\}$. Initially, all terminal states are inserted in the corresponding buckets. Starting at these, the strong pre-image is used to calculate those preceding states that can be solved as well, as all their successors are 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) followed by the solving in backward direction (see Algorithm 2), which operates within the calculated BFS layers.

Starting at the initial state $\mathcal{I}$, in the forward search 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. Also, for smaller BDDs the calculation of the image or pre-image often is faster, so that with this approach most games should be solved in a shorter time using less memory and thus more complex games can be solved.

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 states that appear on different paths in different layers will be expanded more than once.

To find out when we will have to deal with such duplicate states, first of all we need to define a progress measure.

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

1. *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 $\psi\,(s') = \psi\,(s) + 1$.*
2. *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* $\psi(s'') = \psi(s') + 1 = \psi(s) + 1$.

For the game of Tic-Tac-Toe the number of tokens placed on the board is an incremental progress measure: after each player's move the number of tokens increases by exactly one, until either the board is filled or one of the players has succeeded in constructing a line.

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:

1. 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 $\mathcal{I}$. 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.
2. If $\mathcal{G}$ is alternating, 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 $l + 1$ (i. e., if $(s_1, s_2) \in \mathcal{M}_1$). 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 $\mathcal{I}$. 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 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 $s' \in succ(s)$, $\psi(s) = \psi(s')$. All successors s' are inserted into layer $l+1$. For all states s' in layer $l + 1$ and $s'' \in succ(s')$, $\psi(s'') = \psi(s') + 1$. All successors $s'' \in succ(s')$ 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 $\mathcal{I}$ to one of the terminal states. This is due the fact that in such a case each state might reside in every layer.

Algorithm 2: Solving General Two-Player Games

Input: General game description $\mathcal{G}$.

```
1  l ← reach(G);
2  while l ≥ 0 do
3      curr ← load BFS layer l from disk;
4      currTerminals ← curr ∧ T;
5      curr ← curr ∧ ¬currTerminals;
6      for each i, j ∈ {0, ..., 100} do
7          terminals_{l,i,j} ← currTerminals ∧ R_{i,j};
8          store terminals_{l,i,j} on disk;
9          currTerminals ←
           currTerminals ∧ ¬terminals_{l,i,j};
10     end for
11     for each i, j ∈ {0, ..., 100} do in specific order
12         succ_1 ← load terminals_{l+1,i,j} from disk;
13         succ_2 ← load rewards_{l+1,i,j} from disk;
14         succ ← succ_1 ∨ succ_2;
15         rewards_{l,i,j} ← curr ∧ pre-image(succ);
16         store rewards_{l,i,j} on disk;
17         curr ← curr ∧ ¬rewards_{l,i,j};
18     end for
19     l ← l − 1;
20 end while
```

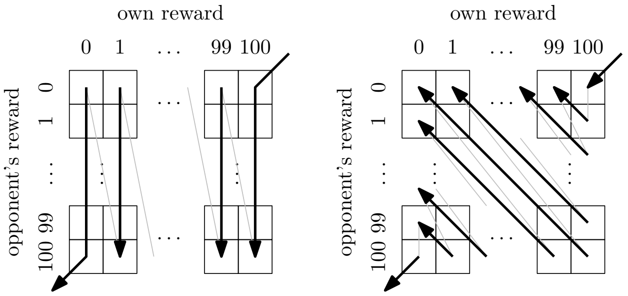
Once all BFS layers are calculated we can start the second stage, the actual solving process, for which we perform a symbolic retrograde analysis (see Algorithm 2). We start at the last generated BFS layer l and move upwards layer by layer until we reach the initial state $\mathcal{I}$ ($l = 0$).

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 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.

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.

For the game Tic-Tac-Toe we start in layer 9, where all cells are filled. All these states are terminal states, thus we



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

Figure 3: Order to process the reward combinations.

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 of `o`s, 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. 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 forward search's correctness comes immediately from the use of BFS. We generate all reachable states, no matter if we remove duplicates or not. 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 $\mathcal{R}$. When tracing back towards $\mathcal{I}$, the terminal states again are immediately solvable by $\mathcal{R}$. 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 $\mathcal{I}$ and thus not yet solved, so we could not correctly solve such a state when reaching it. Also note that due to the layer-wise operation we can omit the costly strong pre-images of our existing approach, so that the new one should be faster for those games that contain an incremental progress measure.

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 each 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 results on disk, solve the remaining states for the other player, load the previous results, calculate the disjunction, and store the total results 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[3], which we instantiated automatically[4]. Clobber (Albert et al. 2005) and the two-player version of Chinese Checkers are the only games for which general rewards are provided, while all other games are designed to be zero-sum.

We implemented the presented algorithm in Java using JavaBDD[5], which provides a native interface to the CUDD package[6], a BDD library written in C++.

Our system consists of an Intel Core i7 920 CPU with 2.67 GHz and 24 GB RAM. Some of the detailed runtime results for our new approach as well as the existing one are presented in Table 1, while Figure 4 compares the results of all solved games using the two approaches.

From these we can see that for most games, which contain an incremental progress measure, the new approach looses slightly if the runtime is less than one second, as 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 the larger games the new approach clearly outperforms the existing one: 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 but get the solvable states immediately by loading the next layer.

The games Chomp and Nim, which both do not contain an incremental progress measure, were scaled to different sizes, to see how well the new approach performs. From these we see that it takes a longer total runtime until the new appraoch at least matches the existing one. This is due to the forward

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

[4] 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.

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

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

Table 1: Results of solving two-player turn-taking games. All times in m:ss. An entry o.o.m. denotes the fact that the corresponding appraoch exceeded the available memory.

Game	Time (New)	Time (Existing)	Opt. Result
Catch a Mouse	0:19.04	1:14.73	100/0
Chinese Checkers 2	6:45.11	63:52.62	50/50
Chomp (8×7)	0:04.21	0:01.59	100/0
Chomp (10×10)	0:48.26	0:58.96	100/0
Clobber 4×5	7:24.13	55:03.91	30/0
Connect 4 (5×6)	30:27.85	139:06.55	50/50
Connect 4 (6×6)	563:48.46	o.o.m.	0/100
Cubi Cup 5	565:36.74	o.o.m.	100/0
Nim 4	0:04.15	0:00.82	100/0
Number Tic-Tac-Toe	1:03.33	3:25.31	100/0
Sheep and Wolf	0:12.90	0:44.88	0/100
Tic-Tac-Toe	0:00.60	0:00.09	50/50

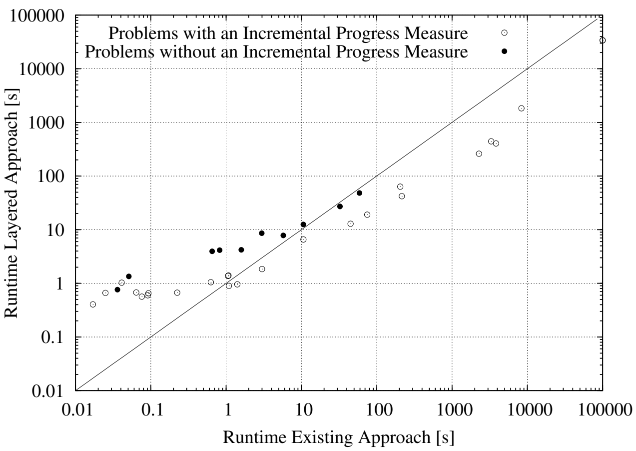


Figure 4: Comparison of the runtimes for the two approaches.

search, which takes longer with the new approach, as it finds more states in more layers (8,498,776 states in 100 layers opposed to 369,510 states in 10 layers for Chomp (10×10) and 64 layers with 2,179,905 states opposed to 5 layers with 149,042 states for Nim 4). Nevertheless, both approaches perform the same number of backward steps, so that for the more complex games the total runtime of the new approach again is smaller, because the loading of a layer then takes less time than the calculation of a strong pre-image.

The most complex games we can solve are instances of Cubi Cup[7] and Connect Four. In Cubi Cup cubes are stacked with a corner up on top of each other on a three dimensional board. A new 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 of one player needing to perform several moves in a row it is clear that in one BFS layer both players might be active (in different states), so that we need 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.[8] Using the existing approach we are not able to solve this instance, as it needs too much memory: After nearly ten hours of computation less than 60% of all states are solved but the program already starts swapping.

For Connect Four, both approaches can solve the game on a board of size 5×6, but on a 6×6 board the existing approach also needs too much memory to be able to solve it, while the new one finishes it in nearly one day. The normal version on a 7×6 board was originally (weakly) solved in 1988 independently by James D. Allen and Victor Allis (Allis 1988). For this instance we are 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[9].

Discussion

For the 7×6 board of Connect Four 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.

Unfortunately, BDDs are rather unpredictable. Their size greatly depends on the encoding of the states, though for some domains, such as the 15-Puzzle, no variable ordering will save an exponential number of BDD nodes (Ball and Holte 2008; Edelkamp and Kissmann 2008a).

[7] See `http://english.cubiteam.com` for a short description of the game by its authors.

[8] 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 appears in the existing approach, as it does not load or store any BDDs.

[9] We even performed experiments on a machine with 64 GB RAM, but this is still insufficient to solve all states.

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 $|\mathcal{P}|$-dimensional matrix of (possible) reward combinations, which gives us an opponent model. Unfortunately, for multi-player games this is not found trivially: In general game playing the agents get no information as to which other agents they 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: 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 no strong pre-images are applied, as all the successors of a given layer are solved 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. This might be generalizable to partitioning the BDDs according to any of the mutually exclusive variables, but it is not yet clear 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.

Acknowledgments

Thanks to Deutsche Forschungsgemeinschaft (DFG) for support in project ED 74/11-1. We also thank the anonymous reviewers for their helpful comments.

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 *ICAPS*, 2–11. AAAI Press.

Bercher, P., and Mattmüller, R. 2008. A planning graph heuristic for forward-chaining adversarial planning. In *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 *AAAI*, 1452–1453. AAAI Press.

Edelkamp, S., and Kissmann, P. 2008b. Symbolic classification of general two-player games. In *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 *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 *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 *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 *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 *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 *AAAI*, 1191–1196. AAAI Press.