Lex-Partitioning: A New Option for BDD Search

Stefan Edelkamp

Faculty of Mathematics and Computer Science
Universität Bremen, Germany
edelkamp@tzi.de

Peter Kissmann

Department of Computer Science
Universität des Saarlandes, Saarbrücken, Germany
kissmann@cs.uni-saarland.de

Álvaro Torralba

Planning and Learning Group
Universidad Carlos III de Madrid, Spain
atorralb@inf.uc3m.es

For the exploration of large state spaces, symbolic search using binary decision diagrams (BDDs) can save huge amounts of memory and computation time. State sets are represented and modified by accessing and manipulating their characteristic functions. BDD partitioning is used to compute the image as the disjunction of smaller subimages.

In this paper, we propose a novel BDD partitioning option. The partitioning is lexicographical in the binary representation of the states contained in the set that is represented by a BDD and uniform with respect to the number of states represented. The motivation of controlling the state set sizes in the partitioning is to eventually bridge the gap between explicit and symbolic search.

Let n be the size of the binary state vector. We propose an $O(n)$ ranking and unranking scheme that supports negated edges and operates on top of precomputed satcount values. For the uniform split of a BDD, we then use unranking to provide paths along which we partition the BDDs. In a shared BDD representation the efforts are $O(n)$. The algorithms are fully integrated in the CUDD library and evaluated in strongly solving general game playing benchmarks.

1 Introduction

In this paper we are concerned with the space-efficient traversal of state spaces with explicit-state search frontiers that are too large to be kept in main memory. One of the options is a symbolic representation of state sets in form of characteristic functions, which are manipulated in the exploration.

Following this approach, *binary decision diagrams* (BDDs) [6] have been shown to be an efficient data structure for representing and exploring large state sets in Model Checking [16] and Planning [7], For some domains they can save a tremendous amount of memory (wrt. an explicit representation). BDDs are also used as space-efficient construction *pattern database* search heuristics [2].

General game playing (GGP) goes in a similar direction as automated action planning, but it can be seen to be even more general. In general game playing the agent, i.e., the player, has to handle any game that can be described by the used description language without intervention of a human, and also without the programmer knowing what game will be played – it has to come up with a good strategy on its own. The most used description mechanism is the *game description language* GDL [14]. A BDD classification algorithm for (strongly) solving general single- and two-player games has been proposed in [12].

A *rank* is a number uniquely representing a state and the inverse process, called unranking, reconstructs the state given its rank. The approach advocated in this paper builds on top of findings of [8], who illustrated that ranking and unranking of states in a state set represented as a BDD is available in time linear to the length of the state vector (in binary representation). In other words, BDD ranking aims at

the symbolic equivalent of constructing a perfect hash function in explicit-state space search [3]. For the construction of the perfect hash function, the underlying state set to be hashed is generated in advance in form of a BDD. This is plausible when computing *strong solutions* to problems, i.e., the game-theoretical value for each reachable state. Applications are, e.g., endgame databases, or planning tasks where the problem to be solved is harder than computing the reachability set.

Perfect hash functions to efficiently rank and unrank states have been shown to be very successful in traversing single-player games, such as Rubik's Cube or the Pancake Problem [13], or two-player games like Awari [19]. They are also used for creating pattern databases [4]. The problem of the construction of perfect hash functions for algorithms like *two-bit breadth-first search* is that they are problem-dependent.

BDD partitioning approaches have been proposed to address the so-called state-explosion problem, which refers to the observation that the size of a state space of a system tends to grow exponentially in the number of its variables. In this paper we indicate that *lex-partitioning*, short for *lexicographical partitioning*, can advance symbolic state space search.

As our approach refines the image operation, it applies to most BDD exploration approaches in AI and beyond. We start with an introduction to BDDs, to symbolic search and to computing strong solutions in general games (Section 2), as well as to the basic idea of partitioning BDDs (Section 3). Next, we turn to ranking and unranking (Section 4), extended to the setting where BDDs can have negated edges. Then, we consider the partitioning of a BDD into sub-BDDs of an equal number of satisfying assignments (Section 5). Finally, we provide results in solving general games (Section 6), discuss further implications and conclude (Section 7).

2 Binary Decision Diagrams for Strongly Solving Games

BDDs are a memory-efficient data structure used to represent Boolean functions as well as to perform set-based search. In short, a BDD is a directed acyclic graph with one root and two terminal nodes, the 0- and the 1-sink. Each internal node corresponds to a binary variable and has two successors, one (along the Else-edge) representing that the current variable is false (0) and the other (along the Then-edge) representing that it is true (1). For any assignment of the variables derived from a path from the root to the 1-sink the represented function will be evaluated to 1.

Bryant [5] proposed a fixed variable ordering, for which he also provided two reduction rules (eliminating nodes with the same Then- and Else-successor and merging two nodes representing the same variable that share the same Then-successor as well as the same Else-successor). These BDDs are called reduced ordered binary decision diagrams (ROBDDs). Whenever we mention BDDs in this paper, we actually refer to ROBDDs. We also assume that the variable ordering is the same for all the BDDs and has been optimized prior to the search.

BDDs have been shown to be very effective in the verification of hard- and software systems, where BDD traversal is referred to as *symbolic model checking* [16]. Adopting terminology to state space search, we are interested in the *image* of a state set S with respect to a transition relation *Trans*. The result is a characteristic function of all states reachable from the states in S in one step.

The image *Succ* of the state set S is computed as $Succ(x') = \exists x\, (Trans(x,x') \wedge S(x))$. The *preimage*, which determines all predecessors of the state set S, is computed as $Pre(x) = \exists x'\, (Trans(x,x') \wedge S(x'))$.

Using the image operator implementing a symbolic breadth-first search (BFS) is straight-forward. All we need to do is to repeatedly apply the image operator to the set of states reachable from the initial state found so far. The search ends when a fix-point is reached, i.e., when no new successor states can be found. We store the set of all reachable states as one BDD, so that, due to the structure of a BDD, this

does not contain any duplicates of states.

In general game playing (GGP) we are concerned with the problem of automatically playing a game that the player probably has never seen before, which is very similar to the action planning. There are several differences, the first and foremost of course being that in GGP we are not restricted to only one player, but rather an arbitrary number of participants is supported. While in classical action planning the goal is to find a plan, i.e., a sequence of actions transforming the initial state to a goal state, as short as possible, in GGP each terminal state has a specific outcome for each participating player and the goal is to maximize the own outcome.

Allis [1] proposed three kinds of *solutions* for two-player zero-sum games. In practice a *weak solution* is often enough, as it allows the game to be played *optimally* in the sense that the player following the solution will never achieve an outcome worse than what was predicted for the initial state, independent of the moves the opponent chooses. A problem arises only when not following the solution at some step; in that case it might be that the game-theoretic value as well as the best move to take are not known for the successor state. In GGP this problem might arise when we first played following some heuristic that told us what to do and finished the calculation of the solution only afterward. That is why we chose to calculate *strong solutions*, which corresponds to finding the game-theoretic value for each reachable state, and thus to be able to determine the best move to take for every state that might ever be encountered.

For the case of single-player games we might also speak about weak and strong solutions. In that case a weak solution corresponds to a plan that lets us reach the best possible outcome from the initial state, while a strong solution again tells us the best possible outcome for each reachable state, so that we can continue playing optimally even after a suboptimal move has been chosen.

In order strongly solve single- or two-player games [12], we find all the reachable states by performing symbolic BFS, but instead of storing all reachable states in one BDD we store each layer separately. The solving starts in the last reached layer and performs regression search towards the initial state, which resides in layer 0. This final layer contains only terminal states (otherwise the forward search would have progressed further), which can be solved immediately by calculating the conjunction with the BDDs representing the rewards for the two players. Once this is done, the search continues in the preceding layer, because the remainder of the layer is empty. If another layer contains terminal states as well, those are solved in the same manner before continuing with the remaining states of that layer. The rewards are handled in a certain order (e.g., in the order win–draw–loss for the currently active player in case of a zero-sum game or in decreasing order in case of a single-player game). All the solved states of the successor layer are loaded in this order and the preimage is calculated, which results in those states of the current layer that will achieve the same rewards and are thus solved.

3   Partitioning

For several domains constructing a transition relation *Trans* prior to the search consumes huge amounts of the available computational resources. Fortunately, it is not required to build *Trans* monolithically, i.e., as one big relation.

Provided a set of actions $\mathscr{A}$, we can partition *Trans* into individual transition relations Trans_a for each action $a \in \mathscr{A}$, s.t. $\mathit{Trans} = \bigvee_{a \in \mathscr{A}} \mathit{Trans}_a$. For such a *disjunctive partitioning* of the transition relation the image now reads as

$$Succ(x') = \exists x \left(\bigvee_{a \in \mathscr{A}} \mathit{Trans}_a(x, x') \wedge S(x) \right) = \bigvee_{a \in \mathscr{A}} \left(\exists x \left(\mathit{Trans}_a(x, x') \wedge S(x) \right) \right).$$

This image computation applies disjunctive splits for the different actions to be applied and can accelerate BDD exploration compared to a monolithical representation. One reason is that the relational product for computing an image results in many intermediate BDDs and reveals an NP hard problem (3-SAT can be reduced to an image operation [15]). The execution sequence of the disjunction has an effect on the overall running time. In this case, we organize the partitioned image in form of a binary tree, trying to have intermediate BDDs of similar size.

A partitioning of S into k disjoint sets $S_1, \ldots, S_k$ ($S_i \cap S_j = \emptyset$ for $i \neq j$) can lead to further simplified sub-images, so that we have

$$Succ(x') = \bigvee_{1 \leq l \leq k} \bigvee_{a \in \mathscr{A}} \left(\exists x \left(Trans_a(x, x') \wedge S_l(x) \right) \right).$$

Our partitioning method refines the following notion of a partitioned BDD.

Definition 1 (Partitioned BDD [18]). *Given a Boolean function $f : \{0,1\}^n \to \{0,1\}$, defined over n inputs $X_n = \{x_1, \ldots, x_n\}$, the partitioned BDD representation is a set of k function pairs $(w_1, f_1), \ldots, (w_k, f_k)$, where $w_i, f_i : \{0,1\}^n \to \{0,1\}$ are also defined over X_n and satisfy the following four conditions.*

1. *w_i and f_i are represented as BDDs respecting the same variable ordering as f, for $1 \leq i \leq k$.*

2. *$w_1 \vee \ldots \vee w_k = 1$.*

3. *$w_i \wedge w_j = 0$, for all $i \neq j$.*

4. *$f_i = w_i \wedge f$, for $1 \leq i \leq k$.*

We refer to the lexicographical ordering of bitvectors by using the subindex *lex*: for $a, b \in \{0,1\}^n$ we have $a <_{lex} b$ if there is an $i \in \{1, \ldots, n\}$ such that $a_i < b_i$ and for all $j \in \{1, \ldots, i-1\}$ we have $a_j = b_j$. Moreover, $a \leq_{lex} b$ iff $a <_{lex} b$ or $a = b$.

Definition 2 (Lex-Partitioned BDD). *Given a Boolean function $f : \{0,1\}^n \to \{0,1\}$, defined over n inputs $X_n = \{x_1, \ldots, x_n\}$, the lex-partitioned BDD representation of f is a set of k assignments $a_1 \ldots, a_k \in \{0,1\}^n$ and k functions $f_1, \ldots, f_k : \{0,1\}^n \to \{0,1\}$ that are also defined over X_n and satisfy the following conditions.*

1. *f_i are represented as BDDs respecting the same variable ordering as f, for $1 \leq i \leq k$.*

2. *$a_k = (1, \ldots, 1)$ and, for all $i < k$, we have $a_i <_{lex} a_{i+1}$.*

3. *$f_1 \vee \ldots \vee f_k = f$.*

4. *$f_i \wedge f_j = 0$ for all $i \neq j$.*

5. *$f_1 = f \wedge \bigvee_{a \leq_{lex} a_1} a$ and $f_i = f \wedge \bigvee_{a_{i-1} <_{lex} a \leq_{lex} a_i} a$, for all $1 < i \leq k$.*

Using coefficients $w_1 = \bigvee_{a \leq_{lex} a_1} a$ and $w_i = \bigvee_{a_{i-1} <_{lex} a \leq_{lex} a_i} a$ for $1 < i \leq k$ the definition specializes the one of partitioned BDDs. The advantage is that by the lexicographical ordering we obtain more control over the evolution of BDDs resulting from a split.

4   Ranking and Unranking

Linear-time ranking and unranking functions with BDDs have been given in [8]. *Ranking* is a minimal perfect hash function from the set of satisfying assignments to the position of it in the lexicographical ordering of all satisfying assignments. *Unranking* is the inverse operation to ranking.

```
1  precomputeSatCount ()
2    i = level(root);
3    satcount = 2^i * satCountAux(root);


1  satCountAux(n)
2    if (n == 1-sink()) return 1;
3    if (n == 0-sink()) return 0;
4    if (res = lookup(n)) return res;
5    t = sign(n) * Then(|n|); e = sign(n) * Else(|n|);
6    i = level(|n|); j = level(t); k = level(e);
7    satcount = (2^(j-i-1)) * satCountAux(t) +
8                (2^(k-i-1)) * satCountAux(e);
9    insert(n, satcount);
10   return satcount;
```

Figure 1: Satisfiability Counting with Negated Edges.

Definition 3 (Ranking and Unranking). *The* rank *of an assignment $a \in \{0,1\}^n$ is the position in the lexicographical ordering of all satisfying assignments of the Boolean function f, while the* unranking *of a number r in $\{0,\ldots,C_f-1\}$ is its inverse, with C_f being the total number of satisfying assignments of f.*

We have implemented the pseudo-code algorithms for the CUDD library [21]. The proposal in [8] does not support negated edges. Negated edges, however, are crucial, since otherwise function complementation is not a constant time operation, at least for a BDD in a shared representation [17].

Definition 4 (Edge Complementation, Satcount, Conversion). *The* index *of a BDD node is its unique position in the shared representation. For the ease of notation we take the negation of the node index to represent* edge complementation, *i.e., $-n$ is the negated and $|n|$ is the regular node index. The function* $\text{sign}(n)$ *returns -1 if the edge is complemented and 1 if it is not,* $\text{variable}(n)$ *returns the variable associated with n and* $\text{level}(n)$ *its position in the variable ordering. Moreover, we assume the 1-sink to have the node index 1 and the 0-sink to have the node index -1. Let $C_f = |\{a \in \{0,1\}^n \mid f(a) = 1\}|$ denote the number of satisfying assignments (*satcount*) of f. With* bin *(and* invbin*) we denote the* conversion *of the binary value of a bitvector (and the inverse operation).*

For ranking and unranking the satcount values are precomputed for every essential subfunction and stored in the unique table for the shared BDD. This table is used by two functions: *insert(n,v)* sets a value *v* for a node *n* in the unique table and *lookup(n)* retrieves it. Memory is allocated if a node is new.

As BDDs are reduced, not all variables on a path are present, but need to be accounted for in the satcount procedure. Figure 1 shows the pseudo-code of the function that does not only compute the values but also stores all the intermediate results and follows the proposal of [6]. We see that the time (and space) complexity is $O(|G_f|)$, where $|G_f|$ is the number of nodes of the BDD G_f representing f.

With negation on edges there are subtle problems to be resolved for storing the satcount values. While the number of satisfiable paths for a node might fit into a computer word this is not necessarily true for the negated subfunction. Therefore, we allow up to two satisfiability values to be stored together with a node: one wrt. reaching it on a negated edge, and the other one wrt. reaching it on a non-negated edge. In contrast to standard satisfiability count implementations (as in CUDD) this way we ensure that only satcount values of at most *c* are stored, where *c* is the satcount value of the root node. E.g., in ConnectFour 7×6 with 85 binary variables (yielding 2^{85} possible values), long integers are sufficient to

store intermediate satcount values, which are all smaller than the satcount value of the entire reachable set (4,531,985,219,092).

Figures 2 and 3 extend the proposal of [8] and show the ranking and unranking functions and thus realize an invertible minimal perfect hash function for f mapping an assignment $s \in \{0,1\}^n$ to a value $r \in \{0, \ldots, C_f - 1\}$. The procedures determine the rank given a satisfying assignment and vice versa. They access the satcount values on the Else-successor of each node (adding for the ranking and subtracting for the unranking). Missing nodes (due to BDD reduction) have to be accounted for by their binary representation, i.e., gaps of l missing nodes are accounted for 2^l. Edge complementation changes the sign of the node n and is progressed to evaluation of the sinks. While the ranking procedure is recursive the unranking procedure is not. Both procedures track the gap imposed by the distance in the levels of the current and the successor node

```
1  rank(s)
2    i = level(root);
3    d = bin(s[0..i-1]);
4    return d*satCount(root) + rankAux(root,s) - 1;

1  rankAux(n,s)
2    if (|n| == 1) return 1;
3    t = sign(n) * Then(|n|); e = sign(n) * Else(|n|);
4    i = level(|n|); j = level(e); k = level(t);
5    if (s[i] == 0)
6      return bin(s[i+1..j-1]) * satCount(e) + rankAux(e,s);
7    else
8      return 2^(j-i-1) * satCount(e) +
9             bin(s[i+1..k-1]) * satCount(t) + rankAux(t,s);
```

Figure 2: Ranking with Negated Edges.

Once the satcount values have been precomputed, both functions require linear time $O(n)$, where n is the number of variables in the function represented in the BDD. Dietzfelbinger and Edelkamp provide invariances showing that the procedures work correctly [8].

5 Splitting

Given the BDD G_f and any assignment $s \in \{0,1\}^n$, the *split* function computes the BDDs G_g and G_h with the satisfying sets $S_g = S_f \cap \{b \in \{0,1\}^n \mid b \leq_{lex} s\}$ and $S_h = S_f \cap \{b \in \{0,1\}^n \mid b >_{lex} s\}$. If we choose the assignment as the result of unranking $\lfloor C_f/2 \rfloor$ we get $C_g = \lfloor C_f/2 \rfloor$ and $C_h = \lceil C_f/2 \rceil$.

Figure 4 shows the pseudo-code of the recursive split algorithm. The input is the state vector in form of an assignment along which the BDD should be split. The result consists of two BDDs: the left BDD represents all the assignments lexicographically smaller or equal than the selected assignment a and the right BDD all the others. The algorithm traverses the path imposed by the input vector s bottom-up. Whenever needed, it allocates new nodes. If a node already exists, no allocation takes place. Depending on the truth value of the bitvector position *lev* currently processed, we swap the attachment of sub-BDDs.

Figure 5 shows how the algorithm works in a part of the BDD. Each node in the path represented by the assignment $a = 011 \ldots$ is split into two, depending on the value of the assignment for the associated variable.

```
1  unrank(r)
2    i = level(root); d = r / satCount(root);
3    s[0..i−1] = invbin(d);
4    n = root;
5    while (|n| > 1)
6      r = r % satCount(n);
7      t = sign(n) * Then(|n|); e = sign(n) * Else(|n|);
8      j = level(e); k = level(t);
9      if (r < (2^(j−i−1) * satCount(e)))
10        s[i] = 0;
11        d = r / satCount(e);
12        s[i+1..j−1] = invbin(d);
13        n = e; i = j;
14      else
15        s[i] = 1;
16        r = r − (2^(j−i−1) * satCount(e));
17        d = r / satCount(t);
18        s[i+1..k−1] = invbin(d);
19        n = t; i = k;
```

Figure 3: Unranking with Negated Edges.

If the assignment is 0, the recursion is made over the Else-edge. The result of the recursion is set as the Else-edges of the left and right part, respectively. The Then-edge points to the 0-sink in the left part and to the Then-successor of the original node in the right part. In the example N_1 is divided into N_1^L and N_1^R. All the assignments with $V_1 = 1$ are considered in the right part, while the others are split in the recursion. Symmetric rules apply in case that the assignment is 1 (node N_3 in the example).

The base case corresponds to the constant node, returning the 1-sink for the left part and the 0-sink for the right one, assigning s to the left part. Finally, if some node in the path is missing (due to the elimination rule of nodes with the same Then- and Else-successor), the algorithm still splits it into two following the same rules described above. In the example, N_6^L and N_6^R are the result of the split over a missing node in the original BDD with both edges pointing to N_3.

Theorem 1 (Time Complexity Split Function). *In a shared BDD representation given the BDD G_f and an assignment $a \in \{0,1\}^n$ the split function computes the BDDs G_g and G_h in $O(n)$ time.*

Proof. As at most *n* nodes are processed in post-order, the time complexity is immediate. Moreover, the BDDs that are constructed are reduced. All original nodes remain valid in the shared representation and each new node that is created in the bottom-up traversal is checked for applicability of the BDD reduction rules (by issuing a look-up in the unique table). □

Theorem 2 (Space Complexity Split Function). *We have $|G_g| \leq |G| + n$ and $|G_h| \leq |G| + n$. In a shared BDD representation we also have $|G_g \cup G_h| \leq |G| + 2n$.*

Proof. As at most 2*n* nodes are created in the shared representation the second result $|G_g \cup G_h| \leq |G| + 2n$ is immediate. For each individual function G_g and G_h we have constructed at most *n* new nodes. If we extract a BDD from the shared representation we duplicate nodes from G that are shared between the two structures. □

```
1   pair  split(s)
2     return  splitAux(root,0,s);
3
4   pair  splitAux(n,  lev,  s)
5     z = 0-sink();
6     if  (lev < level(|n|))
7       (s1,s2) = splitAux(n,lev+1,s);
8       if(s[lev])
9         left = new node(var(lev),n,s1);
10        right = new node(var(lev),z,s2);
11      else
12        left = new node(var(lev),s1,z);
13        right = new node (var(lev),s2,n);
14    else if  (n == 1 or n == 0)
15      left = n; right = z;
16    else
17      t = sign(n) * Then(|n|); e = sign(n) * Else(|n|);
18      if(s[lev])
19        (t1,t2) = splitAux(t,lev+1,s);
20        left  = new node(variable(|n|),t1,e);
21        right = new node(variable(|n|),t2,z);
22      else
23        (e1,e2) = splitAux(e,lev+1,s);
24        left  = new node(variable(|n|),z,e1);
25        right = new node(variable(|n|),t,e2);
26    return  (left,  right);
```

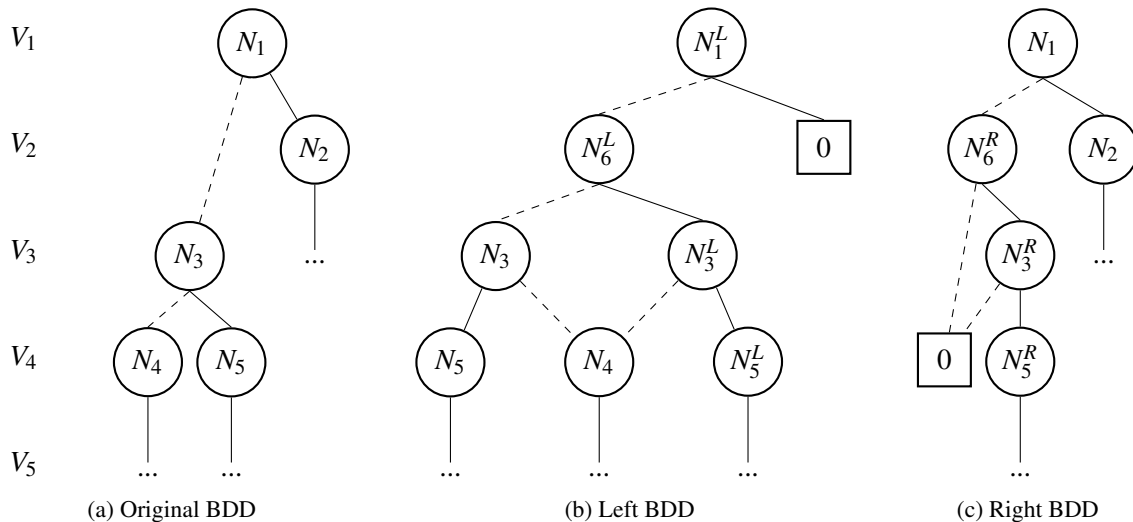Figure 4: Splitting with Edge Complements.

(a) Original BDD   (b) Left BDD   (c) Right BDD

Figure 5: Example of the split algorithm. The original BDD is split into its left and right part with respect to the assignment $011 \ldots$ (Dashed lines represent Else-edges).

By applying repeated splits we can uniformly partition a BDD into k parts in time $O(kn)$.

6   Experimental Evaluation

We have implemented our algorithms by extending the CUDD package [21] to support the different satcount procedures, ranking, unranking and various split options. The solver program is written in Java and connected to Fabio Somenzi's BDD package CUDD with the Java Native Interface (JNI).

We performed the experiments on one core of a 64-bit desktop computer (model Intel(R) Xeon(R) CPU X3470 with 2.93 GHz) running Linux (Ubuntu). This computer is equipped with 8 GB main memory and 8,192 KB cache. For the experiments there was no need to use virtual memory. We compiled the CUDD package using the GNU C++ compiler (gcc version 4.3 with option -O3).

We conducted experiments in different games provided in the general description language GDL [14]. The selection of games indicates the generality of the approach: seven single-player games (8-Puzzle, Asteroids Parallel, Knight's Tour, Lightsout, Lightsout 2, Peg Solitaire, and Tpeg) and five two-player games (Catch a Mouse, CephalopodMicro, ConnectFour 5 × 5, NumberTicTacToe, and TicTacToe) For the description of the games and their implementations in GDL we refer to the commonly used GGP server[1]. We used a timeout of one hour for every experiment.

For each game, the exploration is performed in two phases [12]. First, a breadth-first search generates the reachable states, organized in layers. Then, a backward exploration classifies the states in each layer according to their reward by computing the preimage of the classified sets of the next layer. In case the explorations are completed, the games are strongly solved, i.e., the game-theoretical value of each reachable state is computed. This amounts to a combined forward and backward exploration to compute the set of reachable states and to classify it into sets of different game-theoretical values. The number of backward images is usually greater than the number of forward images as different classification sets have to be computed by calling the image operator.

We compare our partitioning method to others already implemented in the CUDD library. There are different strategies, e.g., splitting for balancing the number of states, for balancing the number of nodes, and other disjunctive subset algorithms.

When balancing the number of states, it is possible to limit the number of states in each BDD by splitting the original BDD in as many parts as necessary (*States*) or to split the BDD in a fixed number of folds, all of them with the same number of states (*FoldStates*). To get partitions with the desired number of states we make use of our lexicographic partitioning (*Lex*). We do not report experiments with another state-selection strategy included in the CUDD library, given that there is not significant differences wrt. our lexicographic version. The main difference is that our version respect the lexicographic order, which may be an advantage when ranking/unranking states or when assigning states to different cores in a distributed version.

In order to get partitions with balanced number of nodes we consider also limiting the maximum number of nodes in each BDD (*Nodes*) or splitting the BDD in a fixed number of parts with balanced number of nodes (*FoldNodes*). The CUDD library includes several algorithms that allow splitting a BDD according to the number of nodes:

- Shortest Path (*SPath*): Procedure to subset the given BDD choosing the shortest paths (largest cubes) in the BDD.

- Compress (*Comp*): Finds a dense subset using several techniques in series. It is more expensive than other subsetting procedures, but often produces better results.

---

[1]http://ggpserver.general-game-playing.de

There are also other methods that allow for a disjunctive decomposition in two parts (*Disj*) according to different criteria:

- Iterative (*Ite*): Iterated use of supersetting to obtain a factor of the given function. The two parts tend to be imbalanced.

- Generation (*Gen*): generalizes the decomposition based on the cofactors with respect to one variable.

- Variable selection (*Var*): Decomposes the BDD according to the value of a variable, chosen to minimize and balance the size of the resulting BDDs.

	FStates Lex 8	States Lex 100000	FNodes Comp 8	FNodes SPath 8	Nodes Comp 10000	Nodes SPath 10000	Disj Var	Disj Ite	Disj Gen
8-puzzle	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
ConnectFour 5×5	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Knights Tour	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Lightsout	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Peg	0.84	0.76	0.79	0.82	0.53	0.56	0.79	0.79	0.71
Asteroids Parallel	1.04	0.11	0.99	0.99	0.10	0.10	0.99	0.99	0.99
Lightsout 2	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
TPeg	0.89	0.80	0.88	0.89	0.14	0.15	0.85	0.89	0.81
Catcha Mouse	1.00	0.22	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Cephalopod Micro	0.91	0.69	0.85	0.81	0.19	0.19	0.88	0.91	0.81
Number TicTacToe	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
TicTacToe	1.00	0.19	1.00	1.00	0.55	0.55	1.00	1.00	1.00
Total	0.97	0.76	0.95	0.95	0.68	0.69	0.95	0.96	0.93

Table 1: Number of computed layers by each partitioning scheme wrt. not applying any partitioning

Table 1 shows a comparison of the number of layers explored using each partitioning before the timeout. In half of the domains all the partitioning versions are able to finish the exploration. In the other domains most partitions are close to the exploration without partitioning solving more than 90% of the layers, except the three versions without a bounded number of partitions (*States Lex 100000*, *Nodes Shortest Path 10000* and *Nodes Compress 10000*). Fold States Lex 8 dominates the other partitioning methods in all the domains, being the only one able to compute more layers than the version without partitioning in one domain: Asteroids Parallel.

Tables 2 and 3 show the relative time spent in solving each game and the maximum number of nodes per image, respectively, for each partitioning configuration wrt. the "None" partitioning version. For those games where some partitioning strategy did not finish due to a timeout, only layers solved by the algorithm and the "None" partitioning version were taken into account.

In distributed or external memory settings the maximum number of nodes involved in a single image determines the memory needed. Thus, the results show that BDD partitioning can help to solve problems by reducing the memory requirements, at the cost of increasing the time spent. Versions splitting the BDDs in an unbounded number of parts achieve good memory reductions but they do not scale well, both when balancing the BDDs according to the number of states or according to the number of nodes.

	FStates Lex 8	States Lex 100000	FNodes Comp 8	FNodes SPath 8	Nodes Comp 10000	Nodes SPath 10000	Disj Var	Disj Ite	Disj Gen
8-puzzle	1.17	1.04	1.77	1.43	1.25	1.07	1.91	1.11	2.15
ConnectFour 5 × 5	1.44	1.57	3.24	2.28	7.72	5.74	2.97	1.50	3.75
Knights Tour	1.37	1.40	2.66	1.92	5.45	4.24	2.90	1.45	3.05
Lightsout	1.21	1.23	1.56	1.32	2.63	2.20	1.57	1.26	2.47
Peg	1.03	1.06	1.04	1.05	1.13	1.08	1.05	1.03	1.03
Asteroids Parallel	0.60	0.35	0.95	0.97	0.18	0.19	0.50	0.66	0.76
Lightsout 2	1.22	1.23	1.56	1.32	2.65	2.21	1.57	1.26	2.46
TPeg	1.02	1.04	1.02	0.98	1.10	1.08	1.04	1.01	1.03
Catcha Mouse	1.71	166.53	1.73	1.29	1.21	1.06	2.22	1.01	1.79
Cephalopod Micro	1.03	1.21	1.05	1.07	1.21	1.17	1.05	1.03	1.10
Number TicTacToe	1.18	1.20	1.71	1.89	2.38	2.55	1.55	1.42	2.16
TicTacToe	1.84	3.30	1.88	1.65	2.71	2.55	1.73	1.34	2.96
Total	0.99	1.23	1.12	1.09	1.15	1.08	1.01	0.97	1.19

Table 2: Relative time spent in solving each game for each partitioning scheme wrt. not applying any partitioning

	FStates Lex 8	States Lex 100000	FNodes Comp 8	FNodes SPath 8	Nodes Comp 10000	Nodes SPath 10000	Disj Var	Disj Ite	Disj Gen
8-puzzle	0.36	1.00	0.83	0.80	0.98	0.86	0.79	0.72	0.82
ConnectFour 5 × 5	0.23	0.15	0.85	0.57	0.84	0.74	0.62	0.60	0.87
Knights Tour	0.20	0.22	0.56	0.56	0.56	0.56	0.58	1.00	0.92
Lightsout	0.51	0.34	0.76	0.70	0.78	0.73	0.78	0.78	0.82
Peg	0.25	0.06	0.11	0.58	0.02	0.55	0.68	0.65	0.68
Asteroids Parallel	0.28	0.01	0.29	0.19	0.00	0.01	0.55	0.51	1.09
Lightsout 2	0.51	0.34	0.76	0.70	0.78	0.73	0.78	0.78	0.82
TPeg	0.23	0.05	0.62	0.52	0.02	0.58	0.67	0.61	0.70
Catcha Mouse	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Cephalopod Micro	0.22	0.01	0.90	0.61	0.01	0.42	0.61	0.73	0.65
Number TicTacToe	0.27	0.27	1.01	0.96	0.10	0.98	0.65	0.79	0.86
TicTacToe	0.41	0.00	0.10	0.96	0.04	0.84	0.83	0.93	1.31
Total	0.29	0.04	0.40	0.46	0.07	0.35	0.63	0.64	0.96

Table 3: Relative maximum number of nodes per image for each partitioning scheme wrt. not applying any partitioning

On the other hand, versions applying a partitioning in 2 or 8 folds do not impose a large overhead, being around 50 percent slower than the no-partitioning version.

When comparing the way to select subsets, in general the lexicographic partitioning allows great reductions in the necessary memory while not producing a large overhead in time. It gets the best coverage results among all partitioning methods, being the only one able to improve the coverage of the non-partitioning version in one domain. Furthermore, of all the strategies with a fixed number of partitions, it is the one achieving largest memory savings and the second best in overall time.

In the plots (Figures 6–8) we provide information on the images computed during the exploration in 6 different games of varying difficulty comparing different partitioning options. We omit the methods with less coverage (*States Lex 100000*, *Nodes Shortest Path 10000* and *Nodes Compress 10000*) to visualize well the differences between the other versions. Both search directions are provided in one plot, separated by a vertical line.

All the selected partitioning options fail to finish the exploration on TPeg and Asteroids Parallel, while they complete the search in all the other domains. In TPeg, the non-partitioning version is the one exploring more layers. On the other hand, in Asteroids Parallel, the lexicographic partitioning achieves the best results.

Figure 6 shows the time spent in computing each layer by the different partitioning methods. In general, applying partitioning over the BDDs slows the image computation, except in Asteroids Parallel. However, the overhead does not appear prohibitive for the application of partitioning in order to reduce the memory requirements. The efficiency of the partitioning methods is consistent within each domain, across all the layers. Among the different partitioning options, the *States Lex* version is the closest to the none partitioning across the different domains, followed by *Disjunctive Iterative* and *FoldNodes SPath 8*.

Figure 7 shows the total number of nodes involved in computing each layer, when applying each different method. For all partitioning methods, the sum of the nodes of each part is larger than the original, represented by the none partitioning version. On the other hand, the total number of nodes of *States Lex* partitioning is not lower than that of the other partitioning methods, suggesting that the improvements in runtime with respect other partitioning methods is thanks to the fast computation of the lexicographic splitting procedure.

Figure 8 represents the maximum size of the BDDs involved in a single image, determining the minimum memory requirements for performing the task. In this case, applying partitioning seems to help in most cases. Partitionings balancing the number of nodes, *Fold Nodes Compress 8* and *Fold Nodes Shortest Path 8* achieve the best performance in this metric. The lexicographic partitioning is also able to obtain significant reductions in the amount of required memory.

7   Conclusions and Future Work

In this paper we have shown an insightful approach for a balanced disjunctive image computation by splitting the BDD representing the state set into equally sized subsets to strongly solve games. A new BDD-splitting method based on a lexicographic criteria have been presented and compared to other partitionings in game solving tasks, showing that it is an effective method to reduce the memory requirements.

Even though we concentrated on game playing in AI the methods we proposed are general and likely applicable to other areas, such as the formal verification of sequential circuits. In the model checking domain BDD partitioning techniques, e.g., by [20, 10], often show significant advances for executing the image operation, but usually provide no control on the size of the BDD, nor on the number of states represented. Moreover, the actual work executed for computing the image appears to correlate not only
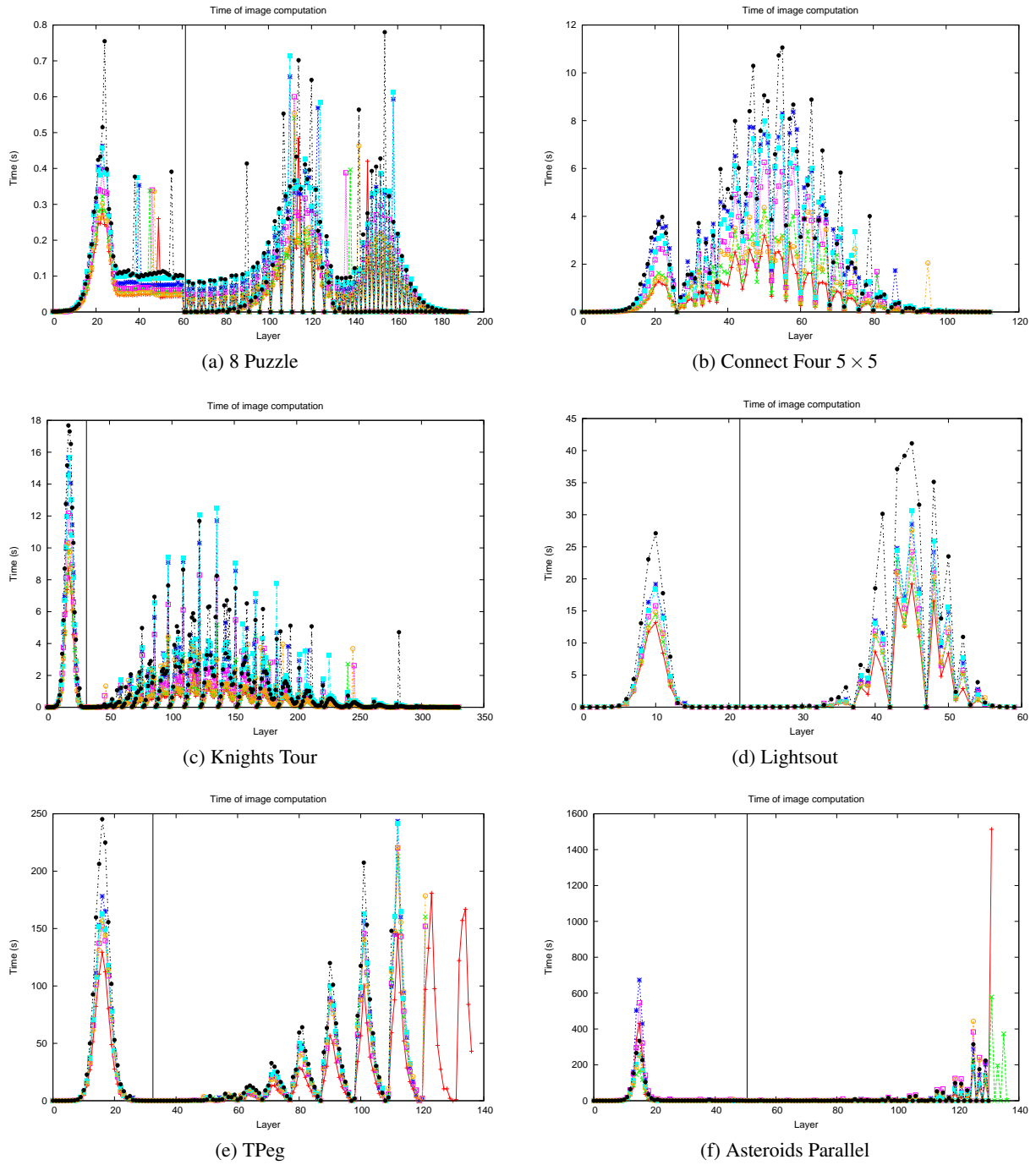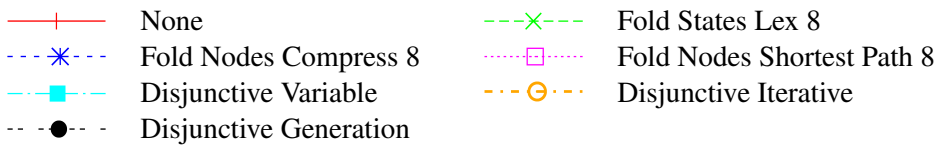
(a) 8 Puzzle

(b) Connect Four 5×5

(c) Knights Tour

(d) Lightsout

(e) TPeg

(f) Asteroids Parallel

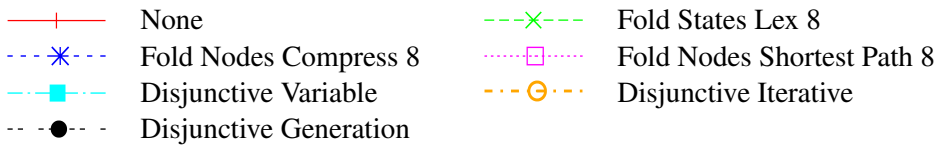Figure 6: Total time used for each partitioning algorithm in each layer.

	None		Fold States Lex 8
	Fold Nodes Compress 8		Fold Nodes Shortest Path 8
	Disjunctive Variable		Disjunctive Iterative
	Disjunctive Generation		

(a) 8 Puzzle

(b) Connect Four 5×5

(c) Knights Tour

(d) Lightsout

(e) TPeg

(f) Asteroids Parallel

Figure 7: Total number of nodes used for each partitioning algorithm in each layer.

(a) 8 Puzzle

(b) Connect Four 5×5

(c) Knights Tour

(d) Lightsout

(e) TPeg

(f) Asteroids Parallel

Figure 8: Maximum image size used for each partitioning algorithm in each layer.

None

Fold States Lex 8

Fold Nodes Compress 8

Fold Nodes Shortest Path 8

Disjunctive Variable

Disjunctive Iterative

Disjunctive Generation

with the BDDs in the input but also with the number of states represented [22]. The core advantage of partitioned BDDs is that one can gain depth in some of the partitions and locate errors fast. A complete search of such prioritization is involved, as information for backtracking has to be maintained. For this case lexicographic partitioning can help to gain a better search control. Some potential may be obtained in using different variable orderings in different partitionings, but this will make the algorithms more complicated.

Previous work on improved BDD partitioning and guided exploration in AI includes work by [9, 11] that partitions the state sets along different g- and h-values and along the difference in the h-value. Using a lex-partitioning in the state sets prior and posterior to computing the image, the search can be distributed. Each computing node is responsible for computing images in the lexicographical window it is assigned to. As the images are computationally expensive, sending around the BDDs is negligible. By adapting the window sizes different forms of dynamic load balancing are immediate.

Explicit search can be more space-efficient if perfect hash functions are available. With ranking and unranking we can eventually connect a symbolic state space representation with BDDs and an explicit bitvector based exploration. The BDDs can serve as a basis for a linear-time ranking and unranking scheme. As we have control over the number of states in a BDD, we can switch between symbolic and explicit state space generation when the available main memory is sufficient to cover the partitioned state sets. This provides a combination of the two methods, where the BDDs are used to define hash functions for addressing states in the bitvector representation of the state space.

In game playing we can think of a layered approach to perform forward search with a BDD and retrograde analysis that changes from symbolic to explicit-state representation to strongly solve a game, by means that the solvability status for all states is computed. In case of exploring domains with complex cost functions in AI planning, a breadth-first BDD enumeration might be feasible, while computing the optimal cost is much harder.

As the computation of the reachability set is done in compact form the problem of invalid (unreachable) states in the backward traversal can be avoided. Due to the partial description of the goals there are many planning domains where the set of backward reachable states is much larger than the one in forward search. Consider the sliding-tiles puzzle with n tiles and with the blank position not mentioned in the goal state. The inverse of planning operators that move a tile has the position of the blank and the tile to be moved in the precondition, and the exchange of tile and blank in the effects. Since the blank position is not known to the planner backward exploration will generate states with tiles on top of each other, so that with n^n the set of backward reachable states is exponentially larger than the number of $n!/2$ forward reachable states.

References

[1] Louis Victor Allis (1994): *Searching for Solutions in Games and Artificial Intelligence*. Ph.D. thesis, Rijksuniversiteit Limburg te Maastricht.

[2] Marcel Ball & Robert C. Holte (2008): *The Compression Power of Symbolic Pattern Databases*. In: *ICAPS*, pp. 2–11.

[3] Fabiano C. Botelho, Rasmus Pagh & Nivio Ziviani (2007): *Simple and Space-Efficient Minimal Perfect Hash Functions*. In: *WADS*, pp. 139–150, doi:10.1007/978-3-540-73951-7_13.

[4] Teresa M. Breyer & Richard E. Korf (2010): *1.6-Bit Pattern Databases*. In: *AAAI*, pp. 39–44. Available at http://www.aaai.org/ocs/index.php/AAAI/AAAI10/paper/view/1850.

[5] Randal E. Bryant (1985): *Symbolic Manipulation of Boolean Functions Using a Graphical Representation*. In: *22nd Design Automation Conference (DAC)*, ACM Press, pp. 688–694, doi:10.1145/317825.317964.

[6] Randal E. Bryant (1986): *Graph-Based Algorithms for Boolean Function Manipulation*. IEEE Transactions on Computers 35(8), pp. 677–691, doi:10.1109/TC.1986.1676819.

[7] Alessandro Cimatti, Marco Pistore, Marco Roveri & Paolo Traverso (2003): *Weak, Strong, and Strong Cyclic Planning via Symbolic Model Checking*. Artificial Intelligence 147(1–2), pp. 35–84, doi:10.1016/S0004-3702(02)00374-0.

[8] Martin Dietzfelbinger & Stefan Edelkamp (2009): *Perfect Hashing for State Spaces in BDD Representation*. In: *KI*, pp. 33–40, doi:10.1007/978-3-642-04617-9_5.

[9] Stefan Edelkamp & Frank Reffel (1998): *OBDDs in Heuristic Search*. In: *KI*, pp. 81–92, doi:10.1007/BFb0095430.

[10] Subramanian Iyer, Debashis Sahoo, E. Allen Emerson & Jawahar Jain (2006): *On Partitioning and Symbolic Model Checking*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 25(5), pp. 780–788, doi:10.1109/TCAD.2006.870410.

[11] Rune M. Jensen, Manuela M. Veloso & Randal E. Bryant (2008): *State-Set Branching: Leveraging BDDs for Heuristic Search*. Artificial Intelligence 172(2-3), pp. 103–139, doi:10.1016/j.artint.2007.05.009.

[12] Peter Kissmann & Stefan Edelkamp (2010): *Layer-Abstraction for Symbolically Solving General Two-Player Games*. In: *SoCS'10*, pp. 63–70.

[13] Richard E. Korf (2008): *Minimizing Disk I/O in Two-Bit Breadth-First Search*. In: *AAAI*, pp. 317–324.

[14] Nathaniel C. Love, Timothy L. Hinrichs & Michael R. Genesereth (2006): *General Game Playing: Game Description Language Specification*. Technical Report LG-2006-01, Stanford Logic Group.

[15] Kenneth L. McMillan (1992): *Symbolic Model Checking: An Approach to the State Explosion Problem*. Ph.D. thesis, Carnegie Mellon University.

[16] Kenneth L. McMillan (1993): *Symbolic Model Checking*. Kluwer Academic Publishers, doi:10.1007/978-1-4615-3190-6.

[17] Shin-ichi Minato, Nagisa Ishiura & Shuzo Yajima (1990): *Shared Binary Decision Diagram with Attributed Edges for Efficient Boolean Function Manipulation*. In: *DAC*, ACM Press, pp. 52–57, doi:10.1109/DAC.1990.114828.

[18] Amit Narayan, Jawahar Jain, M. Fujita & A. Sangiovanni-Vincentelli (1996): *Partitioned ROBDDs – A Compact, Canonical and Efficiently Manipulable Representation for Boolean Functions*. In: *ICCAD*, pp. 547–554, doi:10.1109/ICCAD.1996.569909.

[19] John W. Romein & Henri E. Bal (2002): *Awari is Solved*. International Computer Games Association (ICGA) Journal 25(3), pp. 162–165.

[20] Debashis Sahoo, Subramanian K. Iyer, Jawahar Jain, Christian Stangier, Amit Narayan, David L. Dill & E. Allen Emerson (2004): *A Partitioning Methodology for BDD-Based Verification*. In: *Formal Methods in Computer-Aided Design*, pp. 399–413, doi:10.1007/978-3-540-30494-4_28.

[21] Fabio Somenzi (2009): *CUDD: CU Decision Diagram Package, Release 2.4.2*. Available at http://vlsi.colorado.edu/~fabio/CUDD/.

[22] Bwolen Yang, Randal E. Bryant, David R. O'Hallaron, Armin Biere, Olivier Coudert, Geert Janssen, Rajeev K. Ranjan & Fabio Somenzi (1998): *A Performance Study of BDD-Based Model Checking*. In: *FMCAD*, pp. 255–289, doi:10.1007/3-540-49519-3_18.