Basic Research in Computer Science

BRICS RS-95-29

N. Klarlund: An n log n Algorithm for Online BDD Refinement

An n log n Algorithm for Online BDD Refinement

Nils Klarlund

BRICS Report Series **RS-95-29**

ISSN 0909-0878 **May 1995**

**Copyright © 1995, BRICS, Department of Computer Science
University of Aarhus. All rights reserved.**

**Reproduction of all or part of this work
is permitted for educational or research use
on condition that this copyright notice is
included in any copy.**

See back inner page for a list of recent publications in the BRICS Report Series. Copies may be obtained by contacting:

**BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK - 8000 Aarhus C
Denmark**

**Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk**

BRICS publications are in general accessible through WWW and anonymous FTP:

`http://www.brics.dk/`
`ftp ftp.brics.dk (cd pub/BRICS)`

An $n \log n$ algorithm for online BDD refinement

Nils Klarlund
BRICS*
Department of Computer Science
University of Aarhus
Ny Munkegade
DK-8000 Aarhus C, Denmark

Abstract

Binary Decision Diagrams are in widespread use in verification systems for the canonical representation of Boolean functions. A BDD representing a function $\varphi : \mathbb{B}^\nu \to \mathbb{N}$ can easily be reduced to its canonical form in linear time.

In this paper, we consider a natural online BDD refinement problem and show that it can be solved in $O(n \log n)$ if n bounds the size of the BDD and the total size of update operations.

We argue that BDDs in an algebraic framework should be understood as minimal fixed points superimposed on maximal fixed points. We propose a technique of controlled growth of equivalence classes to make the minimal fixed point calculations be carried out efficiently. Our algorithm is based on a new understanding of the interplay between the splitting and growing of classes of nodes.

We apply our algorithm to show that automata with exponentially large, but implicitly represented alphabets, can be minimized in time $O(n \cdot \log n)$, where n is the total number of BDD nodes representing the automaton.

1 Introduction

Binary Decision Diagrams [1] form the backbone of many symbolic methods for verification of hardware and software. BDDs are essentially acyclic automata whose state spaces are shrunk by a technique called *path compression*. In this paper, we study a fundamental algorithmic problem that have applications to the efficient representation of automata on large alphabets.

Given a BDD representing a function $\varphi : \mathbb{B}^\nu \to \mathbb{N}$ and an initial partition of its codomain, we formulate an algorithm that maintains the induced canonical

*Basic Research in Computer Science, Centre of the Danish National Research Foundation.

partition of all its nodes after each update operation specifying a refinement of the current partition. A simple algorithm based on the linear time reduction of BDDs [9] implies that each node is touched potentially as many times as the number of operations. Thus an $O(n^2)$ algorithm arises.

In this paper, we formulate an $O(n \cdot \log n)$ algorithm based on an algebraic analysis of BDDs. We introduce the concept of *decision partition* to analyze the result of what is known as a *Split* operation in partition refinement algorithms such as [8]. BDDs cannot, however, be analyzed simply as usual partition refinements. In fact, we show that the split operation must be followed by a *Grow* operation. The *Grow* operation cannot be used with Hopcroft's "process the lesser half" strategy [4], since all decision blocks must be grown as opposed to the situation in partition refinement algorithms, where the largest blocks created can be ignored.

Fortunately, a variation *FGrow* of the grow operation allows certain blocks resulting from the normal *Grow* operation to be fused. Even though information is lost, we are able to show that if a partition is a fixed point under *Split* and *FGrow*, then it is also a fixed point under *Split* and *Grow*.

In our online algorithm, we show that if a large block is being calculated, then it can be thrown away by being fused with another block, while the expense of calculating it can be attributed to a third block known to be small.

It has been known for a long time [4] that deterministic finite-state automata can be minimized in time $O(m \cdot n \log n)$, where n is the number of states and m is the size of the input alphabet. BDDs allow automata with n states and 2^n letters—each inducing a different behavior in the automaton—to be represented by graphs of polynomial size in n; see [3], where also a relatively straightforward $O(n^2)$ minimization algorithm is presented together with its application to a practical implementation of Monadic Second-order Logic on Strings.

We show that an easy application of our online BDD refinement algorithm allows minimization to be carried out in only $O(n \cdot \log n)$ steps, where n is the size of the representation. To our knowledge, the only other algorithm for large alphabets that reach a similar bound is that of [2], where incompletely specified transition functions are considered. The compression possible with the BDD representation is exponentially greater.

It should also be noted here that if automata are represented with BDDs that are not path compressed, then an $O(n \log n)$ algorithm follows easily by considering the automaton as working on words over $\mathbb{B}$ [7]. Path compression, however, seems to be of major practical significance although the asymptotic gain is only slight [6].

Finally, we mention that online minimization of automata on large, implicitly represented state spaces (not alphabets) have been considered in [5]. Online minimization here refers to incremental exploration of the state space. This algorithm bears a superficial resemblance to ours in that it also alternates between minimal and maximal fixed point iterations.

Overview

In Section 2, we define the online BDD refinement problem and we develop an algebraic treatment of BDD properties. A simpe online BDD refinement algorithm when path compression is omitted is discussed in Section 3. In Section 4, we use the properties of Section 2 to extend the simple algorithm to the case of BDDs with path compression. Section 5 discusses the application to BDD represented automaton minimization.

2 Online BDD Refinement

Assume we are given a set $x_0, x_1, \ldots, x_{\nu-1}$ of Boolean variables. A *truth assignment* to these variables is a vector $\vec{u} \in \mathbb{B}^\nu$. An *assignment prefix* $\vec{u}$ up to i is a truth assignment to variables $x_0, \ldots x_i$. A *Binary Decision Diagram* or *BDD* φ is a rooted directed graph. The root is named $\hat{}\varphi$. Each node v in φ is either an *internal node* or a *leaf*. An internal node possesses an *index* denoted $v.i$. Also, it contains edges $v \cdot 0$, which points to a node called the *low successor* of v, and $v \cdot 1$, which points to the *high successor*. The index of a successor of v is always greater than the index of v. A leaf has no successors and no index. Let the set of leaves be $\mathcal{L}$. The graph φ denotes a function, also called φ, from $\mathbb{B}^\nu \to \mathcal{L}$. To calculate $\varphi(\vec{x})$, one starts at the root. If the root is a leaf, then the value $\varphi(\vec{x})$ is the root; otherwise, let i be the index of the root. If x_i is 1 then go to the high successor and if x_i is 0 go to the low successor. Continue in this way until a leaf is reached. This leaf is the value of $\varphi(\vec{x})$. (Since there may be jumps greater than one in the index of some of the variables, some of the values in the assignment may be irrelevant.) In general, if v is a node of index i and $\vec{u}$ is a value assignment to $x_i, \ldots, x_j$, then $v \cdot \vec{u}$ denotes the node reached by following $\vec{u}$ from v.

The BDD φ defines a partition $\equiv_\varphi$ of assignment prefixes given by $\vec{u} \equiv_\varphi \vec{u}'$ if $\hat{}\varphi \cdot \vec{u} \equiv_\varphi \hat{}\varphi \cdot \vec{u}'$.

We shall consider the case where the leaves are used to differentiate between finer and finer partitions of $\mathbb{B}^\nu$. The partition is given by a *leaf discriminator* $D : \mathcal{L} \to \mathbb{N}$, which is implemented by having field $v.d$ denote the value $D(v)$ for a leaf v. In this way, a function $D \circ \varphi$ is defined by $D \circ \vec{\varphi}(\vec{x}) = v.d$, where v is the value of $\varphi(\vec{x})$. Two assignments $\vec{x}$ and $\vec{y}$ are then equivalent if $D \circ \varphi(\vec{x}) = D \circ \varphi(\vec{y})$.

BDDs may also be shared. For example, we use $\vec{\varphi} = \varphi_0, \cdots, \varphi_{n-1}$ to denote a directed graph with roots $\hat{}\varphi_i$ such that the nodes reachable from each root constitute a BDD. If D is a discriminator for the leaves, then we say that $R : [n] \to \mathbb{N}$ is a *function discriminator* for $D \circ \vec{\varphi}$ if $D \circ \varphi_i = D \circ \varphi_j$ iff $R(i) = R(j)$. Note that if D is a constant discriminator (i.e. if D is a constant function), then all $D \circ \varphi_i$ are equivalent.

The BDD online refinement problem is to maintain a function discriminator

R for $D \circ \vec{\varphi}$ when D is updated piecemeal. Each update operation specifies a partial mapping $E : \mathcal{L} \to \mathbb{N}$, which defines the change to D. In order to assure that the new D specifies a partition refining the one given by the current D, we require that the range of E is disjoint from the range of the current D. Thus the desired functionality can be summarized as follows.

Multiple BDD Online Problem
Input: n shared BDDs $\vec{\varphi}$ with leaves $\mathcal{L}$ and constant discriminator D.
Maintained : A functional discriminator R of length n.
Update: A partial mapping $E : \mathcal{L} \to \mathbb{N}$ such that **range**D does not intersect the current discriminator. The current discriminator D is updated according to E. After each update operation, the contents of R discriminates $D \circ \vec{\varphi}$. The size of operation E is the size of domain(E).
Output: A list of numbers i for which $R(i)$ has changed.

In Section 5, we prove:

Theorem 1 Multiple BDD Online Refinement can be solved in time $O(n \min(k, \log n) + k)$, where n is the number of nodes in the BDDs and k is the total size of all operations. Thus, if n also bounds k, then the algorithm is $O(n \log n)$.

The Canonical BDD

We define the canonical BDD for function $\psi : \mathbb{B}^\nu \to D$, where D is finite, as follows. A *partial assignment $\vec{u}$ from i to j* is a truth assignment to variables $x_i, \ldots, x_j$. The partial assignment $\vec{u}$ may be narrowed to a partial assignment from i' to j', where $i \le i' \le j' \le j$. It is denoted $\vec{u}[i'..j']$. If only a prefix of $\vec{u}$ up to $i' - 1$ is cut off, we write $\vec{u}[i'..]$. An *extension $\vec{v}$ of $\vec{u}$ up to j* is a partial assignment from $i + 1$ to j. A *full* extension is one that assigns up to $\nu - 1$. For any assignment prefix $\vec{u}$ up to i, we may consider the *residue* function $\psi_{\vec{u}} : \vec{v}' \mapsto \psi(\vec{u}\vec{v}')$, where $\vec{v}'$ is a full extension. Define $\vec{u} \sim_\psi \vec{u}'$ if $\psi_{\vec{u}} = \psi_{\vec{u}'}$. The equivalence class of $\vec{u}$ is denoted $[\vec{u}]_\psi$. In particular, if $\vec{u} \sim_\psi \vec{u}'$ then $\vec{u}$ and $\vec{u}'$ are assignment prefixes up to i, which is called the *index* of the equivalence class $[\vec{u}]_\psi = [\vec{u}']_\psi$.

The equivalence classes of $\sim_\psi$ correspond to the states of a canonical automaton that upon reading a value assignment is in a state designating the value of ψ.

The *path compression* of BDDs can now be understood as a least fixed point calculation that involves coalescing equivalence classes. If $[\vec{u}0]_\psi = [\vec{u}1]_\psi$, then $[\vec{u}]_\psi$ and $[\vec{u}0]_\psi = [\vec{u}1]_\psi$ are coalesced. Note that if also $[\vec{v}0]_\psi = [\vec{v}1_\psi]$ for some $\vec{v}$, then this identity still holds after $[\vec{u}]_\psi$ and $[\vec{u}0]_\psi = [\vec{u}1]_\psi$ are coalesced. Thus there is a unique least fixed point reached by repeatedly coalescing $\sim_\psi$ classes. The equivalence classes of the resulting partition $\approx_\psi$ is the *canonical* BDD for ψ.

Each such new equivalence class M consists of a number of equivalence classes of $\sim_\psi$. The index $M.i$ of M is defined as the highest index of an old class. It can be seen that there is at most one old class in M of highest index. The high successor $M.1$, defined if the index is less than ν, is the equivalence class of $\vec{u} \cdot 1$, where $\vec{u}$ is a prefix of maximal length in M. The low successor is defined similarly.

Lemma 1 Consider i and $\vec{u}$. The residue function $\psi_{\vec{u}.\vec{v}}$ is the same function for all extensions $\vec{v}$ up to i if and only if $u \approx_\psi u \cdot v$ for all such v.

Proof By induction on the length of $\vec{v}$. □

Lemma 2 $\equiv_\varphi$ refines $\approx_\varphi$.

Proof Assume $\vec{u} \equiv_\varphi \vec{u}'$. If $\vec{u}$ and $\vec{u}'$ have the same length, then $\vec{u} \sim_\varphi \vec{u}'$ and thus $\vec{u} \approx_\varphi \vec{u}'$. Otherwise, if $\vec{u}$ assigns up to i and $\vec{u}'$ up to j, with $i < j$, then the prefix $\vec{u}''$ of $\vec{u}'$ up to i is also equivalent (modulo $\equiv_\varphi$) to $\vec{u}$ and all extensions $\vec{v}$ up to j make $\vec{u}\vec{v}$ and $\vec{u}''\vec{v}$ equivalent (modulo $\equiv_\varphi$) to $\vec{u}$. In particular, all residues are the same, hence by the preceding lemma, $\vec{u} \approx_\varphi \vec{u}''\vec{v}$, where we choose $\vec{v}$ to be the extension of to j such that $\vec{u}''\vec{v} = \vec{u}'$.

□

Partitions of BDD Nodes

A *partition* $\mathcal{P}$ of a BDD φ is a set of non-empty, disjoint subsets or *blocks* of nodes, whose union is the set of all nodes. Alternatively, $\mathcal{P}$ may be viewed as an equivalence relation $\equiv_\mathcal{P}$ defined by $v \equiv_\mathcal{P} v'$ iff $\exists B \in \mathcal{P} : v, v' \in B$. Since any assignment prefix $\vec{u}$ leads to a unique node $\varphi, \equiv_\mathcal{P}$ induces an equivalence relation on assignment prefixes that is also denoted $\equiv_\mathcal{P}$.

To simplify matters, we assume in the following that *all partitions are over the same BDD* φ. Also for simplicity, we shall often write $\mathcal{P}$ for $\equiv_\mathcal{P}$.

A partition $\mathcal{Q}$ defines a function $\varphi_\mathcal{Q} : \mathbb{B}^\nu \to D$, where we define a labeling D of the leaves of φ such that for leaves v and v', $D(v) = D(v')$ iff $v \equiv_\mathcal{Q} v'$. The canonical BDD for this function is denoted $\approx_\mathcal{Q}$. Note that it is only dependent on the partition of the leaves defined by $\mathcal{Q}$. In particular, it is not necessarily the case that $\approx_\mathcal{Q}$ refines $\equiv_\mathcal{Q}$, since $\mathcal{Q}$ may have introduced too fine distinctions above the leaves. We usually regard the canonical BDD $\approx_\mathcal{Q}$ as a partition of the nodes of φ.

Decision Partitions

An important part of our algorithm is to work with partitions that only become refinements of canonical partitions after moving nodes around.

A node v is a *decision node* if it is a leaf or it has at least one successor outside its own block. Any other node is *redundant*.

A *decision partition* $\mathcal{M}$ of a partition $\mathcal{Q}$ specifies a partition of the decision nodes of each block B in $\mathcal{Q}$ into *decision blocks*. Any decision block M must contain nodes of the same index. If for each B, all decision notes of B are gathered in just one decision block, then $\mathcal{M}$ is said to be the *stable* decision partition.

The *Split* Operator

Given $\mathcal{Q}$, we can form a decision partition $\mathcal{M} = \mathit{Split}(\mathcal{Q})$ as follows. For every block B, put all decision nodes v with the same index and the same behavior, i.e. having the same equivalence classes determined by $v \cdot 0$ and $v \cdot 1$, in the same decision block. All leaves in B are also put into a decision block. Formally, $\mathcal{M}$ is defined as

$$\begin{aligned}\{M \neq \emptyset \mid \quad & \exists B, B_0, B_1 \in \mathcal{M} : \exists i : \\ & M = \{v \mid v \in B \text{ and } v \text{ is a leaf}\} \text{ or} \\ & M = \{v \mid v.i = i \text{ and } v \in B,\ v0 \in B_0, \text{ and } v1 \in B_1\}\}\end{aligned}$$

Partition $\mathcal{Q}$ is *stable* if $\mathit{Split}(\mathcal{Q})$ is the stable decision partition. This amounts to saying that $\mathcal{Q}$ is a fixed point under $\mathit{Grow} \circ \mathit{Split}$, i.e. $\mathit{Grow} \circ \mathit{Split}(\mathcal{Q}) = \mathcal{Q}$.

Note that if $\mathcal{Q}$ is stable, then both successors of any non-leaf decision node are outside its own block (for if some block B contained a decision node v with only one successor not in B, then by following successors from v, we would eventually reach a decision node in B that is in a different decision block from that of v, but that would contradict that $\mathcal{Q}$ is stable).

Note also that $\approx_{\mathcal{Q}}$ is stable.

The *Grow* Operator

For any node v and any extension $\vec{u}$, there will be a first decision node w in some decision block M along u. In this case, we say that extension $\vec{u}$ from v *hits* M. In particular, if $v \in M$, then any extension hits M.

If M is a decision block, then its *closure*, denoted $\mathit{Cl}(\mathcal{Q}, M)$, is the set of nodes all of whose extensions hit M. Clearly, if M is contained in a block B, then $\mathit{Cl}(\mathcal{Q}, M)$ is also contained in B. Also if M and M' are different decision blocks, then $\mathit{Cl}(\mathcal{Q}, M)$ and $\mathit{Cl}(\mathcal{Q}, M')$ are disjoint. For each block B, let the *remainder*, denoted $\mathit{Rem}(\mathcal{Q}, \mathcal{M}, B)$, be defined as B minus all nodes in $\mathit{Cl}(M)$, where M is contained in B, i.e. $\mathit{Rem}(\mathcal{Q}, \mathcal{M}, B) = B \setminus \bigcup_{M \in \mathcal{M}, M \subseteq B} \mathit{Cl}(\mathcal{Q}, M)$. Then, $\mathit{Cl}(\mathcal{Q}, M), M \in \mathcal{M}$, together with $\mathit{Rem}(\mathcal{Q}, \mathcal{M}, B), B \in \mathcal{Q}$, form a partition, called $\mathit{Grow}(\mathcal{Q}, \mathcal{M})$.

Sometimes it is convenient to assume that the result of $\mathit{Split}(\mathcal{Q})$ includes the argument Q. In this way, we may apply a *Grow* operator after a *Split* as in $\mathit{Grow} \circ \mathit{Split}(\mathcal{Q})$, which denotes $\mathit{Grow}(\mathcal{Q}, \mathit{Split}(\mathcal{Q}))$.

Lemma 3 (a) $\mathit{Grow} \circ \mathit{Split}(\mathcal{Q})$ refines $\mathcal{Q}$.
(b) If $\mathcal{P}$ refines $\mathcal{Q}$, then $\mathit{Grow} \circ \mathit{Split}(\mathcal{P})$ refines $\mathit{Grow} \circ \mathit{Split}(\mathcal{Q})$.

Proof (a) Let $\mathcal{P}' = \mathit{Grow} \circ \mathit{Split}(\mathcal{P})$. Since $\mathcal{P}'$ is gotten from $\mathcal{P}$ by carving out closures of decision blocks, it follows that $\mathcal{P}'$ refines $\mathcal{P}$.

(b) Let $\mathcal{M} = \mathit{Split}(\mathcal{P})$, $\mathcal{N} = \mathit{Split}(\mathcal{Q})$, $\mathcal{P}' = \mathit{Grow} \circ \mathit{Split}(\mathcal{P})$, and $\mathcal{Q}' = \mathit{Grow} \circ \mathit{Split}(\mathcal{Q})$. It can be seen that it is sufficient to prove that each closure with respect to $\mathcal{Q}$ of a decision block N in $\mathcal{N}$ is a union of closures of decision blocks M of $\mathcal{M}$. This is established by showing that each decision block of $\mathcal{N}$ is a union of decision blocks of $\mathcal{M}$. The details are omitted. □

Lemma 4 Let $\mathcal{P}$ be a stable partition and let $v \equiv_{\mathcal{P}} v'$, where v is of index i and v' of index j with $i \leq j$. Then for any extension $\vec{u}$ from v, $v \cdot \vec{u} \equiv_{\mathcal{P}} v' \cdot \vec{u}[j..]$.

Proof Let $v, v' \in B \in \mathcal{P}$. We proceed by an inductive argument. If the decision nodes of B are leaves, then clearly $v \cdot \vec{u} \equiv_{\mathcal{P}} v$ and $v' \cdot \vec{u}[j..] \equiv_{\mathcal{P}} v' \cdot \vec{u}[j..]$, whence $v \cdot \vec{u} \equiv_{\mathcal{P}} v' \cdot \vec{u}[j..]$.

Otherwise, all decision nodes of B point to blocks below B, so we assume by induction that the Lemma holds for all blocks below B. We must now show that it holds for v, v' in B. Now there is an h such that $v \cdot \vec{u}[..h]$ and $v' \cdot \vec{u}[j..h]$ are the first nodes outside B from v and v' along $\vec{u}$ and $\vec{u}[j..]$ (unless both $v \cdot \vec{u}$ and $v' \cdot \vec{u}[j..]$ are in B, which is a trivial case). But by assumption that $\mathcal{P}$ is stable

$$v \cdot \vec{u}[..h] \equiv_{\mathcal{P}} v \cdot \vec{u}[j..h].$$

Thus, by inductive hypothesis

$$v \cdot \vec{u} = v \cdot \vec{u}[..h] \cdot \vec{u}[h+1..] \equiv_{\mathcal{P}} v' \cdot \vec{u}[j..h] \cdot \vec{u}[h+1..] = v' \cdot \vec{u}[j..].$$

□

Let $\mathcal{M}$ be a decision partition of $\mathcal{Q}$. We say that $\mathcal{M}$ *respects* a partition $\mathcal{P}$ if whenever v and v' in are different decision blocks of $\mathcal{M}$, they are in different blocks of $\mathcal{P}$.

Lemma 5 Let stable $\mathcal{P}$ refine $\mathcal{Q}$ and let $\mathcal{M}$ be a decision partition of $\mathcal{Q}$ respecting $\mathcal{P}$. Then $\mathcal{P}$ refines $\mathit{Grow}(\mathcal{Q}, \mathcal{M})$.

Proof Let $\mathcal{Q}' = \mathit{Grow}\ (\mathcal{Q}, \mathcal{M})$. It can be seen that it is sufficient to consider $v, v' \in B \in \mathcal{Q}$ with $v \equiv_{\mathcal{P}} v'$. We must prove that $v \equiv_{\mathcal{Q}'} v'$. We establish this by proving that any extension hits the same decision block in $\mathcal{M}$ whether followed from v or v'. Assume that $i \leq j$ where $i = v.i$ and $j = v.i$. Consider an extension $\vec{u}$ from v such that $v \cdot \vec{u}$ is the first decision node met along $\vec{u}$. There are now three cases.

Case 1. The node $v \cdot \vec{u}$ is a leaf. Then $v' \cdot \vec{u}[j..]$ is a leaf. If they are in different blocks of $\mathcal{M}$, then—since $\mathcal{M}$ respects $\mathcal{P}$—they are in different blocks of $\mathcal{P}$, but that contradicts Lemma 4.

Case 2. No decision node is encountered along $v' \cdot \vec{u}[j..]$ and both $v \cdot \vec{u}$ and $v' \cdot \vec{u}[j..]$ are not leaves. Now, since $v \cdot \vec{u}$ is a decision node, either $v \cdot \vec{u} \cdot 0$ or $v \cdot \vec{u} \cdot 1$ is not in B. Thus there is an extension $\vec{u}'$ such that $v \cdot \vec{u} \cdot \vec{u}'$ is not in B while $v' \cdot \vec{u}[j..]\vec{u}'$ is the first decision node in B encountered from v'. Thus $v \cdot \vec{u} \cdot \vec{u}'$ and $v' \cdot (\vec{u} \cdot \vec{u}')[j..]$ are in different blocks of $\mathcal{Q}$, which is a contradiction for the same reason as above.

Case 3. A decision node is encountered along $v' \cdot \vec{u}[j..]$ and $v \cdot \vec{u}$ and $v' \cdot [j..]$ are not leaves. Then reasoning similar to that of Case 2 applies. □

Lemma 6 If stable $\mathcal{P}$ refines $\mathcal{Q}$, then $\mathcal{P}$ respects $\mathit{Split}(\mathcal{Q})$.

Proof Nodes v and v' equivalent in $\mathcal{Q}$ can become inequivalent in $\mathit{Split}(\mathcal{Q})$ only if $v0$ and $v'0$ or $v1$ and $v'1$ are inequivalent in $\mathcal{Q}$. But then v and v' cannot be equivalent in $\mathcal{P}$ since $\mathcal{P}$ is assumed to be stable and assumed to refine $\mathcal{Q}$. □

Proposition 1 If stable $\mathcal{P}$ refines $\mathcal{Q}$, then $\mathcal{P}$ refines $\mathit{Grow} \circ \mathit{Split}(\mathcal{Q})$.

Proof By Lemma 5 and by Lemma 6. □

Proposition 2 If $\mathcal{Q} = \mathit{Grow} \circ \mathit{Split}(\mathcal{Q})$, then $\mathcal{Q}$ refines $\approx_{\mathcal{Q}}$.

Proof For $v, v' \in B \in \mathcal{Q}$, we must prove that $v \approx_{\mathcal{Q}} v'$. We proceed by induction and prove in addition that all decision nodes in B have the same index.

If v and v' are leaves, then certainly $v \approx_{\mathcal{Q}} v'$ and v and v' have the same index.

If v and v' are decision nodes of B, then by assumption that $\mathcal{Q} = \mathit{Grow} \circ \mathit{Split}(\mathcal{Q})$, they have the same index and behave similarly with respect to hitting lower classes of $\mathcal{Q}$ along their 0 and 1 successor. By inductive assumption, lower classes are contained in $\approx_{\mathcal{Q}}$ classes. Thus, the mappings $\vec{w} \mapsto [v \cdot \vec{w}]_{\approx_{\mathcal{Q}}}$ and $\vec{w} \mapsto [v' \cdot \vec{w}]_{\approx_{\mathcal{Q}}}$ are the same and thus $v \approx_{\mathcal{Q}} v'$.

If v is a redundant node of B at level j and all decision nodes of B are of index i in a block M of $\approx_{\mathcal{Q}}$, then $\vec{w} \mapsto [v \cdot \vec{u} \cdot \vec{w}]_{\approx_{\mathcal{Q}}}$ is the same function for all extensions from v up to i, since v is contained in the closure of the decision nodes of B by assumption that $\mathcal{Q} = \mathit{Grow} \circ \mathit{Split}(\mathcal{Q})$. Thus $v \in M$ by Lemma 1. □

Proposition 3 If $\approx_{\mathcal{Q}}$ refines $\mathcal{Q}$ and if $\mathcal{Q}' = (\mathit{Grow} \circ \mathit{Split})^i(\mathcal{Q})$ is stable, then $\mathcal{Q}'$ is $\approx_{\mathcal{Q}}$.

Proof By Proposition 2, $\mathcal{Q}'$ refines $\approx_{\mathcal{Q}}$. By repeated applications of Proposition 1, $\approx_{\mathcal{Q}}$ refines $\mathcal{Q}'$. □

The *FGrow* Operator

The *FGrow* operator is defined as $Grow(\mathcal{Q}, \mathcal{M})$ except that for each block B of $\mathcal{Q}$, $Rem(\mathcal{Q}, \mathcal{M}, B)$ may or may not be fused with some designated $Cl(\mathcal{Q}, M)$ with M a decision block in B. Thus the operation is not fully specified, but whether fusion takes place or not and with which $Cl(\mathcal{Q}, M)$ will be inconsequential for the properties to follow. Even though information since to be dropped by *FGrow*, a fixed point involving *FGrow* is also a fixed point involving *Grow*.

Proposition 4 If $FGrow \circ Split(\mathcal{Q}) = \mathcal{Q}$, then $Grow \circ Split(\mathcal{Q}) = \mathcal{Q}$.

Proof Assume $FGrow \circ Split(\mathcal{Q}) = \mathcal{Q}$. Let $\mathcal{M} = Split(\mathcal{Q})$. We prove that for each $B \in \mathcal{Q}$, $Rem(\mathcal{Q}, \mathcal{M}, B)$ is empty. For a contradiction, assume that $v \in Rem(\mathcal{Q}, \mathcal{M}, B)$. Then there are at least two decision blocks of $\mathcal{M}$ in B. Therefore, there is at least one decision block M such that $Cl(\mathcal{Q}, M)$ is not fused with $Rem(\mathcal{Q}, \mathcal{M}, B)$. But this contradicts that $FGrow \circ Split(\mathcal{Q}) = \mathcal{Q}$.

Since all remainder sets are empty, the effect of *FGrow* is the same as that of *Grow* on $Split(\mathcal{Q})$. Thus, $Grow \circ Split(\mathcal{Q}) = \mathcal{Q}$. □

Theorem 2 If $\approx_{\mathcal{Q}}$ refines $\mathcal{Q}$ and if $\mathcal{Q}' = (FGrow \circ Split)^i(\mathcal{Q})$ is stable, then $\mathcal{Q}'$ is $\approx_{\mathcal{Q}}$.

Proof The partition $\approx_{\mathcal{Q}}$ certainly refines $\mathcal{Q}'$ since $(FGrow \circ Split)^i(\mathcal{Q})$ is coarser than $(Grow \circ Split)^i(\mathcal{Q})$ by Lemma 3(b). On the other hand, $\mathcal{Q}'$ is a fixed point for $Grow \circ Split$ by the preceding proposition. So $\mathcal{Q}'$ refines $\approx_{\mathcal{Q}}$ by Proposition 2. □

For our purposes, it is convenient to represent the partition of the leaves as a decision partition $\mathcal{M}$ of a partition $\mathcal{Q}$. Such a partition where the only non-trivial decision blocks are those that contain leaves is called a *leaf partition*. A canonical equivalence relation $\approx_{\mathcal{M}}$ is defined as before for $\approx_{\mathcal{Q}}$. We will regard an update operation as specifying a leaf partition and $\mathcal{Q}$ will be the current partition.

Theorem 2 then can be formulated

Theorem 3 If $\approx_{\mathcal{M}}$ refines $\mathcal{Q}$ and if $\mathcal{Q}' = FGrow \circ (Split \circ FGrow)^i(\mathcal{Q}, \mathcal{M})$ is stable, then $\mathcal{Q}'$ is the canonical partition $\approx_{\mathcal{M}}$.

3 Online Algorithm without Path Compression

In this section, we formulate the online refinement problem for BDDs without path compression. We exhibit a simple algorithm that potentially touches each node with every update operation. Next, we show how to obtain an algorithm where each internal node is touched at most $\log n$ times.

To simplify the problem, we consider a single BDD instead of the multiple shared ones and formulate algorithms that maintain canonical equivalence classes at all levels, not only the root. We can then solve the vectorized problem by

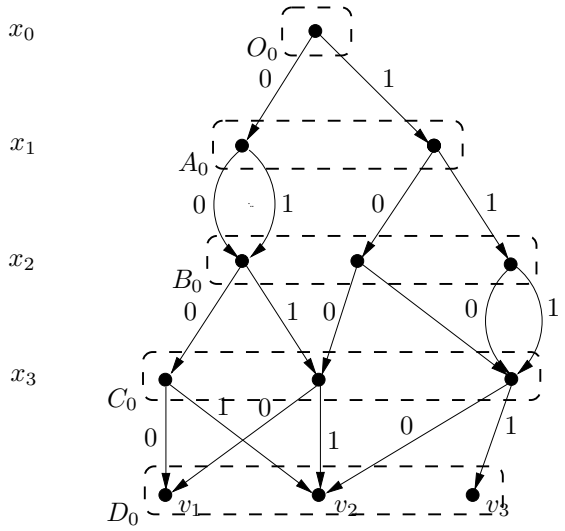

Figure 1: A BDD without path compression

inserting dummy binary variables. For example, if we consider BDDs φ_0 and φ_1 of variables $x_1, \ldots, x_n$, we may insert the dummy variable x_0 and edges from a new root to the old ones $\wedge\varphi_0$ and $\wedge\varphi_1$ so that if x_0 is 0, then φ_0 is followed, and if it is 1, then φ_1 is followed. Equivalence of the two BDDs under the current leaf discriminator then amounts to whether the old roots are in the same block.

A Simple Online Algorithm

A BDD without path compression is equivalent to an automaton that classifies all words in $\mathbb{B}^\nu$. For example, the BDD in Figure 1 classifies truth assignments $\vec{u}$ to $x_0, \ldots, x_3$ into three classes according to which leaf is reached. If the discriminator $v.d$ is the same, say 0, for all leaves v, then the corresponding canonical partition is as indicated. That is, at each level all nodes are in the same equivalence class. Now consider an update operation $\mathit{update}([v_3 \mapsto 1])$, which places the third leaf into its own class D_1. The resulting canonical partition is indicated in Figure 2.

For each level, the block of node v is determined by its discriminator $v.d$. When an $\mathit{update}(E)$ operation is received, we may assume that the current partition is the canonical one according to the value of D. We then need to further split blocks in order to reflect the perturbation E. If we assume a perfect hash function $h : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$, then internal nodes v are discriminated simply according to their behavior with respect to the equivalence classes of

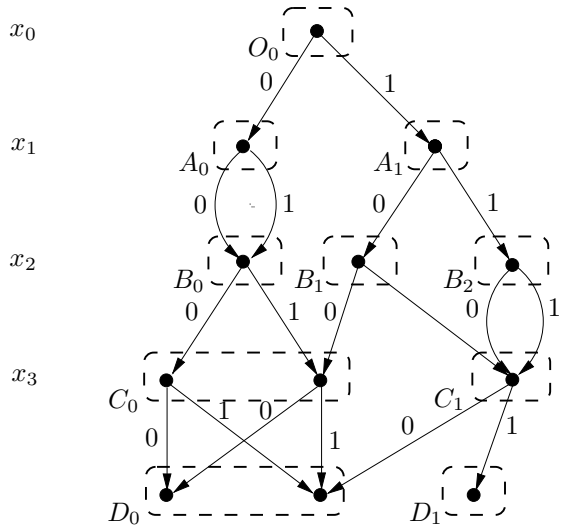

Figure 2: After splitting leaves

their successors. Thus, $h((v \cdot 0).d, (v \cdot 1).d)$ classifies nodes at level ℓ once nodes at level $\ell - 1$ been classified. Thus the algorithm

for $\ell := \nu - 1, \ldots, 0$ do
    for each v at level ℓ do
        $v.d := h((v \cdot 0).d, (v \cdot 1).d)$

calculates the new canonical partition. However, since this algorithm looks at every node in the BDD for every update, its complexity is $O(n \cdot k)$.

An $O(n \cdot \min(k, \log n) + k)$ Algorithm

We can use Hopcroft's "process the lesser half" idea to obtain an algorithm where each internal node is touched at most $\log n$ times.

To understand how this is done, we consider the example again and note that C_0 can be calculated at no cost if its discriminating value is preserved: as C_1 is created, it can simply be subtracted from the original C_0 and what remains is the new C_0. The new block C_0 has the property that all of its successors hits the largest block of leaves after the update. Hopcroft's observation here amounts to the fact that parent nodes whose successors are both in such blocks need not be explicitly considered when the split is carried out at the parent level.

Note that if we had specified E to be $[v_1 \mapsto 1, v_2 \mapsto 1]$ instead of $[v_3 \mapsto 1]$, then we would need to swap $D_1 = \{v_1, v_2\}$ and $D_0 = \{v_3\}$ so as to preserve the

$\Delta_\nu := \text{domain}(E)$;
for $\ell := \nu, \ldots, 0$ do
    $new := [v.d \mapsto \emptyset \mid v \in \Delta_\ell]$;
    $\Delta_{\ell-1} := \emptyset$;
    for all v in Δ_ℓ do
        $d_{old} := v.d$;
        $d_{new} := \begin{cases} E(v) & \text{if } \ell = \nu \\ h((v \cdot 0).d, (v \cdot 1).d) & \text{if } \ell < \nu \end{cases}$
        remove v from $L^\ell(d_{old})$;
        add v to $L^\ell(d_{new})$;
        $v.d := d_{new}$;
        $new(d_{old}) := new(d_{old}) \cup \{d_{new}\}$;
    for all d_{old} in domain(new) do
        let $d_{max} \in new(d_{old})$ such that $|L^\ell(d_{max})|$ is maximal;
        if $|L^\ell(d_{max})| > |L^\ell(d_{old})|$ then
            $switch(L^\ell(d_{max}), L^\ell(d_{old}))$;
        for all $d_{new} \in new(d_{old})$
            add parents of all $v \in L^\ell(d_{new})$ to $\Delta_{\ell-1}$

where

$switch(L(d), L(d')) =$
    for all v in $L(d)$
        $v.d := d'$;
    for all v in $L(d')$
        $v.d := d$;
    swap elements of $L(d)$ and $L(d')$

Figure 3: Online minimization of BDDs without path compression.

discriminating value in the largest block.

The data structures that we need to implement these ideas are: for each node v, a list of all parents; and for each level ℓ and each discriminating value d calculated, a doubly-linked list $L^\ell(d)$ of nodes in the block determined by d. The length $|L^\ell(d)|$ of the list is maintained in memory.

The iterative splitting and renaming procedure is then realized by for each level ℓ to register the nodes Δ_ℓ that have changed discriminator. Also for each block d_{old} that is split, the set $new(d_{old})$ of new discriminators replacing d_{old} is calculated. This information together with length information can then be used to decide for each d_{old} when swapping is needed.

The details of this are as in Figure 3.

The running time of this algorithm is the sum of the time spent for the leaves

($\ell = \nu$) and for the inner nodes ($\ell < \nu$). Each time the parents of a node v are added to $\Delta_{\ell-1}$, v is in a block at level ℓ, which is at most half the size of the block that contained v when it was last considered. Thus a parent v' at level $\ell-1$ is thrown into $\Delta_{\ell-1}$ at most $2 \cdot \log n'$ times, where n' is the number of nodes at level ℓ. So the work for inner nodes is $O(n \log n)$, but also $O(n \cdot k)$. Thus the online refinement problem for BDDs without path compression is solvable in $O(n \cdot \min(k, \log n) + k)$.

4 Online Algorithm with Path Compression

In this section, we solve the following variation on the problem considered in Section 2.

Single BDD Online Problem
Input:A BDDs $\vec{\varphi}$ with leaves $\mathcal{L}$ and constant discriminator D.
Maintained : For each node v, a discriminator value $v.d$ is maintained.
Update: A partial mapping $E : \mathcal{L} \to \mathbb{N}$ such that **range**D does not intersect the current discriminator. The current discriminator D is updated according to E. After each update operation, the discriminator values of nodes induce an equivalence relation, which is the same as $\approx_D$.
Output: A list of nodes for which $v.d$ has changed.

Since we here have an online problem with path compression, we must alternately split and grow blocks of nodes. For example, the BDD in Figure 2 becomes partitioned as shown in Figure 4 after equivalence classes have been coalesced. In general after the current leaf partition D has been perturbed by E resulting in a new partition D', we may calculate the new partition according to Theorem 3 by applying $\mathit{Split} \circ \mathit{FGrow}$ until a new fixed point is reached. (Note that the conditions for applying Theorem 3 are satisfied: since $\mathcal{Q}$ is $\approx_D$ (by assumption that the algorithm maintains the canonical partition) and D', which represents $\mathcal{M}$ of Theorem 3, refines D, it is the case that $\approx_{D'}$ refines $\mathcal{Q}$.) Since equivalence classes for BDDs with path compression may contain nodes at several levels, we use lists $L(d)$ that are not indexed by level. After each iteration, the lists $L(d)$ hold a partition refined by the canonical one. We maintain $L(d)$ such that its decision nodes are placed before its redundant nodes.

The algorithm implementing *FGrow* uses a mapping *new* as in the previous algorithm for expressing the set consisting of each discriminator d_{old} whose block B is subjected to a non-trivial decision partition. The lists $L(d)$, $d \in \mathit{new}(d_{old})$, then represent the decision blocks of B. We assume that the nodes of these decision blocks have been removed from $L(d_{old})$. We adopt Hopcroft's idea by allowing one decision block to be only implicitly represented. This block consists of the decision nodes of B that remain in $L(d_{old})$.

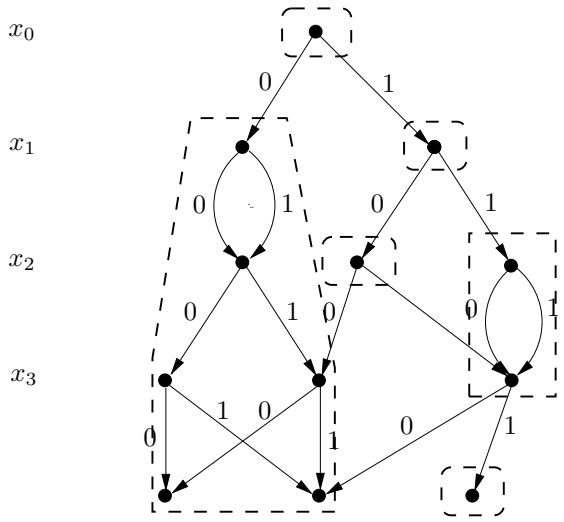

Figure 4: A BDD with path compression

Initially, we call the *FGrow* algorithm with *new* initialized according to E as calculated in the previous algorithm.

The *FGrow* algorithm grows the decision blocks by removing nodes from $L(d_{old})$. What remains of $L(d_{old})$ is the remainder of B, possibly fused with the closure of a decision block. The *FGrow* algorithm returns a list of all nodes possessing a successor whose discriminator has changed.

The *Split* algorithm simply calculates the new discriminator of the nodes in the list. As in standard BDD reduction, we hash on the discriminators of the successors and the index. The *new* mapping is calculated as before.

4.1 Main Idea

The main idea behind the *FGrow* algorithm is that when the decision blocks of a block B are to be grown, then all unfinished blocks are grown in parallel until either (a) a block becomes too big (say half the size of B) or (b) until only one block is unfinished or (c) until all blocks are finished. In case (a) and (b), the block in question is fused with the remainder. Despite this loss of information, a proper fixed point will still be reached by Theorem 3. In case (a), all remaining blocks are then finished and they will all be small since a big one already was found. In case (b), all blocks, possibly except the last one, will by the absence of the condition in case (a) be small.

In both case (a) and (b), the work involved in building the aborted block

$Split(\Delta) =$
  $new := [v.d \mapsto \emptyset \mid v \in \Delta]$;
  for all v in Δ do
    $d_{old} := v.d$;
    $d_{new} := \begin{cases} E(v) & \text{if } \ell = \nu \\ h(v.i, (v \cdot 0).d, (v \cdot 1).d) & \text{if } \ell < \nu \end{cases}$
    remove v from $L(d_{old})$;
    add v to $L(d_{new})$;
    $v.d := d_{new}$;
    $new(d_{old}) := new(d_{old}) \cup \{d_{new}\}$;
  FGrow (new);

Figure 5: Online minimization of BDDs with path compression (*Split*).

can be charged to a finished block, if such a block exists. The finished block will then be small. For this argument to be correct, it is crucial that the work done is the same for all the blocks grown in parallel.

In case (a), a finished block always exists, but in case (b), there may be no such block. This situation occurs when there is only one decision block to begin with. In this case, the work involved will be proportional to the size of the decision block and it will be charged to the time it took to build the decision block. The algorithm makes sure that the original discriminating value of the whole block is maintained despite a possible new value assigned to the decision block. In this way, only blocks that are really split may result in further splitting.

In case (c), all blocks will be small.

4.2 Detailed Description

The different stages of the algorithm are shown in Figures 5, 6, and 7. The *Split* is implemented in Figure 5. The *FGrow* operation is implemented using the following concepts. Throughout the operation, parent nodes with a successor changing discriminator are accumulated in a list Δ. For each block B represented by a discriminator d_{old} described by *new*, decision blocks must be grown according to the main idea. The blocks that are explicitly described have discriminators $new(d_{old})$. In addition, we select a new discriminator value d_{new} for the decision nodes of $L(d_{old})$ (we use an otherwise unused discriminator that cannot be the value of the hash function). Since we must process all blocks in parallel, the list $L(d_{new})$ is initially empty. We calculate the size *oldsize* of the block B by adding up the sizes of $L(d_{old})$ and all $L(d)$ for $d \in new(d_{old})$. The unfinished blocks are identified in the variable *current*, which initially consists of d_{old} and $new(d_{old})$. We use a flag *halfsize_found* to indicate whether case

```
FGrow (new) =
    Δ := ∅;
    for each d_old in domain(new)
        oldsize := |L(d_old)| + Σ_{d∈new(d_old)} |L(d)|;
        d_new := a new discriminating value;
        current := {d_new} ∪ new(d_old);
        halfsize_found := false;
        finished_dec_nodes := false;
        while (|current| ≥ 2 and not halfsize_found)
                or (|current| ≥ 1 and halfsize_found) do
            for each d in current do
                if d = d_new then
                    if not finished_dec_nodes then
                        if next node v in L(d_old) is a decision node then
                            move v to L(d_new)
                        else
                            finished_dec_nodes := true;
                if d ≠ d_new or finished_dec_nodes then
                    select_new_parent(d, v_d, w_d);
                if w_d is defined then
                    if w_d·0.d = d and w_d·1.d = d then
                        move w_d to end of L(d);
                    if L(d) ≥ oldsize/2 then
                        move L(d) nodes to L(d_old);
                        current := current\{d};
                        halfsize_found := true;
                else
                    current := current\{d};
        if not halfsize_found and |current| = 1
            let d be such that {d} = current;
            move L(d) nodes to L(d_old);
        for each d ∈ new(d_old) ∪ {d_new} and each v_d in L(d) do
            for each w_d in P(v_d)
                if w_d.d ≠ d then
                    Δ := Δ + {w_d};
    if Δ ≠ ∅ then
        Split(Δ);
```

Figure 6: Online minimization of BDDs with path compression (*FGrow*).

```
Select_new_parent(d, v_d, w_d)=
    if v_d is not defined and L(d) is not empty then
        v_d := first node in L(d);
    if w_d is not defined then
        w_d := first node in P(v_d);
    else
        if w_d is not last node in P(v_d) then
            w_d := next node in P(v_d);
        else
            if v_d is not last node in L(d)
                v_d := next node in L(d);
                w_d := first node in P(v_d);
            else
                w_d := undefined;
```

Figure 7: Online minimization of BDDs without path compression (*Select_new_parent*).

(a) above occurred and a flag *finished_dec_nodes* to indicate when all decision nodes of B have been moved to $L(d_{new})$.

For each d in *current*, we maintain the *current node* v_d of $L(d)$ whose parents are being considered and the *current parent* w_d. The list of parents of a node v is denoted $P(v)$. We assume that these lists have been pre-calculated. The parent nodes are explored according to Figure 7.

When nodes are moved from one list to another, their discriminator is changed accordingly. Also, they are placed so that decision nodes occur before redundant nodes.

To help understanding the algorithm, let us consider an example where a block B denoted by d_{old} has been split according to a decision partition that has placed all decision nodes of B into a list $L(d)$. Thus $new(d_{old})$ is $\{d\}$ and there are no decision nodes in $L(d_{old})$. Before the first iteration, *current* has been set to $\{d_{new}, d\}$. When d_{new} is selected to be grown, the algorithm discovers that there are no more decision nodes in $L(d_{old})$, and the current node v_d cannot be defined and as a result, d_{new} is removed from *current*. When d is considered, the first node in $L(d)$ is selected as v_d and a first parent v_d is selected as well. This parent may even be moved to $L(d)$ if both of its successors are in $L(d)$. However, since *current* is now a singleton, there will be no more iterations and the nodes in $L(d)$ are moved to $L(d_{old})$. Therefore, no parents are thrown into Δ at the end of *FGrow*.

To show that the algorithm terminates, it is sufficient to establish that when

a block has only one decision block, then the original discriminator is restored and no parents are placed in D. We have just analyzed one such case above. The other cases to be analyzed are those where *halfsize_found* is set and where the decision block is represented implicitly as nodes in $L(d_{old})$ (in which case $new(d_{old})$ is empty).

To solve the BDD online problem, we also need to collect as output the nodes whose discriminator change.

4.3 Complexity Analysis

Theorem 4 The Single BDD Online Problem can be solved in time $O(n \min(k, \log n) + k)$, where n is the number of nodes in the BDDs and k is the total size of all operations. Thus, if n also bounds k, then the algorithm is $O(n \log n)$.

Proof The total time spent initializing *new* before *FGrow* is called is $O(k)$. The algorithm guarantees that any decision block that is fully grown is at most half the size of the containing block. Thus every time any current node or current parent is touched, then computation time is charged to a block, which is at most half as big as the previous time. Thus the time spent on each node is $O(\log n)$. This figure excludes time that is incurred when a block has only one decision block. In the case that the decision block is created during a *Split* phase, the time can be charged to the creation of the decision block (since we have argued that no further calculations arise from such a block). In the case that the decision block is created during initialization, the time (which is proportional to the length of the description of E pertaining to the block) can be attributed to the total length of the input.

Thus the total time is $O(n \log n + k)$. We do not achieve the $n \min(k, \log n) + k$ bound as in Section 3, unless we modify the algorithm: as long as the total size k of the update operations is less than $\log n$, we use the straightforward method of reducing the whole BDD with each update at a total cost of $n \cdot k$. When k becomes greater than $\log n$, we use our online algorithm and initialize with the current leaf partition. □

As explained in Section 3, this result can be used to establish the time bound in Theorem 1 for the Multiple BDD Online Problem.

Avoiding Hashing

In the *Split* step, a linear time bucket sort technique, see [9], can replace the use of hashing. The idea is to sort all triples $h(v.i, (v \cdot 0).d, (v \cdot 1).d)$ before new discriminating values are assigned. Thus our time bounds do not depend on perfect hashing.

$E := [v \mapsto 0 \mid v \in \mathcal{L}$ and v represents a state in $F]$
$\quad \cup [v \mapsto 1 \mid v \in \mathcal{L}$ and v represents a state not in $F]$;
repeat
$\quad$ *changed_states* := *update* $(\vec{\varphi}, E)$;
$\quad E := []$;
$\quad$ for each i in *changed_states*
$\quad\quad$ let v be the leaf representing i;
$\quad\quad$ add $v \mapsto R(i)$ to E;
until *changed_states* $= \langle\rangle$;

Figure 8: Minimization of BDD-represented automaton.

5 Minimizing BDD Represented Automata

We consider languages over the alphabet $\mathbb{B}^\nu$. Thus a letter $\vec{x}$ is a vector $x_0, \cdots, x_{\nu-1}$ of ν bits. An *automaton* A over $\mathbb{B}^\nu$ with state space $[N]$, where $[N] = \{0 \ldots N-1\}$, is specified as $(\vec{\varphi}, \mathcal{L}, T, F)$, where $\vec{\varphi}$ is consists of n shared BDDs, $\mathcal{L}$ is the set of leaves of $\vec{\varphi}$, $T : \mathcal{L} \to [N]$ is the *transition mapping* and $F \subseteq [N]$ is the set of final states. State 0 is the *initial state*. There is a transition $i \xrightarrow{\vec{x}} j$ iff $T \circ \varphi_i(\vec{x}) = j$. This representation is discussed in detail in [3].

The minimization algorithm consists of first reducing $\vec{\varphi}$ with respect to initial partition $\{F, [N] \backslash F\}$ and then repeatedly applying the update operation in order to split states. The output of the update operation is conjoined with the previous partition in order to define the leaf partition of the next update operation. This process is continued until a fixed point is reached.

If we assume that n bounds both N and the number of nodes in the shared BDD representation, then the straightforward implementation (described in [3]) carries out each update operation in time $O(n)$ and there are at most n iterations. With the BDD online algorithm, however, we can do better than $O(n^2)$.

For simplicity, we assume that the shared BDD φ has exactly N leaves, one for each $i \in [N]$. Thus the minimization algorithm can be written as in Figure 8.

To analyze the running time of this algorithm, we observe that each i is included in *changed_states* at most *logn* times. Thus the total size of all parameters E is $O(n\text{log}n)$. Thus the total time is $O(n\text{log}n)$ by Theorem 1.

Theorem 5 Minimization is $O(n \log n)$ for BDD-represented automata, where n bounds the number of states and the number of BDD nodes.

Acknowledgments

Thanks to Bob Paige and Theis Rauhe for discussions on earlier versions of the ideas presented here.

References

[1] R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing surveys*, 24(3):293–318, September 1992.

[2] A. Cardon and M. Crochemore. Partitioning a graph in $O(|A| \log_2 |V|)$. *TCS*, 19:85–98, 1982.

[3] J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. Technical Report RS-95-21, BRICS, Department of Computer Science, University of Aarhus, 1995. Accepted for the TACAS Workshop, 1995; available through http://www.brics.aau.dk/~klarlund.

[4] J. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In Z. Kohavi and Paz A., editors, *Theory of machines and computations*, pages 189–196. Academic Press, 1971.

[5] D. Lee and M. Yannakakis. Online minimization of transition systems. In *Proc. STOC*, pages 264–274. ACM, 1992.

[6] H-T. Liaw and C-S. Lin. On the OBDD-representation of general Boolean functions. *IEEE Trans. on Computers*, C-41(6):661–664, 1992.

[7] R. Paige. Personal communication. 1995.

[8] R. Paige and R. Tarjan. Three efficient algorithms based on partition refinement. *SIAM Journal of Computing*, 16(6), 1987.

[9] D. Sieling and I. Wegener. Reduction of OBDDs in linear time. *IPL*, 48:139–144, 1993.

Recent Publications in the BRICS Report Series

RS-95-29 **Nils Klarlund.** ***An $n \log n$ Algorithm for Online BDD Refinement.*** **May 1995. 20 pp.**

RS-95-28 **Luca Aceto and Jan Friso Groote.** ***A Complete Equational Axiomatization for MPA with String Iteration.*** **May 1995. 39 pp.**

RS-95-27 **David Janin and Igor Walukiewicz.** ***Automata for the μ-calculus and Related Results.*** **May 1995. 11 pp. To appear in** ***Mathematical Foundations of Computer Science: 20th Int. Symposium,*** **MFCS '95 Proceedings, LNCS, 1995.**

RS-95-26 **Faith Fich and Peter Bro Miltersen.** ***Tables should be sorted (on random access machines).*** **May 1995. 11 pp. To appear in** ***Algorithms and Data Structures: 4th Workshop,*** **WADS '95 Proceedings, LNCS, 1995.**

RS-95-25 **Søren B. Lassen.** ***Basic Action Theory.*** **May 1995. 47 pp.**

RS-95-24 **Peter Ørbæk.** ***Can you Trust your Data?*** **April 1995. 15 pp. Appears in Mosses, Nielsen, and Schwartzbach, editors,** ***Theory and Practice of Software Development. 6th International Joint Conference CAAP/FASE,*** **TAPSOFT '95 Proceedings, LNCS 915, 1995, pages 575–590.**

RS-95-23 **Allan Cheng and Mogens Nielsen.** ***Open Maps (at) Work.*** **April 1995. 33 pp.**

RS-95-22 **Anna Ingólfsdóttir.** ***A Semantic Theory for Value–Passing Processes, Late Approach, Part II: A Behavioural Semantics and Full Abstractness.*** **April 1995. 33 pp.**

RS-95-21 **Jesper G. Henriksen, Ole J. L. Jensen, Michael E. Jørgensen, Nils Klarlund, Robert Paige, Theis Rauhe, and Anders B. Sandholm.** ***MONA: Monadic Second-Order Logic in Practice.*** **May 1995. 17 pp.**

RS-95-20 **Anders Kock.** ***The Constructive Lift Monad.*** **March 1995. 18 pp.**

RS-95-19 **François Laroussinie and Kim G. Larsen.** ***Compositional Model Checking of Real Time Systems.*** **March 1995. 20 pp.**