Graph driven BDDs – a new data structure for Boolean functions

Detlef Sieling*, Ingo Wegener

FB Informatik, LS II, Universität Dortmund, 44221 Dortmund, Germany

Received August 1992; revised October 1993
Communicated by M.S. Paterson

Abstract

(Ordered) binary decision diagrams (OBDDs) are used as a data structure for Boolean functions in the logical synthesis process, for verification and test pattern generation, and as part of CAD tools. For several important functions like arithmetical and logical units with quite different functions, the indirect storage access function or the hidden weighted bit function OBDDs have exponential size for any ordering of the variables. Since an ordering of the variables may be stored as a list, OBDDs may also be called list driven BDDs. Two new generalized models of graph driven BDDs are presented. The above mentioned and many other functions can be represented in small polynomial size in this model and the usual operations on OBDDs can be performed efficiently also for graph driven BDDs.

1. Introduction

Data structures for Boolean functions have to allow succinct representations of as many Boolean functions as possible and they have to admit efficient algorithms for the most important operations on this data structure, i.e. evaluation for a given input, satisfiability test and satisfiability count, minimization and reduction, synthesis, replacement of variables by constants, replacement of variables by functions and equality test.

Many data structures like circuits, formulas, branching programs (also called BDDs) and read-once branching programs (also called free BDDs) allow succinct representations of many Boolean functions but not all operations can be performed in polynomial time. For other data structures like disjunctive or conjunctive normal forms or decision trees certain simple functions require representations of exponential size.
size. Ordered binary decision diagrams (OBDDs) introduced by Bryant [7, 8] are a compromise. OBDDs are highly restricted branching programs. A branching program is a directed acyclic graph with one source. The sinks are labelled by Boolean constants. The inner nodes are labelled by Boolean variables and have two outgoing edges labelled by 0 and 1. The evaluation for an input \( a \) starts at the source. At a node labelled \( x_i \) the outgoing edge with label \( a_i \) is chosen. The label of the sink which is reached defines \( f(a) \) for the function \( f \) computed by the branching program. A branching program (or BDD) is called read-once branching program (or free BDD) if on each path there is for each variable \( x_i \) at most one node labelled \( x_i \). A free BDD is called ordered BDD if the following property is fulfilled for some ordering \( \leq \) of the variables. If a node with label \( x_j \) is a successor of a node with label \( x_i \), then the condition \( x_j \geq x_i \) and \( x_j \neq x_i \) has to be fulfilled.

The BDD package of Brace et al. [5] is a package of efficient algorithms for operations on OBDDs. Although OBDDs are a nice data structure with many advantages there are several functions which only have OBDDs of exponential size but which appear to be simple enough to be represented in small polynomial size (see Section 3 for examples). This observation has lead to several extensions of the OBDD model. A possible extension is to allow the test at the nodes to be some function rather than a variable. Aborhey [1] has investigated parity functions as tests. This approach may lead to diagrams whose depth is larger than \( n \) and the satisfiability test may be more difficult than a check whether the source is connected with the 1-sink. Akers [2] has already worked with output inverters. Such an inverter at the head of an edge has the consequence that the function computed at this position becomes negated. It is possible to compute a subfunction and its complement at a single node. Minato et al. [14] introduce input inverters to negate the next variable tested. These generalizations may only halve the size of the data structures.

Variable shifters are also presented by Minato [14]. If a computation path runs through an edge marked by \( k \in \{1, \ldots, n-1\} \), the number \( k \) is added to the indices of all variables tested later. With variable shifters it is possible to merge subgraphs with the same structure but different variables at the nodes. This approach may reduce the size of the data structure at most by a factor of \( n \).

We are interested in generalized models which allow an exponential reduction of the size for certain functions. Ashar et al. [3] allow variables to be tested more than once on a computation path. Then the data structure of minimal size is not unique, the evaluation may take more time than \( n \), the satisfiability test is NP-complete and the equality test is co-NP-complete. Hence, we have to work with heuristic algorithms. The same holds for the extended BDDs proposed by Jeong et al. [12]. They allow existential and/or universal quantifiers at the nodes. Such an approach in complexity theory has been suggested also by Meinel [13]. Our new model will not work with heuristic algorithms. We shall ensure that the satisfiability test and the equality test can be performed in linear time. Independently from our approach Gergov and Meinel [11] have...
investigated for which read-once branching programs the algorithms of Bryant [7, 8] can be used.

OBDDs may be called list driven, since the ordering of the variables may be described by a list. This requirement (or oracle) which determines the ordering in which the variables may be tested is replaced in our approach by a graph. This implies that the ordering in which the variables are tested may depend on the input. In Section 2 we define carefully this new model. In Section 3 we present examples of interesting and important functions like ALUs with quite different functions, the indirect storage access function and the hidden weighted bit function [9] for which OBDDs need exponential size while graph driven BDDs may represent these functions in small polynomial size.

In Section 4 we present a list of problems for which efficient algorithms on graph driven BDDs should exist. In Section 5 we discuss why the oracle graph should be reduced. Moreover we present algorithms for the preprocessing of the oracle. In Section 6 we prove the uniqueness (up to isomorphism) of reduced graph driven BDDs and we describe structural properties of reduced graph driven BDDs which simplify the design of the following algorithms. Section 7 is devoted to linear time reduction algorithms. Efficient algorithms for the other problems are presented in Section 8 and Section 9, since we distinguish two models of graph driven BDDs. A package of efficient algorithms for graph driven BDDs is available.

2. Graph driven binary decision diagrams

OBDDs are defined as free BDDs or read-once branching programs with a given ordering of the variables. The order in which the variables have to be tested is the same for all inputs. In our generalized models the order in which the variables are tested shall be fixed for a given input but the order may depend on the input. In order to work with free BDDs it is not possible to allow arbitrary orderings of the variables for different inputs, e.g. the variable tested first has to be the same for all inputs. Oracle graphs are the most general description of different variable orderings for different inputs under the assumption that these requirements can be fulfilled for all Boolean functions by free BDDs.

**Definition 1.** An oracle graph is a branching program with a single sink labelled “end”. On each path from the source to the sink there is for each variable $x_i$ exactly one node labelled $x_i$.

**Definition 2.** A free BDD $G$ is called $G_0$ driven for an oracle graph $G_0$ if the following property is fulfilled for all inputs $a$. Let $G(a)$ be the list of labels on the computation path for $a$ in $G$ without the label of the sink. Let $G_0(a)$ be defined similarly for the oracle graph $G_0$. If $x_i$ and $x_j$ are contained in $G(a)$, then $x_i$ and $x_j$ have to occur in the same order in $G(a)$ and $G_0(a)$.
Now it is easy to see that the conditions for oracle graphs are necessary. If some variable $x_i$ is not tested on some path in $G_0$, some Boolean functions, e.g. the parity function, are not computable by $G_0$ driven BDDs. If some variable $x_i$ is tested twice on some path in $G_0$, there is some input $a$ such that the ordering of the variables is not uniquely determined by $G_0$. Moreover, it is obvious (and will follow from results in Section 8 and 9) that each Boolean function $f$ can be represented by a $G_0$ driven BDD if $G_0$ is defined on the same variables as $f$.

The algorithms for OBDDs work efficiently only if all OBDDs are based on the same ordering of the variables which has to be fixed in advance. Hence, we require that the oracle graph is fixed in advance. We distinguish two data structures.

**Definition 3.** The LBDD data structure (loosely structured graph driven BDDs) is based on a fixed oracle graph $G_0$. Each $G_0$ driven free BDD is called a $G_0$ driven LBDD.

**Definition 4.** The WBDD data structure (well-structured graph driven BDDs) is based on a fixed oracle graph $G_0$. A $G_0$ driven free BDD $G$ is called a $G_0$ driven WBDD if there exists a representation function $\alpha : V \rightarrow V_0$ with the following properties. The nodes $v$ and $\alpha(v)$ are labelled by the same variable and for all inputs $a$ such that $v$ is contained in $G(a)$ the node $\alpha(v)$ is contained in $G_0(a)$. The node $\alpha(v)$ in $G_0$ is called the representative of $v$.

Why do we distinguish these two data structures? The reason is a time-space trade-off between the WBDD model and the LBDD model. If we reach the node $v$ of a WBDD $G$ for some input $a$, then it follows that the node $\alpha(v)$ of the oracle graph $G_0$ is also reached for this input. In the LBDD model it is possible that the node $v$ with label $x_i$ of an LBDD $G'$ is reached for the inputs $a$ and $b$ while the nodes with label $x_i$ in $G_0(a)$ and $G_0(b)$ are different. This special property of WBDDs leads to the design of simpler and faster algorithms for WBDDs than for LBDDs.

Since WBDDs are restricted LBDDs, minimal size LBDDs for some functions $f$ are no larger than minimal size WBDDs for $f$. In Section 9 we prove that the maximal quotient of the minimal size WBDD and the minimal size LBDD is bounded by $|G_0|$, the size of the oracle graph. There are examples where this quotient is almost $|G_0|$. Let us consider the Boolean function $f(x_1, \ldots, x_n) := x_i$. The minimal size LBDD consists of one inner node and two sinks for an arbitrary oracle graph $G_0$. The minimal size WBDD contains at least as many nodes labelled by $x_i$ as $G_0$.

We shall work with minimal size WBDDs and LBDDs. For some particular operation our WBDD algorithm may be faster than our LBDD algorithm. The running time of algorithms is always compared for the same input size. One should note that the LBDD algorithm works for a particular Boolean function on the minimal size LBDD $G_L$ and the WBDD algorithm works on the minimal size WBDD $G_W$ where $|G_L|$ may be much smaller than $|G_W|$. Moreover, applications of BDD data structures fail much more often because of lack of space than because of lack of time.
We introduce some special classes of oracles. Some algorithms will work more efficiently for these oracles.

**Definition 5.** (i) An oracle graph \( G_0 \) is called a tree oracle if \( G_0 \) becomes a tree by eliminating the sink and replacing multiedges between nodes by simple edges.

(ii) An oracle graph \( G_0 \) is called a list oracle if \( G_0 \) becomes a list by eliminating the sink and replacing multiedges between nodes by simple edges.

(iii) A variable \( x_i \) is called a branching variable with respect to the oracle graph \( G_0 \) if some node \( v \) in \( G_0 \) labelled by \( x_i \) has two different successors. An oracle graph \( G_0 \) is called \( x_i \)-oblivious if \( x_i \) is not a branching variable for \( G_0 \).

OBDDs are graph driven BDDs with list oracles. A list oracle is \( x_i \)-oblivious for each variable \( x_i \).

Before we discuss the advantages of the new model we have to explain in which situations we like to apply it. If a variable ordering leads to small size OBDDs there is no need for a generalized model. But, if OBDDs have large or even too large size, our model is in many situations an alternative leading to more succinct representations and allowing almost as efficient operations as OBDDs. Since \( G_0 \) driven BDDs tend to have a size not smaller than \( G_0 \), in practice we should work with oracle graphs of moderate size, e.g. quasilinear size as \( O(n \log n) \) or at most quadratic size \( O(n^2) \). The theory and the algorithms work for all oracle graphs \( G_0 \). First we show that we obtain the expressive power of read-once branching programs.

**Theorem 1.** Let \( G \) be a read-once branching program, i.e. there is no path where some variable is tested twice. In time \( O(n|G|) \) we can compute an oracle graph \( G_0 \) such that \( G \) is a \( G_0 \) driven WBDD.

**Proof.** First we merge all sinks of \( G \) to a sink with label “end”. Then we compute for each edge \((v, w)\) the set \( V(v, w) \) of variables tested on some path from the source to this edge. We use a depth first search approach to compute a numbering of the nodes of \( G \), which constitutes a topological order for the nodes and therefore also for the edges. The sets \( V(v, w) \) are computed in this order. The set \( V(v, w) \) is described as a bit array of length \( n \). For the edges leaving the source the information is easily available, since only the source has been tested. If the information for the edge \((v, w)\) is available, it is added to the information of all edges leaving \( w \). If this has been done for all edges entering \( w \) and if the information that the variable which is the label of \( w \) is also reached is added, the information for the edges leaving \( w \) is correctly computed. The time bound holds, since the information for each edge \((v, w)\) has to be copied only twice for the two edges leaving \( w \).

In the second phase we add nodes. For this purpose we run again through the list of nodes sorted with respect to some topological order. For node \( w \) we consider all incoming edges \((v_1, w), \ldots, (v_r, w)\) and note that the work has already been done for \( v_1, \ldots, v_r \), i.e. we have added nodes in such a way that on each path from the source to \( v_i \) a node with label \( x_j \) is reached, if in \( G \) a node with label \( x_j \) is reached on some path
from the source to \(v_i\). We now compute \(V(w)\) as the union of \(V(v_1, w), \ldots, V(v_r, w)\) and set \(D((v_i, w)) := V(w) - V(v_i, w)\). The edge \((v_i, w)\) is replaced by a chain of nodes \((v_i, z_1, \ldots, z_l, w)\) where \(l := |D((v_i, w))|\). The nodes \(z_1, \ldots, z_l\) are labelled by the variables in \(D((v_i, w))\) and the edge \((v_i, w)\) is replaced by an edge \((v_i, z_1)\). Both edges leaving \(z_k, 1 \leq k \leq l\), lead to \(z_{k+1} := w\). For the sink \(s\) we have to define \(V(s) := \{x_1, \ldots, x_n\}\) in order to include also variables not tested at all in \(G\).

The procedure ensures that the resulting graph \(G_0\) is an oracle graph such that \(G\) is \(G_0\) driven. Moreover, \(G\) is a WBDD, since \(\alpha(v) := v\) (remember that \(V \subseteq V_0\)) is a suitable representation function. \(\square\)

Theorem 1 contains an exact statement about the expressive power of the new model. The algorithm will usually not be used. In a specific environment we shall use heuristic ideas and our intuition to construct an appropriate oracle graph as the ordering of variables for the OBDD model is usually constructed by heuristic ideas and intuition. There is no efficient algorithm known for the computation of an optimal oracle as there is no efficient algorithm known for the computation of an optimal ordering of the variables.

The main objection against the new model may be that BDDs for very simple functions may become larger than in the old model. This is indeed the case. In the \(W\)-model we need for the simple function \(x_i\) a BDD whose size is at least the number of nodes labelled \(x_i\) in \(G_0\). In the \(L\)-model a single test is sufficient. But consider the function \(x_i \land x_j\). If in \(G_0\) there are many paths where \(x_i\) is tested before \(x_j\) and many paths where \(x_j\) is tested before \(x_i\), we need many tests in order to decide which variable has to be tested first. It is easy to describe graphs \(G_0\) of any (meaningful) size such that the minimal \(G_0\) driven LBDD for \(x_i \land x_j\) has size \(\Theta(|G_0|)\). But we have already explained that we would like to use the new model only in situations where we have to consider functions which do not have small size OBDDs for each ordering of the variables. In this context we may save a lot for difficult functions and have to pay a little for very simple functions. Again we stress that \(G_0\) should be of moderate size.

The reader may believe that the example \(x_i \land x_j\) proves the impossibility of efficient algorithms for the LBDD model. Minimal \(G_0\) driven LBDDs for \(x_i\) and \(x_j\) have size 1 while the minimal \(G_0\) driven LBDD for \(x_i \land x_j\) may be of size \(\Theta(|G_0|)\). But we measure the running time of algorithms with respect to the size of the given LBDDs \(G_1\) and \(G_2\), the oracle \(G_0\) and the number of variables \(n\).

3. Examples of efficient graph driven BDDs

ALUs are typical examples where we like to work with BDDs. The inputs of an ALU can be usually partitioned into two disjoint sets, a set of control variables and a set of data variables. A rule of thumb says that we should test control variables first. But this rule does not help, if for different values of the control variables we need different orderings of the data variables to obtain small BDDs. In a typical situation
we have a small number of \( k \) control variables and a much larger number of \( n \) data variables. The oracle may consist of a complete binary tree for the test of the control variables. For each of the \( 2^k \) assignments of the control variables we may choose a different but fixed ordering of the data variables. The oracle \( G_0 \) consists of a complete binary tree of depth \( k \) where the leaves are replaced by lists of length \( n \) (see Fig. 1).

The size of the oracle graph is \( 2^k - 1 + 2^k n \). If \( k = \log n \), the size \( n^2 + n - 1 \) is quadratic with respect to the input size \( n + \log n \). There are many examples where such an oracle admits small tree driven WBDDs while all list driven BDDs have exponential size.

We present an ALU of only theoretical interest with only one control variable \( y \) and \( n = m^2 \) data variables \( x_{ij}, 1 \leq i, j \leq m \), which are arranged as a quadratic matrix. Let \( f(y, X) = 1 \) for \( X = (x_{ij}) \) iff \( y = 0 \) and \( X \) contains a row consisting of ones only or \( y = 1 \) and \( X \) contains a column of ones only. It follows by well-known lower bound techniques that each list driven BDD for \( f \) has exponential size but the oracle tree of Fig. 1 with \( k = 1 \) where the control variable \( y \) is tested and a row oriented list ordering for \( y = 0 \) and a column oriented list ordering for \( y = 1 \) admits a tree driven WBDD of linear size. The construction of this WBDD is obvious.

The next example is the indirect storage access function \( ISA \) on \( 2^k + k \) variables \( x_0, \ldots, x_{n-1} \), where \( n = 2^k \), and \( y_0, \ldots, y_{k-1} \). The \( x \)-variables are partitioned to \( n/k \) (without loss of generality a natural number) groups numbered \( 0, \ldots, n/k - 1 \) of size \( k \) each. The \( y \)-vector \( (y_{k-1}, \ldots, y_0) \) is interpreted as a binary number \( |y| \in \{0, \ldots, 2^k - 1\} \). If this number is at least \( n/k \), the output of \( ISA \) is 0. Otherwise we have to look at the \( |y| \)-th group of \( x \)-variables. This group is again interpreted as a number \( r \in \{0, \ldots, n-1\} \). Then \( ISA(x, y) := x_r \). Breitbart et al. [6] have shown that each list driven BDD for \( ISA \) has exponential size while we again obtain a natural oracle tree \( G_0 \) which admits a \( G_0 \) driven WBDD for \( ISA \) of quadratic size. We start with a complete binary tree for the \( y \)-variables which obviously are control variables. At the leaves of this tree, where \( |y| \geq n/k \), we choose appropriate suboracles (lists) for the \( x \)-variables. The choice of these suboracles is only important for other functions which should be represented by \( G_0 \) driven BDDs. For leaves where \( |y| < n/k \) we attach a list of the \( x \)-variables of the appropriate group followed again by some list of the
other variables. A $G_0$ driven WBDD for ISA starts with a complete binary tree for the $y$-variables. For $|y| \geq n/k$ the $0$ sink is reached. For $|y| < n/k$ the appropriate $x$-variables are tested in a complete binary tree. Finally, that $x$-variable which determines the output is tested if this has not been done before. Altogether the BDD consists of $n/k + 1$ binary trees whose depth is bounded by $k + 1$. Hence, the size of this $G_0$ driven WBDD is $O(n^2/\log n)$.

Up to now we have seen only examples where we could choose trees as appropriate oracles. The function ISA was already an example where several variables, namely the $x$-variables, are control variables and data variables simultaneously. The same holds for the hidden weighted bit function $HWB$ due to Bryant [9] who showed that each list driven BDD for $HWB$ has exponential size. The function $HWB$ works on $n$ variables and is defined by

$$HWB(a_1, \ldots, a_n) := a_1 + \cdots + a_n,$$

where $a_0 := 0$.

The key idea for the construction of the oracle graph $G_0$ is the following. If $i$ ones have been found, the value of $x_1, \ldots, x_{i-1}$ need not to be remembered, and if $j$ zeros have been found, the value of $x_{n-j+1}, \ldots, x_n$ need not to be remembered. In this situation it is sufficient to remember the number of ones found and the values of the variables $x_i, \ldots, x_{n-j}$ which have been tested. In order to obtain a small graph driven BDD we try to minimize the number of variables whose values have to be remembered.

The oracle $G_0$ is defined in the following way (see Fig. 2). The source is labelled by $x_1$. For each node $v$ on the levels $0, \ldots, n - 2$ the following holds. The left successor, i.e. the successor reached via the 0-edge, is labelled by $x_i$, where

$$i := \max \{ j \mid x_j \text{ is not tested on any path ending at } v \}$$

and the right successor, i.e. the successor reached via the 1-edge, is labelled by $x_k$, where

$$k := \min \{ j \mid x_j \text{ is not tested on any path ending at } v \}.$$

On each level we merge all nodes with the same label and the same set of variables tested before. It follows that on each level there are at most two nodes labelled $x_i$. For one of them it holds that it is the left successor of all its predecessors and the other is the right successor of all its predecessors. Hence, the size of the oracle graph is less than $2n^2$ (it is exactly $n^2 - n - 1$ for $n \geq 3$). If a node $v$ on level $l$ is labelled by $x_i$ and is the left successor of all its predecessors, on all paths to $v$ the variables $x_{i+1}, \ldots, x_n$ and $x_1, \ldots, x_{i-1}$ have been tested. If it is the right successor of all its predecessors, the variables $x_1, \ldots, x_{i-1}$ and $x_{n-i}, \ldots, x_n$ have been tested.

Now we design a $G_0$ driven WBDD for $HWB$. If we test $x_i$ on level $l$, we have to decide which information we have to store about the already tested variables. Let us consider in Fig. 2 the node labelled $x_8$ on level 5. We have tested before $x_1, x_2, x_3, x_9, x_{10}$. The number of ones we have tested is by construction 2 and $x_1 + \cdots + x_{10} \in \{2, \ldots, 7\}$. We need four nodes for this oracle node to store the value of
the pair \((x_2, x_3)\). For the neighboring node labelled \(x_4\) the same set of variables has been tested but the number of ones is 3. Therefore, \(x_1 + \cdots + x_{10} \in \{3, \ldots, 8\}\). We need only two nodes for this oracle node to store the value of \(x_3\). Hence, two neighboring nodes in \(G_0\) can be replaced in a \(G_0\) driven WBDD by six nodes. The resulting WBDD will consist of less than \(3n^2\) nodes.

**Theorem 2.** There is an oracle graph \(G_0\) with \(n^2 - n - 1\) nodes (if \(n \geq 3\)) such that the hidden weighted bit function \(HWB\) can be computed by a \(G_0\) driven WBDD with less than \(3n^2\) nodes.

**Proof.** We only have to generalize our considerations above. Let us consider a node at level \(l\).

**Case 1:** The oracle node is labelled \(x_i\) and we have tested before \(x_1, \ldots, x_{i-1}\) and \(x_{n-l+i}, \ldots, x_n\). Then the last test was a 1-test and the number of already tested
1-variables equals $i - 1$, because the $j$th 1-test leads to a test of $x_{j+1}$. The number of ones is contained in \{i - 1, \ldots, n - l + i - 1\} and we need to store only the value of $x_{i-1}$.

**Case 2:** The oracle node is labelled by $x_{n-l+i-1}$ and we have tested before $x_1, \ldots, x_{i-1}$ and $x_{n-l+i}, \ldots, x_n$. Then the last test was a 0-test and the number of already tested 1-variables equals $i - 2$. The number of ones is contained in \{i - 2, \ldots, n - l + i - 2\} and we have to store the value of $(x_{i-2}, x_{i-1})$.

The oracle graph $G_0$ of Fig. 2 can be replaced only by trees of exponential size. We state here without proof that tree driven BDDs for $HWB$ have exponential size for each tree oracle. Hence, the restriction to tree oracles is really a restriction.

Furthermore, we have proved that the clever lower bound technique due to Bryant [9] works only for list driven BDDs and not for graph driven BDDs or for read-once branching programs.

It remains an open problem whether multiplication can be computed by a graph driven BDD of polynomial size.

Our examples have shown that functions with exponential OBDD size for all variable orderings may have small polynomial size graph driven BDDs for suitable oracle graphs. In applications the functions are given by circuits and all functions computed at the gates of the circuits need to have small graph driven BDDs for the same oracle graph $G_0$. We investigate quite typical circuits for our examples. The special ALU and the indirect storage access function have small polynomial size minimal polynomials, i.e. PLA realizations. In this case the use of minimal PLA realizations is quite natural. The $HWB$ function has minimal polynomials of exponential size. Therefore, we investigate a linear size circuit. First the input bits are summed up. Then the binary representation of the sum is translated into a unary one, i.e. $s_i = 1$ iff $x_1 + \cdots + x_n = i$. Finally, $HWB(x)$ is the disjunction of all $x_i \land s_i$. Now it is not hard to verify that these circuits for the special ALU, ISA and $HWB$ have the property that all functions computed at the gates of the circuit have small polynomial size $G_0$ driven BDDs for the oracle graphs $G_0$ considered before. This implies that the new model is a nice and powerful tool if the usual operations on BDDs can be performed efficiently also on graph driven BDDs.

4. A list of problems

After having defined the data structure of graph driven WBDDs and LBDDs and having seen that they can represent many functions and also important functions in a much more succinct way we like to list the problems and operations which should be solvable by efficient algorithms in order to replace OBDDs by the new models. We may need different algorithms for WBDDs and LBDDs.

1. Evaluation problem. Given a data structure $G$ for the Boolean function $f$ and an input $a$. Compute $f(a)$. 

(2) Satisfiability test. Given a data structure G for some Boolean function f. Decide whether \( f(a) = 1 \) for some input \( a \).

(3) Satisfiability count. Given a data structure G for some Boolean function f. Compute \( |f^{-1}(1)| \).

(4) Satisfiability all. Given a data structure G for some Boolean function f. Compute a list of all \( a \in f^{-1}(1) \).

(5) Elimination of nonreachable nodes. Given a data structure G for some Boolean function where G is a graph with a source where the computation starts. Eliminate all nodes not reachable from the source.

(6) Reduction (or minimization). Given \( G_0 \) and a \( G_0 \) driven BDD G. Compute the unique reduced \( G_0 \) driven BDD \( G' \) equivalent to G. (Remark: We still have to prove that \( G' \) is unique up to isomorphism.)

(7) Synthesis problem. Given \( G_0 \) and \( G_0 \) driven BDDs \( G_1 \) and \( G_2 \) for the functions \( f_1 \) and \( f_2 \) and a binary Boolean operation \( \circ \). Compute a \( G_0 \) driven BDD G for \( f' := f_1 \circ f_2 \).

(8) Redundancy test. Given \( G_0 \), an index \( i \in \{1, \ldots, n\} \) and a \( G_0 \) driven BDD G. Decide whether the function computed by G depends essentially on \( x_i \).

(9) Replacement by constants. Given \( G_0 \), an index \( i \in \{1, \ldots, n\} \), a constant \( c \in \{0, 1\} \) and a \( G_0 \) driven BDD G computing f. Compute a \( G_0 \) driven BDD G' for the function \( f_{x_i} = c \).

(10) Replacement by functions. Given \( G_0 \), an index \( i \in \{1, \ldots, n\} \) and \( G_0 \) driven BDDs \( G_f \) and \( G_g \) for f and g. Compute a \( G_0 \) driven BDD G for \( f|_{x_i} = g \).

(11) Equality test. Given \( G_0 \) and \( G_0 \) driven BDDs \( G_f \) and \( G_g \) for f and g. Test whether \( f \equiv g \).

(12) Comparison test. Given \( G_0 \) and \( G_0 \) driven BDDs \( G_f \) and \( G_g \) for f and g. Test whether \( f \leq g \).

(13) Existential quantification. Given \( G_0 \), an index \( i \in \{1, \ldots, n\} \) and a \( G_0 \) driven BDD G computing f. Compute a \( G_0 \) driven BDD G' for \( \exists x_i f := f|_{x_i = 0} \lor f|_{x_i = 1} \).

(14) Universal quantification. Given \( G_0 \), an index \( i \in \{1, \ldots, n\} \) and a \( G_0 \) driven BDD G computing f. Compute a \( G_0 \) driven BDD G' for \( \forall x_i f := f|_{x_i = 0} \land f|_{x_i = 1} \).

(15) (Optional) Consistency test. Given \( G_0 \) and a BDD G. Test whether G is a \( G_0 \) driven BDD.

Since we should construct for an oracle \( G_0 \) only \( G_0 \) driven BDDs, the consistency test will not be used in standard applications.

The first five problems are independent of \( G_0 \). The known algorithms for these problems and list driven BDDs work for all read-once branching programs and therefore also in our situation without any loss in efficiency. The time bounds are \( O(n) \) (or even \( O(d) \) for the depth \( d \) of G) for the evaluation problem, \( O(|G|) \) for the satisfiability test, satisfiability count (if the node \( v \) is reached for \( k \) inputs, then exactly \( k/2 \) inputs are leaving \( v \) via each of the two outgoing edges) and also the elimination of nonreachable nodes. The satisfiability all problem can be solved in time \( O(|G| + n |f^{-1}(1)|) \) and this is also optimal with respect to input and output size. For these algorithms we refer to [7, 8].
The problems comparison test, existential quantification and universal quantification are contained in our list of problems, since they are used in applications. These problems are not really new problems. The comparison test is the nonsatisfiability test for $\bar{g} \land f$. A graph driven BDD for $\bar{g} \land f$ can be computed by the synthesis algorithm. The quantification problems can be solved efficiently if the problems replacement by constants and synthesis can be solved efficiently. No efficient algorithm for the quantification problems is known that does not use an algorithm for the problem replacement by constants as a subroutine.

5. Reduction of the oracle graph and preprocessing steps

For list driven BDDs it is known that the minimal or reduced BDD for a given function $f$ and a given variable ordering is unique up to isomorphism. Moreover, the reduced OBDD can be computed efficiently from an OBDD $G$ for $f$. The algorithm of Bryant [7, 8] with running time $O(|G| \log |G|)$ has been improved recently by Sieling and Wegener [15] to an algorithm with optimal running time $O(|G|)$ and storage space $O(|G|)$. These reduction algorithms are based on two reduction rules for OBDDs.

(R1) If the two edges leaving some node $v$ reach the same node $w$, the node $v$ can be eliminated (deletion rule).

(R2) Let $v_0$ and $v_1$ be the successors of $v$ via the 0-edge and the 1-edge resp. and let $w_0$ and $w_1$ be the corresponding successors of $w$. If $v$ and $w$ are labelled by the same variable, $v_0 = w_0$ and $v_1 = w_1$, we can merge $v$ and $w$ (merging rule).

It has been proved that by applying these rules bottom-up in $G$ one obtains the minimal OBDD for $G$.

One may ask whether one can and/or should reduce also the oracle graph (remember that oracle graphs have a unique sink). First it is not allowed to apply the deletion rule, because any application would lead to a graph not fulfilling Definition 1 (see Section 2). But the merging rule is applicable. What is changed after an application of the merging rule to an oracle graph $G_0$? In any case the new oracle graph $G_0$ is smaller by one node.

**Theorem 3.** Let $G_0$ be an oracle graph and $G_0'$ a graph which is obtained from $G_0$ by some applications of the merging rule. Let $f$ be a Boolean function.

(i) The set of $G_0$ driven LBDDs for $f$ is equal to the set of $G_0'$ driven LBDDs for $f$.

(ii) The set of $G_0$ driven WBDDs for $f$ is contained in the set of $G_0'$ driven WBDDs for $f$. The minimal size of a $G_0$ driven WBDD for $f$ may be exponentially smaller with respect to $n$ but at most by a factor of $O(|G_0|)$ smaller than the size of a minimal $G_0$ driven WBDD.

**Proof.** For each input $a$ the ordering of the variables prescribed by $G_0$ is the same as by $G_0'$. This implies the claim for LBDDs. The requirements of well-structuredness are reduced for $G_0'$ compared with $G_0$. If $v$ and $w$ are merged, it is not necessary to
distinguish tests representing \( v \) and \( w \). This implies the first part of the second claim. In order to simulate a \( G'_0 \) driven WBDD \( G' \) by a \( G_0 \) driven WBDD \( G \) it is sufficient to copy each node \( v' \) where \( \alpha'(v') = w' \) \( t \) times, if \( t \) is the number of nodes of \( G_0 \) which have been merged to produce the node \( w' \) of \( G'_0 \). Moreover, we may have to include tests to reach the right nodes. The number of included tests cannot be larger than \( |G_0| \) for each \( v' \). Finally, let \( G_0 \) be a complete binary tree where the variables are on all paths in the same order. The reduced graph \( G'_0 \) is a linear list. The minimal \( G_0 \) driven WBDD for the parity function has size \( 2^n - 1 \) while the minimal \( G'_0 \) driven WBDD has size \( 2n - 1 \).

We remark that the oracle graph of Fig. 2 is not reduced. The nodes on the last level with the same label can be merged. Because the reduction of oracle graphs has only advantages it should always be performed.

**Theorem 4.** An oracle graph \( G_0 \) can be reduced in linear time \( O(|G_0|) \) with storage space \( O(|G_0|) \). The reduced graph is unique up to isomorphism.

**Proof.** It follows by the definition of oracle graphs that all paths from the source to some node \( w \) have the same length \( l \). This length \( l \) is called the level of \( w \). A node at level \( l \) can be merged only with nodes of the same level. The reduction algorithm works bottom-up level by level. At each level the two-phase bucket sort technique of Sieling and Wegener [15] can be applied. The reduced oracle graph is unique up to isomorphism, since all possible mergings are performed and since a merging on level \( l \) only increases the chance of mergings on level \( l - 1 \).

For a list oracle we may renumber the variables (more formally we compute a dictionary to translate a new number into an old one and vice versa) and then we can assume without loss of generality that the numbering is the trivial one. This ensures that we can decide in time \( O(1) \) whether a node with label \( x_i \) may follow a node with label \( x_j \). The computation of similar information seems to be harder for general oracle graphs \( G_0 \). In order to test whether an edge \((v, w)\) in \( G \) is compatible with an oracle graph \( G_0 \) it is necessary to know with which oracle nodes \( v \) and \( w \) are related. Hence, it is useful to compute for the oracle graph \( G_0 \) a data structure supporting such tests and computations.

**Lemma 1.** Let \( v \) be a node in an oracle graph \( G_0 \), \( c \in \{0, 1\} \) and \( i \in \{1, \ldots, n\} \). Let \( V(v, c, i) \) be the set of nodes labelled \( x_i \) reachable from \( v \) via the outgoing edge labelled \( c \). If \( v' \) is a node in a \( G_0 \) driven WBDD \( G \) such that \( \alpha(v') = v \), the \( c \)-successor of \( v' \) can be labelled by \( x_i \) only if \( |V(v, c, i)| = 1 \).

**Proof.** If \( |V(v, c, i)| = 0 \), the variable \( x_i \) has to be tested earlier. If \( |V(v, c, i)| \geq 2 \), there exists for each \( u \in V(v, c, i) \) an input \( a(u) \) with the following property. Starting with input \( a(u) \) at the source the node \( v \) is reached, \( v \) is left via the \( c \)-edge and later the node
u is reached. If the c-successor w of v is labelled by \( x_i \), we reach for all \( a(u), u \in V(v, c, i) \), the same node w in G. Hence, there is no unique representation of \( w \) in contradiction to the definition of WBDDs. □

The data structure we shall compute for \( G_0 \) is an array of size \( (2|G_0| + 1)n \). It contains entries \( \text{Succ}(v, c, i) \) for all nodes \( v \) of \( G_0 \), \( c \in \{0, 1\} \) and \( i \in \{1, \ldots, n\} \) and \( \text{Succ}(\text{start}, i) \) for all \( i \in \{1, \ldots, n\} \). The entry \( \text{Succ}(v, c, i) \) is the unique node in \( V(v, c, i) \) if \( |V(v, c, i)| = 1 \), \( \text{err} 0 \), if \( |V(v, c, i)| = 0 \) and \( \text{err} 2 \), if \( |V(v, c, i)| \geq 2 \). The entry \( \text{Succ}(\text{start}, i) \) has the same meaning for an imaginary node whose successor is the source of \( G_0 \).

**Theorem 5.** The successor array \( \text{Succ} \) for an oracle graph \( G_0 \) can be computed in time \( O(n|G_0|) \).

**Proof.** We work bottom-up in \( G_0 \). For nodes on the last level all entries are \( \text{err} 0 \). Let \( w \) be the c-successor of \( v \) and let \( \text{Succ}(w, c', i) \) be computed correctly.

We set \( \text{Succ}(v, c, i) := \text{err} 0 \) if \( \text{Succ}(w, 0, i) = \text{Succ}(w, 1, i) = \text{err} 0 \) and the label of \( w \) is different from \( x_i \). We set \( \text{Succ}(v, c, i) := w \) if \( w \) is labelled by \( x_i \). We set \( \text{Succ}(v, c, i) := \text{Succ}(w, 0, i) \) if \( \text{Succ}(w, 0, i) = \text{Succ}(w, 1, i) \) and this entry is a node (because of the definition of oracles it is impossible that \( \text{Succ}(w, 0, i) \) is a node and \( \text{Succ}(w, 1, i) = \text{err} 0 \) or vice versa). Otherwise we set \( \text{Succ}(v, c, i) := \text{err} 2 \). A similar procedure works for \( \text{Succ}(\text{start}, i) \). Hence, each entry in the successor array can be computed in time \( O(1) \) □

In order to simplify the following considerations we have defined \( V(v, c, i) \) and \( \text{Succ}(v, c, i) \) for all edges (where \( (v, c) \) stands for the \( c \)-edge leaving \( v \)) and all \( i \in \{1, \ldots, n\} \). But \( V(v, c, i) \) and \( \text{Succ}(v, c, i) \) depend only on the node \( w \) which is reached via the \( c \)-edge leaving \( v \). Hence, it would be sufficient to consider \( V^*(w, i) \) and \( \text{Succ}^*(w, i) \). Since the number of nodes is half the number of edges, we could save a factor 2 for the storage space.

The data structure has nonlinear size. But we should remember that \( G_0 \) is usually much smaller than the \( G_0 \) driven WBDDs \( G \) for nontrivial functions, usually \( n|G_0| \leq |G| \). We also may save the computation of the successor array if the information \( \text{Succ}(v, c, i) \) can be computed directly from \( G_0 \) in time \( O(1) \). This holds as we have seen for list oracles and for many more oracles including all oracles discussed in Section 3.

**Theorem 6.** For an oracle graph \( G_0 \) with given Succ array and a \( G_0 \) driven WBDD G the representation function \( \alpha \) can be computed in time \( O(|G|) \). Moreover, the consistency test for WBDDs \( G \) can be solved in time \( O(|G|) \).

**Proof.** If the source \( v \) of \( G \) is labelled by \( x_i \), we look at \( \text{Succ}(\text{start}, i) \). If we find during our procedure an entry \( \text{err} 0 \) or \( \text{err} 2 \), we have proved by Lemma 1 that \( G \) is not \( G_0 \) driven. If \( \text{Succ}(\text{start}, i) \notin \{\text{err} 0, \text{err} 2\} \), we set \( \alpha(v) := \text{Succ}(\text{start}, i) \). We run through
G by a depth first search approach. If we reach w for the first time via the c-edge
leaving v, then \( v' = \alpha(v) \) has been computed (later we may prove that v also represents
another node but then we find an error). We test \( \text{Succ}(v', c, j) \) if \( x_j \) is the label of w. If
\( \text{Succ}(v', c, j) \notin \{ \text{err 0}, \text{err 2} \} \), we define \( \alpha(w) := \text{Succ}(v', c, j) \). If we reach w later again, say
via the \( c' \)-edge leaving u and \( u' = \alpha(u) \), we also have proved that G is not \( G_0 \) driven if
\( \alpha(w) \neq \text{Succ}(u', c', j) \). □

If we like, we can subsequently compute in time \( O(|G_0| + |G|) \) a list \( W(v) \) for each
node in \( G_0 \), where \( W(v) \) contains all nodes w in G where \( \alpha(w) = v \).

Let us now discuss similar data structures for oracle graphs \( G_0 \) with respect to the
LBDD model. We remember that in this less structured model a node \( v \) in a \( G_0 \) driven
LBDD \( G \) may represent several nodes of \( G_0 \). First we compute for the nodes in \( G_0 \)
successor lists \( \text{SuccL}(v, c, i) \) and \( \text{SuccL}(\text{start}, i) \) containing all nodes labelled by \( x_i \) in
\( G_0 \) reachable via the \( c \)-edge leaving v.

**Theorem 7.** The successor lists \( \text{SuccL} \) for an oracle graph \( G_0 \) can be computed in linear
time with respect to the output size which is bounded by \( O(|G_0|^2) \).

**Proof.** We have to compute \( (2|G_0| + 1)n \) lists. The lists \( \text{SuccL}(v, c, i), 1 \leq i \leq n \), or
\( \text{SuccL}(\text{start}, i), 1 \leq i \leq n \), are disjoint and contain together at most \( |G_0| \) nodes. Hence,
the total list length is bounded by \( (2|G_0| + 1)|G_0| \).

We compute the lists bottom-up and ensure that the nodes in each list are sorted
with respect to some topological order on the nodes. Let \( w \) be the \( c \)-successor of \( v \). The
list \( \text{SuccL}(v, c, i) \) consists of \( w \) if \( w \) is labelled by \( x_i \), and otherwise is the union of the
lists \( \text{SuccL}(w, 0, i) \) and \( \text{SuccL}(w, 1, i) \). Because the lists are sorted, \( \text{SuccL}(v, c, i) \) can be
computed in time \( O(|\text{SuccL}(v, c, i)|) \). □

For list oracles the size of the data structure is \( \Theta(|G_0|^2) \) while for more complicated
oracles the data structure may be much shorter. This happens in particular for tree
oracles. In any case for oracles of moderate size the size of the data structure is
acceptable. Furthermore, for oracles like those discussed in Section 3 it is not
necessary to compute the lists, since the information whether \( w \) is a successor of \( v \) in
\( G_0 \) can be computed directly in time \( O(1) \).

Instead of the representation function \( \alpha \) for WBDDs we compute for each node \( v \) in
a \( G_0 \) driven LBDD \( G \) a list \( L(v) \) of all nodes \( w \) in \( G_0 \) such that \( w \) is labelled by the same
variable as \( v \) and there exists some input for which we reach \( v \) in \( G \) and \( w \) in \( G_0 \).

**Theorem 8.** For an oracle graph \( G_0 \) with given \( \text{SuccL} \) lists and a \( G_0 \) driven LBDD \( G \) the
lists of representatives \( L(v) \) for \( v \) in \( G \) can be computed in time \( O(|G|) \) where \( l \) is the total
length of all lists. Let \( |G_0^l| \) and \( |G^l| \) be the number of nodes labelled \( x_i \) in \( G_0 \) resp. \( G \). Then

\[
l = O\left( \sum_{1 \leq i \leq n} |G_0^i||G^i| \right) = O(|G_0||G|).
\]

Moreover, the consistency test for LBDDs can be performed in the same time.
Proof. If \( v \) is labelled by \( x_i \), the list \( L(v) \) can contain at most the \( |G_0| \) nodes in \( V_0 \) labelled by \( x_i \). This implies the bound on \( l \).

For the source \( v \) of \( G \) we may define \( L(v) := \text{Succ}(\text{start}, i) \) if \( v \) is labelled by \( x_i \). We proceed in \( G \) according to a topological order of the nodes. If we reach \( v \) labelled by \( x_i \), we consider the list of incoming edges which are the \( c_j \)-edges leaving \( U_j \), \( 1 \leq j \leq r \). The lists \( L(v_j) \) have been already computed. The list \( L(v) \) is the union of all \( \text{Succ}(w, c_j, i) \) for all \( w \in L(v_j) \), \( 1 \leq j \leq r \). If \( v_j \) represents \( w \), then an input for which we reach \( v_j \) in \( G \) and \( w \) in \( G_0 \) can be completed such that we reach \( v \) in \( G \) and \( w' \in \text{Succ}(w, c_j, i) \) in \( G_0 \). There is some problem, if \( \text{Succ}(w, c_j, i) \) is empty. This implies that in \( G_0 \) the variable \( x_i \) is tested on the paths leading to \( w \). We have found an input leading, in \( G \), to \( v \) via \( v_j \), i.e. the variable \( x_k \) which is the label of \( U_j \) is tested before \( x_i \), the label of \( v \), while in \( G_0 \), \( x_i \) is tested before \( x_k \) in contradiction to Definition 2. Hence, the consistency test is included in the computation of the lists \( L(v) \).

The lists for which we have to compute the union have length \( O(l) \). The union can be computed in time \( O(l) \) using an array of length \( |G_0| \) and marking there which elements we have found. The number of lists is \( |G_0| \).

Let us remark that the time bound in Theorem 8 is a worst case estimate. Usually, we shall obtain a much smaller running time.

6. The uniqueness of reduced oracle driven BDDs

In this section we prove the uniqueness (up to isomorphism) of reduced graph driven BDDs which therefore also have minimal size. Moreover, we describe explicitly the structure of the reduced BDD.

Theorem 9. Let \( G_0 \) be an oracle graph and let \( f \) be a Boolean function on the same set of variables. There is up to isomorphism exactly one \( G_0 \)-driven WBDD \( G_{\text{min}} \) computing \( f \) with minimal size. This WBDD \( G_{\text{min}} \) can be described explicitly by properties of \( G_0 \) and \( f \).

Proof. Our proof is organized in the following way. First, we describe the WBDD \( G_{\text{min}} \) and show that it is \( G_0 \)-driven and computes \( f \). Afterwards, we prove that no \( G_0 \)-driven WBDD for \( f \) may have smaller size and that \( G_0 \)-driven WBDDs for \( f \) of the same size as \( G_{\text{min}} \) are isomorphic to \( G_{\text{min}} \). Without loss of generality \( f \) is not a constant function.

We start with the description of the nodes of \( G_{\text{min}} \). For some node \( v \) of \( G_0 \) we describe the set \( V(v) \) of nodes \( w \) of \( G_{\text{min}} \), where \( \alpha(w) = v \). Let \( x_i \) be the label of \( v \). It is known that on all paths from the source of \( G_0 \) to \( v \) the same set of variables is tested. Without loss of generality let \( x_1, \ldots, x_{i-1} \) be the previously tested variables.
Let $A(v) \subseteq \{0, 1\}^{i-1}$ be the set of vectors $(a_1, \ldots, a_{i-1})$ such that $v$ is reached for all inputs $a$ starting with $(a_1, \ldots, a_{i-1})$. Let $S(v)$ be the set of subfunctions $f|_{x_1=a_1, \ldots, x_{i-1}=a_{i-1}}$ with $(a_1, \ldots, a_{i-1}) \in A(v)$. Let $T(v)$ contain the functions $f' \in S(v)$ which depend essentially on $x_i$ and let $U(v) := S(v) - T(v)$.

The set $V(v)$ contains nodes for each function in $T(v)$ and for some functions in $U(v)$. We describe the set $U^*(v)$ of functions $g \in U(v)$ which shall be represented by nodes in $V(v)$. Let $J(v)$ be the first successor of $v$ in $G_0$ which lies on all paths leaving $v$. The node $J(v)$ is well defined, since the unique sink of $G_0$ lies on all paths leaving $v$. By the properties of oracle graphs the same set of variables $Var(v)$ is tested on each path from $v$ to $J(v)$ where $J(v)$ is excluded. The function $g \in U(v)$ is represented by some node in $V(v)$ if and only if $g$ depends essentially on some variable in $Var(v)$.

The node set of $G_{\text{min}}$ consists of the disjoint union of all $V(v)$, a $0$-sink and a $1$-sink. The edges of $G_{\text{min}}$ are defined in the following way. Let us consider a node $w$ labelled $x_i$ representing the subfunction $g$ and the oracle node $v$. For the $c$-successor of $w$ we look at the $c$-successor $v'$ of $v$ in $G_0$. If $g^* := g|_{x_i=c}$ is represented in $G_{\text{min}}$ for $v'$ by $w'$, we choose $w'$ as $c$-successor of $w$ in $G_{\text{min}}$. Otherwise we jump in $G_0$ to $J(v')$. The function $g^*$ does not depend essentially on any of the variables tested in $G_0$ between $v'$ and $J(v')$ where $J(v')$ is excluded. The same process is continued with $g^*$ and $v' := J(v')$. The process stops after a finite number of steps, since finally $v'$ equals the sink of the oracle graph. Then $g^*$ is a constant function since the $c$-successor of $w$ in $G_{\text{min}}$ is the appropriate sink.

By construction we obtain a $G_0$ driven WBDD $G_{\text{min}}$. In order to prove that this BDD computes $f$ we verify for all nodes $w$ that the BDD $G_{\text{min}}(w)$, the sub-BDD of $G_{\text{min}}$ with source $w$, represents the function for which the node $w$ has been included in $V$. This claim is proved by induction on the nodes in $G_{\text{min}}$ with respect to a reversed topological order. The claim holds for the sinks. Let us consider some inner node $w$ with label $x_i$ representing $g$. The $c$-successor $w'$ of $w$ is defined in such a way that it represents $g|_{x_i=c}$. By the induction hypothesis, the claim is proved. The function $f$ is computed by $G_{\text{min}}$, since the source represents $f$.

Finally, we prove that $G_{\text{min}}$ is, up to isomorphism, the unique minimal $G_0$ driven WBDD for $f$. Again it is sufficient to consider the node set $V(v)$. The nodes in $V(v)$ represent by definition different subfunctions of $f$. Hence, it is sufficient to prove that each of these subfunctions has to be represented in $G_{\text{min}}$. This is obvious for the nodes representing functions in $T(v)$. Let $g \in U(v)$. We choose in the oracle graph $G_0$ an arbitrary path between $v$ and $J(v)$ and on this path the first node $v'$ labelled by a variable $x_k$ on which $g$ depends essentially. By definition of $U(v)$ the node $v'$ is well defined and $v' \neq v$. Let us consider a partial input which defines in $G_0$ a path from the source to $v'$ via $v$. Some node between $v$ and $v'$ has to be the first node $z$ represented for this input by a node $z^*$ in $G_{\text{min}}$. We claim that $z = v$. Otherwise $z$ lies behind $v$ and before $J(v)$. Hence, $z^*$ is reached also for some input $b$ such that $z$ is not reached on input $b$. In contradiction to the definition of WBDDs there is no unique representative of $z^*$. Hence, $G_{\text{min}}$ has minimal size. The uniqueness of $G_{\text{min}}$ follows easily, since there was no choice in the definition of the edge set. □
Theorem 10. Let $G_0$ be an oracle graph and let $f$ be a Boolean function on the same set of variables. There is up to isomorphism exactly one $G_0$ driven LBDD $G_{\text{min}}$ computing $f$ with minimal size. This LBDD can be described explicitly by properties of $G_0$ and $f$.

Proof. Let $G_0=(V_0,E_0)$ be the oracle graph. For $v \in V_0$ we denote by $G_0(v)$ the suboracle with source $v$ and by $S(v)$ the set of functions defined in the proof of Theorem 9. Let $A$ be the set of all $(v,g)$, where $v \in V_0$ and $g \in S(v)$. It is obvious that each $G_0$ driven LBDD $G$ for $f$ contains for each $(v,g) \in A$ some node $w(v,g)$ which is the source of a $G_0(v)$ driven LBDD for $g$. We define a relation $\sim$ on $A$. Let $(v,g) \sim (v',g')$ if there exists an LBDD which is a $G_0(v)$ driven LBDD for $g$ and simultaneously a $G_0(v')$ driven LBDD for $g'$. In particular, $g = g'$ if we consider $g$ and $g'$ as functions on all variables $x_1, \ldots, x_n$.

Obviously, the relation $\sim$ is symmetric. Hence, we obtain an undirected graph $G(A)$ on the node set $A$ if we connect $(v,g)$ and $(v',g')$ by an edge, if $(v,g) \sim (v',g')$. We shall construct a $G_0$ driven LBDD for $f$ which contains exactly one node for each connected component in $G(A)$. The minimality of this LBDD $G_{\text{min}}$ is then obvious. The uniqueness (up to isomorphism) will follow easily. The construction of $G_{\text{min}}$ also proves that $\sim$ is indeed an equivalence relation. Before constructing $G_{\text{min}}$ we prove a simple claim.

Claim. Let $v_0$ and $v_1$ be the direct successors of $v \in V_0$ in $G_0$. If $(v,g),(v_0,g),(v_1,g) \in A$ and at least two of these nodes are connected by an edge in $G(A)$, then these nodes are a 3-clique in $G(A)$.

Proof. First we consider the case $(v_0,g) \sim (v_1,g)$. The function $g$ does not depend essentially on the label of $v$. The $G_0(v_0)$ and $G_0(v_1)$ driven LBDD for $g$ is also $G_0(v)$ driven. For the other two cases let us assume without loss of generality $(v,g) \sim (v_0,g)$. Let $G$ be a $G_0(v)$ and $G_0(v_0)$ driven LBDD for $g$. Since $G$ is $G_0(v_0)$ driven, no node of $G$ is labelled by the label of $v$. Since $G$ is also $G_0(v)$ driven, it has to be also $G_0(v_1)$ driven.

Proof of Theorem 10 (continued). We fix a topological ordering of the nodes of $G_0$. For each connected component $C$ of $G(A)$ we select the pair $(v,g)$ such that $v$ is the topologically last node among all nodes $w$, where $(w,g)$ belongs to $C$. The LBDD $G_{\text{min}}$ is constructed with respect to the reversed topological order of the nodes of $G_0$. For each node $v \in V_0$ and each selected pair $(v,g)$ we create a node $w(v,g)$ in $G_{\text{min}}$ and prove (by induction) that $w(v,g)$ is the source of a $G_0(v')$ driven LBDD for $g$ if $(v,g)$ is in the same connected component as $(v,g)$.

The last node of $G_0$ is the sink representing only the constant functions (without loss of generality $f$ is not a constant function). We create the sinks of $G_{\text{min}}$. All pairs $(v,g)$ for constant $g$ can be represented by these sinks.

For the induction step let us consider the selected pair $(v,g)$, where $v$ is labelled by $x_i$. In the following we denote by $v_c$ the $c$-successor of $v$ and by $g_c$ the subfunction of
g for \( x_i = c \). Nodes for the pairs \((v_0, g_0)\) and \((v_1, g_1)\) are already included in \( G_{\text{min}} \). By the claim and the definition of selected pairs, \((v_0, g_0) \not\sim (v_1, g_1)\), i.e. \((v_0, g_0)\) and \((v_1, g_1)\) are represented by different nodes \( w_0 \) and \( w_1 \) in \( G_{\text{min}} \). We create a new node \( w \) with label \( x_i \) and successors \( w_0 \) and \( w_1 \).

We investigate the neighbors \((v', g)\) of \((v, g)\) in \( G(A) \), afterwards the neighbors of \((v'_1, g)\) and so on until we have shown that \( w \) is the source of a \( G_0(v') \)-driven LBDD for all \((v'', g)\) in the connected component of \((v, g)\) in \( G(A) \).

**Case 1.** \( v \) and \( v' \) are not connected in \( G_0 \) and label \((v') = x_i \). Since \((v, g)\) is selected, \((v, g) \not\sim (v_0, g)\) and the source of each \( G_0(v) \)-driven LBDD for \( g \) is labelled by \( x_i \). Hence, the \( G_0(v) \) and \( G_0(v') \)-driven LBDD for \( g \) proves that \((v_0, g_0) \sim (v_1, g_1)\) and \((v_1, g_1) \sim (v'_1, g_1)\). By induction hypothesis \( w \) is the source of a \( G_0(v'_1) \)-driven LBDD for \( g \), where \( c e \{0, 1\} \). Hence, \( w \) is also the source of a \( G_0(v') \)-driven LBDD for \( g \).

**Case 2.** \( v \) and \( v' \) are not connected in \( G_0 \) and label \((v') \neq x_i \). Again the source of each \( G_0(v) \)-driven LBDD for \( g \) is labelled by \( x_i \). Hence, the \( G_0(v) \) and \( G_0(v') \)-driven LBDD for \( g \) is also an \( G_0(v'_1) \)-driven LBDD for \( g \) and \( c e \{0, 1\} \). Therefore, \((v, g) \sim (v_0, g)\) and \((v, g) \sim (v'_1, g)\). We follow the same argumentation for \((v_0, g)\) and \((v'_1, g)\) until we reach Case 1. This proves finally that \( w \) is also the source of a \( G_0(v'_1) \) and \( G_0(v'_1) \)-driven LBDD for \( g \). Since the label of \( w \) is \( x_i \), it is also the source of a \( G_0(v'_1) \)-driven LBDD for \( g \).

**Case 3.** \( v' \) is a predecessor of \( v \) in \( G_0 \). First we assume that \( v' \) is a direct predecessor of \( v \) in \( G_0 \). Without loss of generality \( v = v'_0 \). Since the source of a \( G_0(v') \) and \( G_0(v) \)-driven LBDD for \( g \) cannot be labelled by the label of \( v' \), this LBDD is also \( G_0(v'_1) \) driven and \((v, g) \sim (v'_0, g) \sim (v'_1, g)\). By Cases 1 and 2 the node \( w \) is the source of a \( G_0(v'_0) \) and \( G_0(v'_1) \)-driven LBDD for \( g \). Hence, \( w \) is also the source of a \( G_0(v'_1) \)-driven LBDD for \( g \).

In the general situation of Case 3 the \( G_0(v') \) and \( G_0(v) \)-driven LBDD for \( g \) proves for all nodes \( v'' \) on the path from \( v' \) to \( v \) that \((v'', g) \sim (v, g)\). Hence, we can use the above arguments step by step to prove that \( w \) is also the source of a \( G_0(v') \)-driven LBDD for \( g \).

 Altogether \( G_{\text{min}} \) is a minimal size \( G_0 \) driven LBDD for \( f \). The relation \( \sim \) is an equivalence relation. Minimal size \( G_0 \) driven LBDDs for \( f \) need exactly one node per equivalence class. The label of the node representing an equivalence class of \( \sim \) is necessarily the label of the pair \((v, g)\), where \( v \) is the topological last node represented by this equivalence class. The \( c \)-successor of this node is the unique node which is the source of the \( G_0(v'_1) \) driven LBDD for \( g \). Hence, the \( G_0 \) driven LBDD for \( f \) of minimal size is unique (up to isomorphism). \( \square \)

### 7. Efficient reduction algorithms

Now we are prepared to describe efficient reduction algorithms.

**Theorem 11.** Let \( G \) be a \( G_0 \) driven WBDD for \( f \). The unique reduced \( G_0 \) driven WBDD \( G_{\text{min}} \) for \( f \) can be computed in linear time \( O(|G| + |G_0|) \) with linear storage space, if the successor array \( \text{Succ} \) for \( G_0 \) is given.
Proof. First, we eliminate all nodes not reachable from the source of \( G \). By Theorem 6 and the following remark we can compute in time \( O(|G_0| + |G|) \) the representation function \( \alpha \) and also the lists \( W(v) \) for the nodes \( v \) in \( G_0 \) containing all nodes \( v' \) in \( G \) with \( \alpha(v') = v \).

The deletion rule \( (R1) \) (see Section 5) is applicable to WBDDs but the merging rule \( (R2) \) has to be restricted to nodes \( w \) and \( w' \) where \( \alpha(w) = \alpha(w') \). The reduction algorithm works with respect to a reversed topological order \( v_1, \ldots, v_r \) of the nodes of \( G_0 \). For the node \( v \) in \( G_0 \) the two reduction rules are applied to the nodes in \( W(v) \). The claim is that the resulting WBDD is the reduced one. We prove that after the application of the reduction rules to the nodes in \( W(v_1), \ldots, W(v_k) \) the resulting part of \( G \) is isomorphic to the appropriate part of the reduced WBDD \( G_{\min} \). This claim is proved by induction on \( k \).

For \( k = 1 \) we consider the sink \( v_1 \) of \( G_0 \). We merge all 0-sinks and all 1-sinks. We obtain the same sinks as in \( G_{\min} \). Let the claim be proved for \( k - 1 \). The nodes \( w_1, \ldots, w_l \) in \( W(v_k) \) represent functions in \( S(v_k) \) (defined in the proof of Theorem 9), since all nodes are reachable from the source. If some function \( g \in S(v_k) \) not represented in \( G_{\min} \) is represented by \( w_j \), this node can be deleted by the deletion rule. The reason is that our WBDD is below \( w_j \) isomorphic to \( G_{\min} \) and, by the proof of Theorem 9, the \( G_0(v_k) \) driven WBDDs for \( g|_{x_i=0} \) and \( g|_{x_i=1} \) have the same source. If some function \( g \in S(v_k) \) represented in \( G_{\min} \) is represented in our WBDD by more than one node, these nodes can be merged by the modified merging rule. The reason is again that our WBDD is below the nodes \( w_1, \ldots, w_l \) isomorphic to \( G_{\min} \), i.e. there is a unique source for a \( G_0(v_{k,c}) \) driven WBDD for \( g|_{x_i=c} \), where \( v_{k,c} \) is the \( c \)-successor of \( v_k \) in \( G_0 \) and \( c \in \{0, 1\} \).

The efficient implementation of this algorithm is easy. We have to apply the reduction rules to the lists \( W(v_k) \) for \( k = 1, \ldots, r \). This is exactly the same situation as for list driven BDDs and the list of nodes labelled \( x_i \). The only difference is that we here treat more lists separately. But the total length of all lists is still \( |G| \). Hence, the linear reduction algorithm (linear time and linear space) of Sieling and Wegener \([15]\) can be applied to these lists leading to a linear time and linear space reduction algorithm. \( \square \)

**Theorem 12.** Let \( G \) be a \( G_0 \) driven LBDD for \( f \). The unique reduced \( G_0 \) driven LBDD \( G_{\min} \) for \( f \) can be computed in linear time \( O(|G|) \) with linear storage space \( O(|G|) \).

**Proof.** First we eliminate all nodes not reachable from the source of \( G \). If the resulting LBDD, also called \( \tilde{G} \), has the same number of nodes as \( G_{\min} \), it is by Theorem 10 isomorphic to \( G_{\min} \). We prove that one of the reduction rules is applicable if \( G \) has more nodes than \( G_{\min} \).

The nodes \( w \) of \( G \) correspond to disjoint subsets \( B(w) \) of \( A \) (for the notation see the proof of Theorem 10). Each \( B(w) \) is a subset of some equivalence class \( E \) with respect to \( \sim \). If \( G \) has more nodes than \( G_{\min} \), we investigate a topologically last node \( u \) of
G_{\text{min}} such that the equivalence class \( E(u) \subseteq A \) is represented in \( G \) by more than one node. Let \( W \) be the set of nodes \( w \) in \( G \), where \( B(w) \subseteq E(u) \) and let \( w^* \) be a topologically last node in \( W \).

**Case 1.** There exists some \( w' \in W \) such that \( w^* \) and \( w' \) are not connected by a path in \( G \). In this case we choose \( w' \) in such a way that no successor of \( w' \) is in \( W \) and \( w^* \) and \( w' \) are not connected by a path. From the proof of Theorem 10 it follows that \( w^* \) and \( w' \) are labelled by the same variable \( x_i \) and that they are source of LBDDs for the same function \( g \). By construction the successors of \( w^* \) and \( w' \) belong to that part of \( G \) which is isomorphic to the corresponding part of \( G_{\text{min}} \). Hence, \( w^* \) and \( w' \) can be merged by (R2).

**Case 2.** Each node \( w' \in W \) is connected with \( w^* \) by a path in \( G \). In this case all nodes on the paths from \( w' \) to \( w^* \) in \( G \) represent the same function \( g \). Among all nodes \( w \neq w^* \) on the paths from \( w' \) to \( w^* \) we choose a topologically last node \( w'' \). Then \( w^* \) is a direct successor of \( w'' \). Without loss of generality \( w^* = w_0^* \). Then \( B(w'') \subseteq E(u) \). Since \( w'' \sim w_0^* = w^* \), it follows by the claim in the proof of Theorem 10 that also \( w_i'' \sim w'' \) and \( B(w_i'') \subseteq E(u) \). Hence, by the assumption of this case \( w^* = w_0^* = w_1^* \) are connected by a path, i.e. \( w^* = w_0^* = w_1^* \). Therefore, \( w'' \) can be deleted by (R1).

We conclude that some reduction rule is applicable if \( G \) is not minimal.

For an efficient implementation of the reduction algorithm we would like to identify groups of nodes in \( G \) which may be processed simultaneously and which afterwards will not be merged with other nodes and will not be eliminated later. Let us assume that some bottom part is already reduced, i.e. these nodes will not be merged with other nodes and will not be eliminated later. At the beginning this is the case, if we have merged all 0-sinks and all 1-sinks for the bottom part consisting of the sinks only.

Now we work also with the reversed edges. We run through all edges combining the top part with the bottom part. Let \((v, w)\) be such an edge with \( v \) in the top and \( w \) in the bottom part. We test whether the other edge leaving \( v \) leads to a node in the bottom part. In the negative case we eliminate \((w, v)\) from the list of reversed edges in order to ensure that each reversed edge is used only once. In the positive case we test whether \( v \) can be eliminated. This can be tested definitely now, since the edges leaving \( v \) are leading to the reduced bottom part. If \( v \) can be eliminated we eliminate \( v \) and include all reversed edges \((v, \cdot)\) into the list of edges between the top and the bottom part. If \( v \) cannot be eliminated, we include \( v \) into a list of nodes to be processed in the next phase. After having processed all edges between the top and the bottom part we consider the list of nodes to be processed in the next phase. If we define this set of nodes as the middle part of the graph, it is obvious that for each node \( u \) in the top part there is a path from \( u \) to some sink containing a node of the middle part (see Fig. 3).

We claim that no node \( u \) of the top part will ever be merged with some node \( v \) of the middle part. We have seen that the nodes of the middle part will never be eliminated. Hence, the node \( u \) will always (even after some merging steps) have a successor in the middle part while the node \( v \) will never have a node in the middle part as successor, all successors are in the bottom part. Since nodes of the middle part will never be merged with nodes of the bottom part (by induction) the sub-BDDs with source \( u \) and \( v \) will never become isomorphic.
Hence, the nodes of the middle part can be processed independently from all other nodes. We collect the nodes in the middle part with the same label in lists. By computing a list of all $i$ such that the corresponding list of nodes with label $x_i$ is nonempty we ensure that we never look at empty lists. This is also a trick described in detail in [15]. The order in which we treat the lists is arbitrary. Hence, the reduction algorithm of Sieling and Wegener [15] can be applied to the middle part.

Afterwards we include the surviving nodes of the middle part into the bottom part. Only the edges entering these surviving nodes are now edges connecting the new top part with the new bottom part which have not been processed before.

We start the whole process again until we have processed the source of $G$. The total running time and the storage space are linear, since this holds for all applications of the reduction algorithm which work on disjoint sets of nodes and since each edge is considered only once in the reversed direction. □

8. Further efficient algorithms for WBDDs

We start with the fundamental synthesis problem.

**Theorem 13.** Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be $G_0$ driven WBDDs for the functions $f_1$ and $f_2$ and let $\circ$ be a binary Boolean operation. Let the functions of representatives $\alpha_1$ and $\alpha_2$ and the successor array $\text{Succ}$ for $G_0$ be given. A $G_0$ driven WBDD $G$ for $f_1 \circ f_2$ can be computed in time $O(|G_1||G_2|)$ with storage space $O(|G|)$.
**Proof.** We begin with an informal description of the ideas. The value of \( f_1 \circ f_2(a) \) can be computed easily from \( G_1 \) and \( G_2 \). The bits of \( a \) are tested in \( G_1 \) and \( G_2 \) in the same order but some bits may be not tested in \( G_1 \) and/or \( G_2 \). We run simultaneously through \( G_1 \) and \( G_2 \). Assume that we have reached \( v_1 \) in \( G_1 \) and \( v_2 \) in \( G_2 \). Then \( \alpha_1(v_1) \) and \( \alpha_2(v_2) \) lie on the path for the input \( a \) in \( G_0 \). Hence, \( \alpha_1(v_1) \) is a successor of \( \alpha_2(v_2) \) in \( G_0 \) or vice versa or \( \alpha_1(v_1) = \alpha_2(v_2) \). Using the successor array for \( G_0 \) we can test in constant time which of the three cases occurs. In the first case we test the variable \( x_i \) which is the label of \( \alpha_2(v_2) \) and follow the \( a_1 \)-edge leaving \( v_2 \) in \( G_2 \). In the second case we test the variable \( x_j \) which is the label of \( \alpha_1(v_1) \) and follow the \( a_2 \)-edge leaving \( v_1 \) in \( G_1 \). In the third case we test each variable only once and the order is \( G_0 \) driven. Finally, we reach the \( c_1 \)-sink of \( G_1 \) and the \( c_2 \)-sink of \( G_2 \). Then we know that \( f_1 \circ f_2(a) = c_1 \circ c_2 \). The following graph \( G = (V, E) \) collects all information in a \( G_0 \) driven WBDD.

The node set \( V \) is a subset of \( V_1 \times V_2 \). If \( v_1 \) and \( v_2 \) are sinks in \( G_1 \) resp. \( G_2 \) labelled \( c_1 \) resp. \( c_2 \), \( (v_1, v_2) \) is labelled by \( c_1 \circ c_2 \). Obviously we can merge all 0-sinks and all 1-sinks. If \( v_1 \) is a \( c_1 \)-sink and \( v_2 \) is not a sink, we may test whether \( c_1 \circ x \) is a constant \( c \), i.e. \( c_1 \) controls \( x \), then \( (v_1, v_2) \) is merged with the \( c \)-sink. If \( c_1 \) does not control \( x \), we label the node \( (v_1, v_2) \) by the label of \( \alpha_2(v_2) \), include it in \( V \) and set \( \alpha(v_1, v_2) := \alpha_2(v_2) \). The symmetric case is handled similarly. Let \( v_1 \) and \( v_2 \) be nonsinks. If \( v_1 \) and \( v_2 \) are labelled by \( a_2(v_2) \) and \( \alpha_1(v_1) \) and \( \alpha_2(v_2) \) and follow the \( a_1 \)-edge leaving \( v_1 \) in \( G_1 \) and the \( a_2 \)-edge leaving \( v_2 \) in \( G_2 \). By this procedure we test each variable only once and the order is \( G_0 \) driven. Finally, we reach the \( c_1 \)-sink of \( G_1 \) and the \( c_2 \)-sink of \( G_2 \). Then we know that \( f_1 \circ f_2(a) = c_1 \circ c_2 \). The following graph \( G = (V, E) \) collects all information in a \( G_0 \) driven WBDD.

The node set \( V \) is a subset of \( V_1 \times V_2 \). If \( v_1 \) and \( v_2 \) are sinks in \( G_1 \) resp. \( G_2 \) labelled \( c_1 \) resp. \( c_2 \), \( (v_1, v_2) \) is labelled by \( c_1 \circ c_2 \). Obviously we can merge all 0-sinks and all 1-sinks. If \( v_1 \) is a \( c_1 \)-sink and \( v_2 \) is not a sink, we may test whether \( c_1 \circ x \) is a constant \( c \), i.e. \( c_1 \) controls \( x \), then \( (v_1, v_2) \) is merged with the \( c \)-sink. If \( c_1 \) does not control \( x \), we label the node \( (v_1, v_2) \) by the label of \( \alpha_2(v_2) \), include it in \( V \) and set \( \alpha(v_1, v_2) := \alpha_2(v_2) \). The symmetric case is handled similarly. Let \( v_1 \) and \( v_2 \) be nonsinks. If \( v_1 \) and \( v_2 \) are labelled by \( a_2(v_2) \) and \( \alpha_1(v_1) \) and \( \alpha_2(v_2) \) and follow the \( a_1 \)-edge leaving \( v_1 \) in \( G_1 \) and the \( a_2 \)-edge leaving \( v_2 \) in \( G_2 \). By this procedure we test each variable only once and the order is \( G_0 \) driven. Finally, we reach the \( c_1 \)-sink of \( G_1 \) and the \( c_2 \)-sink of \( G_2 \). Then we know that \( f_1 \circ f_2(a) = c_1 \circ c_2 \). The following graph \( G = (V, E) \) collects all information in a \( G_0 \) driven WBDD.

Let us now consider a node \( (v_1, v_2) \) in \( V \) with label \( x_i \). The c-successor is the node \( (v_1', v_2') \) where \( v_1' \) is the c-successor of \( v_1 \) in \( G_1 \), if \( v_1 \) is labelled by \( x_i \), and \( v_1' := v_1 \) otherwise, and where \( v_2' \) is defined similarly. It is easy to see that \( (v_1', v_2') \in V \). By construction \( G \) is a \( G_0 \) driven WBDD for \( f_1 \circ f_2 \) and \( |G| \leq |G_1| \cdot |G_2| \). Because of our consistency test with \( G_0 \), \( |G| \) is often much smaller than \( |G_1| \cdot |G_2| \). \( \Box \)

Our construction does not ensure that all nodes in \( G \) are reachable from the source and that the graph of reachable nodes builds a reduced \( G_0 \) driven WBDD for \( f_1 \circ f_2 \). The synthesis algorithm of Bryant \([7, 8]\) for list driven BDDs does not create nodes not reachable from the source. The running time is not always linear with respect to the size of the reduced BDD but it is always bounded by \( O(|G_1| \cdot |G_2|) \). The nodes of \( G \) are computed in depth first order. An extra array of size \( O(|G_1| \cdot |G_2|) \) is needed to ensure that for every pair of nodes \( (v_1, v_2) \in V_1 \times V_2 \) at most one node in the BDD for \( f_1 \circ f_2 \) is created. The algorithm can be extended so that it performs the reduction simultaneously bottom-up \([5]\). For this purpose it is necessary to test for \((i, v, w)\)
whether a node labelled $x_i$ with successors $v$ and $w$ has already been constructed. This test can always be done in constant time if an array of size $O(n|G_1|^2|G_2|^2)$ is used which often is not available. Using dictionaries (like AVL trees or 2-3 trees) instead of the two arrays the storage space can be reduced but we have to accept an extra factor of $O(\log |G_1| + \log |G_2|)$ for the running time. In practice the following compromise is reasonable. A hash table is used as dictionary. This leads to a good average case behaviour but one has to accept a bad worst case behaviour. The same ideas can be used to compute directly the reduced $G_0$ driven WBDD for $f_1 \circ f_2$.

The redundancy test for reduced list driven BDDs is very simple, since we have to test only whether the reduced BDD contains a node labelled by $x_i$. This approach works for graph driven WBDDs only if the oracle graph $G_0$ is $x_i$-obvious (see Definition 5). Otherwise a reduced graph driven WBDD may contain a node labelled by $x_i$ even if the represented function does not depend essentially on $x_i$. This test of $x_i$ may be necessary to ensure that the WBDD is $G_0$ driven.

Theorem 14. The redundancy test for graph driven WBDDs is as difficult as the equality test for read-once branching programs.

Proof. We start with a reduction of the equality test problem for read-once branching programs to the redundancy test problem for $G_0$ driven WBDDs. Let $G_1$ and $G_2$ be read-once branching programs where we like to get whether they represent the same function. We compute the following input for the redundancy test problem. Let $x^*$ be a new variable. Let $G$ be the read-once branching program whose source is labelled by $x^*$ and where the 0-edge leaving the source leads to the source of $G_1$ and the 1-edge leaving the source leads to the source of $G_2$. Let $G_0$ be an oracle graph such that $G$ is a $G_0$ driven WBDD (see Theorem 1). Then $G_1$ and $G_2$ represent the same function if and only if $x^*$ is redundant for $G$.

The other direction is even simpler. The redundancy test problem is by definition the equality test for $g_0 := f_{x_i=0}$ and $g_1 := f_{x_i=1}$. It is obvious how to obtain read-once branching programs for $g_0$ and $g_1$ from a graph driven WBDD for $f$. □

Blum et al. [4] have proved that the equality test for read-once branching programs is contained in $\text{co-RP}$, i.e. the inequality can be tested probabilistically with one-sided error in polynomial time but no deterministic polynomial time algorithm is known. The redundancy test is not important enough to conclude that Theorem 14 leads to a major drawback for graph driven BDDs. Nevertheless we mention the result that the situation is much better if $G_0$ is a tree oracle.

Theorem 15. The redundancy test for $G_0$ driven WBDDs $G$ can be performed in time $O(|G_0| |G|^4)$ if $G_0$ is a tree oracle.

Proof. Let $f$ be the function computed by $G$. The function $f$ does not depend essentially on $x_i$ if for each node $v$ in $G$ labelled by $x_i$ the function computed at the
1-successor \( v_1 \) is the same as the function computed at the 0-successor \( v_0 \). Let us consider such a node \( v \). Let \( \alpha(v) \) be the corresponding node in \( G_0 \) and let \( T_0 \) and \( T_1 \) be the subtrees in \( G_0 \) whose common father is \( \alpha(v) \). In \( T_0 \) and \( T_1 \) the same set of variables is tested. We choose the smaller tree, without loss of generality \( T_0 \), which contains at most \(|T_0|\) paths from the source to the sink. For each path \( p \) we compute the minimal set of variables which have to be set to constants such that the computation proceeds along this \( G_0 \)-path. We replace in \( G \) the variables in the described way. Because we have chosen a path in the tree \( T_0 \), the sub-WBDD of \( G \) with source \( v_0 \) is changed to a list driven BDD while the sub-WBDD of \( G \) with source \( v_1 \) is a read-once branching program. The equality test for a list driven BDD and a read-once branching program is possible with an algorithm due to Fortune et al. [10] in time \( O(|G|^3) \). We may have \( O(|G|) \) nodes labelled by \( x_i \) each leading to \( O(|G_0|) \) applications of the equality test algorithm. 

The replacement of a variable may increase the size of reduced \( G_0 \) driven WBDDs exponentially. This perhaps surprising fact can be shown easily. We consider the special ALU function \( f \) defined in Section 3 and the tree driven WBDD of linear size representing \( f \). The function

\[ f^* := f_{y=0} \lor f_{y=1} \quad (= \exists y f) \]

has only read-once branching programs of exponential size. The proof of this fact is omitted here. Hence, \( f_{y=0} \) or \( f_{y=1} \) cannot be represented for the given tree oracle (with only one branching variable) by a WBDD of nonexponential size. Otherwise we could use the synthesis algorithm to represent \( f^* \) in nonexponential size.

**Theorem 16.** The problem replacement (of \( x_i \)) by a constant can be solved for \( G_0 \) driven WBDDs in linear time if the oracle \( G_0 \) is \( x_i \)-oblivious.

**Proof.** We always obtain a free BDD \( G' \) for \( f_{x_i=0} \) by replacing all edges to nodes \( v \) labelled \( x_i \) by edges to the \( c \)-successor of \( v \) and by eliminating the nodes labelled \( x_i \). If the oracle \( G_0 \) is \( x_i \)-oblivious, the resulting BDD \( G' \) is a \( G_0 \) driven WBDD, since for all inputs \( a \) and \( a' \) differing only at position \( i \) the oracle describes the same ordering of the variables. 

**Theorem 17.** The problem replacement (of \( x_i \)) by a function can be solved for \( G_0 \) driven WBDDs in time \( O(|G_f|^2 |G_g|) \) if the oracle graph \( G_0 \) is \( x_i \)-oblivious.

**Proof.** This proof is similar to the proof for OBDDs. Remember that

\[ f_{x_i=g} := (f_{x_i=0} \land \bar{g}) \lor (f_{x_i=1} \land g) \]

is a ternary operation on \( f_{x_i=0} \), \( f_{x_i=1} \) and \( g \). Hence, we use a straightforward generalization of our synthesis algorithm to ternary operations. □
Theorem 18. The equality test for $G_0$ driven WBDDs $G_f$ and $G_g$ can be performed in linear time.

Proof. First the $G_0$ driven WBDDs $G_f$ and $G_g$ are reduced and then the reduced WBDDs are tested whether they are isomorphic. This test can be performed in linear time, since the graphs are directed and acyclic and since the edges leaving a node have different labels. □

The model of well-structured graph driven BDDs has turned out to be a generalized BDD model which often leads to a much more succinct representation of Boolean functions. Many operations can be performed for WBDDs as efficiently as for OBDDs. The exceptions are the redundancy test and the problems where variables are replaced by constants or functions. These problem can be solved efficiently only if the oracle graph is $x_i$-oblivious for the variable $x_i$ which has to be replaced.

9. Further efficient algorithms for LBDDs

Again we start with the fundamental synthesis problem.

Theorem 19. Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be $G_0$ driven LBDDs for the functions $f_1$ and $f_2$ and let $\circ$ be a binary Boolean operation. A $G_0$ driven WBDD $G$ for $f_1 \circ f_2$ of size $O(|G_0||G_1||G_2|)$ can be computed in time $O(|G_0||G_1||G_2|)$ with storage space $O(|G|)$.

It is possible to define examples where the reduced $G_0$ driven LBDD has size $\Theta(|G_0||G_1||G_2|)$. Hence, we can in general not hope for better results. A simple example has already been discussed for $f_1 = x_i, f_2 = x_j, j \neq i, \circ = \land$ and a suitable oracle graph $G_0$. We will see that our algorithm often constructs smaller BDDs in shorter time. The algorithm constructs a WBDD, rather than merely an LBDD.

Proof of Theorem 19. The node set of $G$ is a subset of $V_0 \times V_1 \times V_2$, since we simulate running with an input $a$ simultaneously through $G_0$, $G_1$ and $G_2$. Let us assume without loss of generality that $G_0$ prescribes the variable ordering $x_1, \ldots, x_n$ for input $a$. In $G$ we must reach the vertex $v = (v_0, v_1, v_2)$ for the partial input $(a_1, \ldots, a_i)$ if for this partial input $v_0$ is reached in $G_0$, $v_1$ in $G_1$ and $v_2$ in $G_2$. Hence, the source of $G$ is the triple consisting of the sources of $G_0$, $G_1$ and $G_2$. The label of $v = (v_0, v_1, v_2)$ is equal to the label of $v_0$ if $v_0$ is not the sink of $G_0$, since in $G_0$ each variable is tested on each path. If $v_0$ is the sink of $G_0$, $v_1$ is a sink of $G_1$ and $v_2$ a sink of $G_2$. Then $(v_0, v_1, v_2)$ is a $b$-sink of $G$ where $b = b_1 \circ b_2$ for the labels $b_1$ and $b_2$ of $v_1$ and $v_2$, respectively. The $c$-successor $v_c$ of a nonsink node $v$ is equal to $(v_0, v_1, v_2)$ where $v_f$ is the successor of $v_f$ in $G_f$ if $v_f$ has the same label as $v_0$ and $v_f = v_c$ otherwise ($j \in \{0, 1, 2\}$). The resulting
BDD is obviously \( G_0 \) driven and computes \( f \). It is a WBDD, since \( \alpha((v_0, v_1, v_2)) := v_0 \) is a representation function and it can be computed within the resource bounds stated in the theorem. □

The algorithm described in the proof of Theorem 19 can be improved. Similarly to the synthesis algorithm for WBDDs we may replace a node \((v_0, v_1, v_2)\) by a sink if \( v_1 \) or \( v_2 \) is a \( c \)-sink and this constant controls the operation \( \circ \). In order to construct only the nodes reachable from the source the nodes are constructed in depth first order starting with the source of \( G \). One needs a dictionary (AVL trees or hash tables) to prevent multiple copies of the same node \((v_0, v_1, v_2)\). Also the reduction process can be integrated into the synthesis process. Because of the depth first construction the BDD below a node \( v = (v_0, v_1, v_2) \) is already reduced if we backtrack to \( v \) from the second edge. Then it can be checked easily whether \( v \) can be deleted by (R1). If \( v \) cannot be deleted we ask whether \( v \) can be merged with some node \( w \) for which the BDD below \( w \) is already constructed. For this purpose we use a second dictionary to store the label and the pair of direct successors for all constructed nodes. If we use the merging rule for WBDDs we obtain finally a reduced WBDD and if we use the more general merging rule for LBDDs we obtain finally a reduced LBDD which may be much smaller. If \( v \) cannot be merged with some other node, we declare \( v \) as constructed and insert the label of \( v \) and the pair of direct successors into the second dictionary. The construction of the reduced WBDD takes almost the same time as the construction of the reduced LBDD. The WBDD construction takes a little longer, since the dictionaries may have more entries. Typically, the construction of the reduced LBDD saves a considerable amount of space.

Now we are able to describe algorithms for the translation of WBDDs into LBDDs and vice versa. A WBDD is by definition an LBDD but a reduced WBDD \( G \) is not necessarily a reduced LBDD. The reduction is possible in time \( O(|G|) \). Since \( f \land 1 = f \), we may apply the synthesis algorithm for LBDDs to the oracle \( G_0 \), the LBDD \( G \) and the trivial LBDD for the constant 1 to construct in time \( O(|G| \mid G_0 \mid) \) a reduced WBDD for the function represented by \( G \). It also follows that a reduced WBDD is larger than the reduced LBDD for the same function by at most a factor of \( |G_0| \).

The proofs of the Theorems 14–18 work also for \( G_0 \) driven LBDDs. Hence, we obtain similar results for LBDDs. The time bounds are the same as for WBDDs with the only exception of the problem replacement by a function. Here we obtain an additional factor of \( |G_0| \), since the synthesis algorithm for LBDDs is applied.

10. Conclusion

The model of OBDDs or in our notation list driven BDDs is a data structure for Boolean functions suitable in many situations. If the functions investigated become too difficult to be represented by small size OBDDs but are simple enough to be represented by small size read-once branching programs, graph driven BDDs are
powerful enough to represent those functions in small size. If we can represent all the
functions we consider by BDDs driven by the same graph \( G_0 \), many of the usual
operations on data structures for Boolean functions can be performed with theoretically
and practically efficient algorithms. If a variable \( x_i \) has to be replaced or has to
be tested for redundancy, the algorithms are only efficient if the oracle graph is
\( x_i \)-oblivious. Examples where graph driven BDDs are definitely exponentially more
succinct then list driven BDDs have been presented.

References

Boolean functions, Tech. Report, Univ. of Kentucky, Lexington.
(1986) 617–691.
single variable program schemes, *5th ICALP*, Lecture Notes in Computer Science, Vol. 62 (Springer,
branching programs, *10th STACS*, Lecture Notes in Computer Science, Vol. 665 (Springer, Berlin,
[13] C. Meinel, Polynomial size \( \Omega \)-branching programs and their computational power, *Inform. and
139–144.