Differential BDDs *

Anuchit Anuchitanukul   Zohar Manna

Computer Science Department
Stanford University
Stanford, CA 94305
anuchit@cs.stanford.edu

September 9, 1994

Abstract

In this paper, we introduce a class of Binary Decision Diagrams (BDDs) which we call Differential BDDs (ΔBDDs), and two transformations over ΔBDDs, called Push-up ($\uparrow$) and Delta (δ) transformations. In ΔBDDs and its derived classes such as $\uparrow\Delta$BDDs or $\delta\uparrow\Delta$BDDs, in addition to the ordinary node-sharing in the normal Ordered Binary Decision Diagrams (OBDDs), some isomorphic substructures are collapsed together forming an even more compact representation of boolean functions. The elimination of isomorphic substructures coincides with the repetitive occurrences of the same or similar small components in many applications of BDDs such as in the representation of hardware circuits. The reduction in the number of nodes, from OBDDs to ΔBDDs, is potentially exponential while boolean manipulations on ΔBDDs remain efficient.

1 Introduction

Binary Decision Diagrams (BDDs) are a diagrammatic representation of boolean functions [Ak78] and have been widely accepted as an important tool in a variety of applications. It was recognized that many problems in digital system design, verification, optimization of combinatorial circuits, and test pattern generation, can be formulated as a sequence of operations on boolean functions. This, in return, indicates a need for efficient manipulations on BDDs.

Ordered Binary Decision Diagrams (OBDDs) [Br86,Br92] are BDDs with a constraint that the sequence of variables on any path of an OBDD conforms to a particular ordering and all the nodes representing the same logic function are merged together. Their property of having strong canonical forms and efficient manipulations makes OBDDs the standard representation used in most applications of BDDs.

With the demand to apply BDDs to increasingly larger problems, there have been many suggestions for efficient implementation techniques [BRB90] and various variations of OBDDs, to improve speed and to save memory. Among these variations, the use of typed edges [MB88], and later attributed edges [MIY90], was proposed and shown to reduce the size of the graphs in many practical cases.

In this paper, we define and analyze a new class of BDDs, called Differential BDDs (ΔBDDs), and explore two transformations over ΔBDDs, called the Push-up ($\uparrow$) transformation and its variation called the Delta (δ) transformation. These techniques have the potential to significantly reduce the size of the graphs and

*This research was supported in part by the National Science Foundation under grant CCR-92-23226, by the Defense Advanced Research Projects Agency under contract NAG2-892, and by the United States Air Force Office of Scientific Research under contract F49620-93-1-0139.

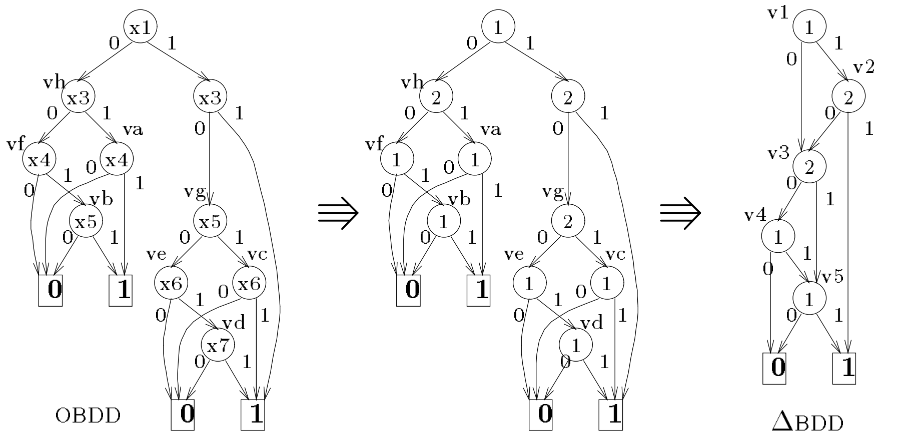

Figure 1: ΔBDD vs OBDD

save memory space. We also explain efficient algorithms for manipulating an interesting class of BDDs, namely $\uparrow\Delta$BDDs, which are derived from ΔBDDs by a $\uparrow$ transformation.

2 ΔBDDs

A Differential Binary Decision Diagram (ΔBDD) is a labeled binary directed acyclic graph (DAG) in which each node is labeled by an integer, representing the variable displacement in the ordering of input variables.

Formally, a ΔBDD $\mathcal{G}$ is a tuple (V, s, l) where

- V is a set of nodes, which includes two special *sink* nodes, $\mathbf{0}$ and $\mathbf{1}$.
- $s : (V - \{\mathbf{0}, \mathbf{1}\}) \times \{0, 1\} \mapsto V$ is the successor function.
- l is a labeling function, mapping a node to an integer.

For each $v \in V - \{\mathbf{0}, \mathbf{1}\}$, there are exactly two successors of v, denoted by $s(v, 0)$ and $s(v, 1)$, and a label $l(v)$. The label $l(v)$ represents the difference in the indexes of the input variables associated with the node v and its predecessor. For example, in Figure 1, the label $l(v2) = 2$ is the distance in the variable ordering of $x1$ and $x3$. For the root node, the label is the index of the input variable, associated with the root node. In the special cases when $v \in \{\mathbf{0}, \mathbf{1}\}$, there are no successors and $l(v) = 0$. If there are n input variables, $x_1 \ldots x_n$, then the sum of all labels on any path must be less than or equal to n and for any $v \in V$, $0 \le l(v) \le n$.

Like OBDDs, we require that for any $v_1, v_2 \in V - \{\mathbf{0}, \mathbf{1}\}$, if $l(v_1) = l(v_2)$ and $s(v, i) = s(v, i)$ for all $i \in \{0, 1\}$, then $v_1 = v_2$.

Shown in Figure 1 is the comparison between the OBDD and ΔBDD representation of the same boolean function. Here, $\{va, vb, vc, vd\}$ are merged into $v5$, $\{ve, vf\}$ into $v4$, and $\{vg, vh\}$ into $v3$.

Later on, we may drop the 0 and 1 annotations on the edges and assume that the left and right branches of a node v always lead to its successors, $s(v, 0)$ and $s(v, 1)$, respectively.

For a ΔBDD $\mathcal{G} = (V, E, l)$, we define the value $val(v, B)$ of a boolean function represented by a node $v \in V$, at a given input bit vector $B = b_1 \ldots b_n$, recursively as follows:

$$val\,(v, B) = \begin{cases} 0 & \text{if } v = \mathbf{0} \\ 1 & \text{if } v = \mathbf{1} \\ val\left(s(v, b_{l(v)}), B'\right) & \text{otherwise} \\ \quad \text{where} \quad B' = b_{l(v)+1} \ldots b_n & \end{cases}$$

For example, the evaluation of the ΔBDD in Figure 1 at bit vector 0010111 is:

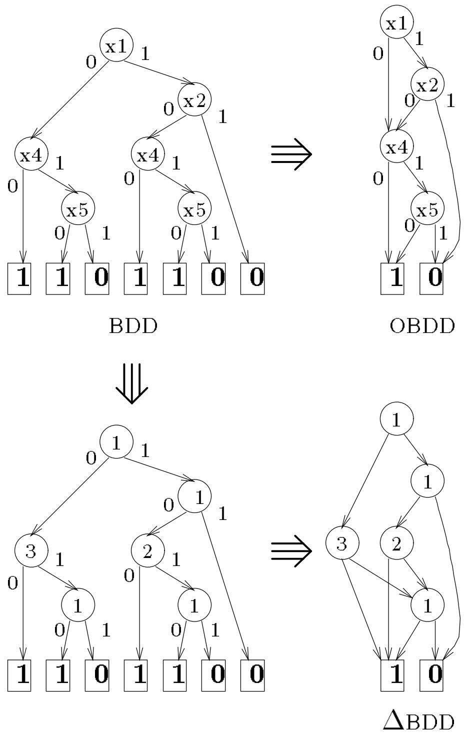

Figure 2: A bad case for ΔBDDs

$$\begin{aligned}\text{val(v1,0010111)} &= \text{val(s(v1,0),010111)}\\ &= \text{val(v3,010111)}\\ &= \text{val(v5,0111)}\\ &= 0\end{aligned}$$

Proposition 2.1 *ΔBDDs have the following properties:*

1. *(Canonicity) Each boolean function has a unique representation (canonical form) in ΔBDDs.*

2. *The reduction in the number of nodes by switching from OBDDs to ΔBDDs is potentially exponential. Namely, if m is the number of nodes of an OBDD, then in the best case, the number of nodes of a ΔBDD representing the same boolean function will be $O(log(m))$. On the other hand, as a function of the number input variables, n, the reduction of nodes is at best up to a factor of $1/n$.*

3. *Boolean operations on ΔBDDs are polynomial in the size of ΔBDDs.*

Proof Outline: It is easy to show that there is a unique translation back and forth between ΔBDDs and OBDDs. From the canonicity propery of OBDDs, we can conclude that ΔBDDs have the canonicity property as well.

Exponential reduction occurs in the cases where the left and right subgraphs of every node in an OBDD are isomorphic. and if x_k is the label of a node v in the left subgraph then the counterpart node of v in the right subgraph is labeled by x_{k+c}, for some big enough c (e.g. bigger than the index of any of the labels in the left subgraph. In these cases, when translated into ΔBDDs, the left and right subgraphs will be merged together repeatedly and the size of the ΔBDDs will be $O(log(m))$. The reduction factor of $1/n$ is obvious since a node in ΔBDDs can replace at most n nodes of the original OBDDs.

Efficient boolean manipulations of ΔBDDs and their analysis are very similar to those described later in section 4. ∎

It is important to note that for some cases, the ΔBDD representation may yield more nodes than the OBDD representation of the same boolean function. Figure 2 demonstrates such a case. This, however, should not be considered as a drawback, because of the following proposition.

Proposition 2.2 *There exists a one-to-one correspondence mapping between all possible OBDDs and ΔBDDs of the same set and ordering of input variables, such that each ΔBDD is mapped to an identical OBDD up to relabeling, and vice versa.*

Proof Outline: First, assume that the variable ordering is decreasing, i.e. the variable indices decrease from the top of OBDDs to the leaves.

To construct an OBDD from a ΔBDD, start from the leaves (sink nodes) and move upward, adding to the label of each node the larger number between the labels of its two successors. Then, for each node with the label n, relabel it

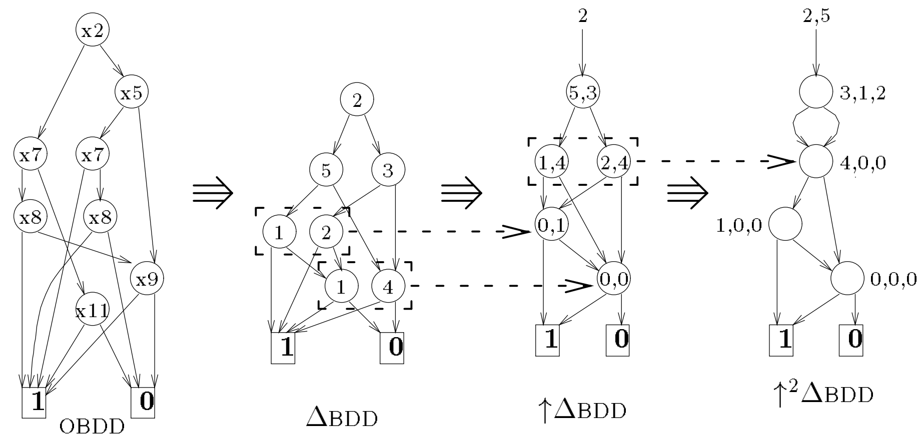

Figure 3: Push-up

with x_n. To check that the result is actually an OBDD, we show that no duplicate nodes are created, by induction on the structure of the subgraphs. In doing so, we use the constraint of ΔBDDs that there are no duplicate nodes in the ΔBDD that we start with.

The other direction, from an OBDD to a ΔBDD, is the reverse of the above construction. ∎

With the canonicity property of both OBDDs and ΔBDDs, we can conclude that for any set of input variables and a variable ordering, the average number of nodes in a ΔBDD is the same as the average number of nodes in an OBDD.

In the next section, we will introduce the $\uparrow$ transformation which can be applied to ΔBDDs, resulting in $\uparrow\Delta$BDDs. $\uparrow\Delta$BDDs inherit the potentially exponential reduction property and easy polynomial boolean operations, and at the same time, guarantee to give a structure with fewer or the same number of nodes.

3 Push-up Transformation

The Push-up $\uparrow$ transformation takes a labeled DAG with n labels on each node, and pushes one of the labels of each node through the incoming edges back to the predecessors of the node.

A binary directed acyclic graph with n labels, $\mathcal{G} = (V, s, l)$, consists of a set V of nodes, a successor function $s : V \times \{0,1\} \mapsto V$, and a labeling function l which maps each node $v \in V$ to an n-tuple label $l(v) = \langle d_0, \ldots, d_{n-1} \rangle$. For each $v \in V$, we will denote the j-th label d_j of v by $d(v, j)$. Given a binary DAG $\mathcal{G} = (V, s, l)$ with n labels, the transformation $\uparrow_i$ (or simply $\uparrow$ if $i = 0$) transforms $\mathcal{G}$ in two steps:

1. relabel each node v with
$$\langle d_0, \ldots, d_{i-1}, d_{i+1}, \ldots, d_{n-1}, d_{s0}, d_{s1} \rangle$$
$$\text{where} \quad \begin{aligned} d_{s0} &= d(s(v,0), i) \\ d_{s1} &= d(s(v,1), i). \end{aligned}$$
If v has no successor (i.e. it is a sink node), then $d_{s0} = d_{s1} = 0$.

2. merge together all the nodes which have the same successors and the same label.

The result is a binary DAG with $n+1$ labels which represents $\mathcal{G}$.

Figure 3 shows the result of applying $\uparrow$ transformations, once ($\uparrow\Delta$BDD) and twice ($\uparrow^2\Delta$BDD), on a ΔBDD. The nodes in each dashed box are merged into one node on the right. Unlike in OBDDs or ΔBDDs, where a boolean function is represented by a node, in $\uparrow\Delta$BDDs, a boolean function is identified by a node *and* an integer, which is the by-product of the transformation.

Proposition 3.1 *Each application of $\uparrow$ to OBDDs or ΔBDDs has the following properties:*

1. *The number of nodes in always reduced up to a factor of $1/n$ where n is the number of input variables.*
2. *Canonicity is preserved.*
3. *Boolean operations on the derived class of BDDs remain polynomial time.*

Proof Outline: For the canonicity property, observe that the $\uparrow$ transformation can be easily shown reversible. Therefore, two OBDDs (ΔBDDs) cannot be transformed into the same graph. OBDDs (ΔBDDs) are canonical and so are the derived classes. ∎

Even though this transformation offers the best case reduction of $1/n$ (compared to the constant factor of 1/2 reduction offered by other methods such as using complement edges), there is a trade-off: each transformation introduces one additional label per node. Therefore, in order to save memory space with the transformation, the following condition must hold:

$$\frac{N_{elim}}{N} > \frac{1}{1 + \frac{|l|}{|dx|}}$$

where

- N is the total number of nodes before the transformation.
- N_{elim} is the number of nodes which are eliminated by the transformation.
- $|l|$ is the total bit-length of all the labels and pointers associated with each node before the transformation. For example, for OBDDs and ΔBDDs, $|l|$ equals $log(n) + 2b$ where n is the number of input boolean variables and b is the bit-length of a pointer.
- $|dx|$ is the number of bits needed to hold the additional label for each node, which should, in fact, be $log(n)$.

There are two supporting reasons for using the $\uparrow$ transformation.

1. $|dx|$ is generally small because there are usually not too many variables. On the other hand, $|l|$ is generally long because each of the two pointers could be 32-bit or 64-bit long and this number is getting larger as the new CPUs on the market have a higher and higher memory addressing capacity. As a result, $1 + |l|/|dx|$ will be a large number (e.g. $5, 6, 7, 8 \ldots$). Thus, we only need to be able to eliminate a small fraction N_{elim}/N of the nodes, in order to save memory space.
2. Some unused bits may be used to hold the additional label. For example, if a word in the system is 16-bit long, but we only need 8 bits per label (i.e. there are fewer than 65536 input variables), then we should apply the $\uparrow$ transformation and keep the additional label in the unused bits. If this is the case, and since the transformation will always reduce or maintain the number of nodes, the transformation is guaranteed to save memory space or at least keep it the same.

4 $\uparrow\Delta$BDDs

A particular class of BDDs, namely $\uparrow\Delta$BDDs, has an interesting property.

Proposition 4.1 *The number of nodes in a $\uparrow\Delta$BDD is always less than or equal to the number of nodes in an OBDD of the same boolean function and the same variable ordering. Moreover, the reduction of nodes is potentially exponential.*

Proof Outline: We can show that each node in an OBDD is translated into at most one node in a $\uparrow\Delta$BDD by induction on the structure of subgraphs. ∎

This property increases the chance that N_{elim}/N, the number of nodes eliminated by switching from OBDDs to $\uparrow\Delta$BDDs divided by the total number of nodes in the OBDD representation, will be greater than $1/(1 + |l|/|dx|)$.

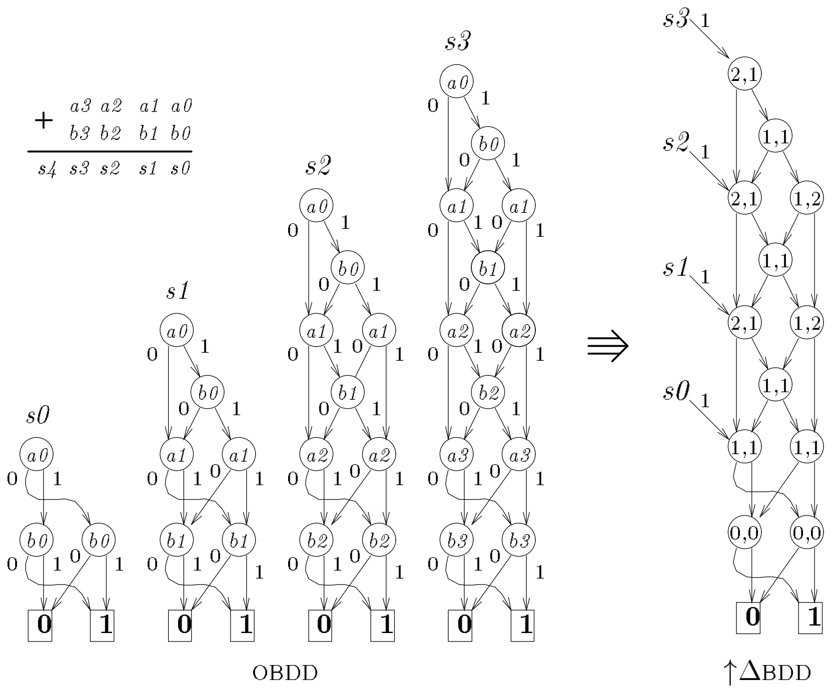

Figure 4: four-bit adder

In some applications of BDDs, particularly in the area of hardware circuit representation, the ability to eliminate some isomorphic structures in the BDD representation is a significant advantage. Many circuits compose of small identical components that cause the BDD representation of the circuits to have isomorphic structures which differ only in variable labeling (input signals to each small component).

Figure 4 shows the ↑ΔBDDs representation of a four-bit adder. The two operands are $a_3a_2a_1a_0$ and $b_3b_2b_1b_0$ and the variable ordering is $a_0b_0a_1b_1 \dots a_3b_3$, i.e. from the least significant bits (a_0 and b_0 are checked first) to the most significant bits. For the output, s_0 and s_3 are the least and most significant bits of the sum, respectively.

We present the following algorithm as an example, to demonstrate how to carry out boolean operations on ΔBDDs and its derived classes of BDDs. In addition to a node (or in fact, a pointer to a node), a boolean function represented in ↑ΔBDDs, is identified by a pair (dx_i, v_i) where dx_i is an integer, and v_i a node. Besides the two successors, $s(v,0)$ and $s(v,1)$, associated with each node v are two integer labels, $d(v,0)$ and $d(v,1)$.

function AND((dx_a, v_a),(dx_b, v_b))
begin
  if $v_a = \mathbf{0}$ **or** $v_b = \mathbf{0}$ **then return** (0,**0**)
  else if $v_a = \mathbf{1}$ **then return** (dx_b, v_b)
  else if $v_b = \mathbf{1}$ **then return** (dx_a, v_a)
  endif

  if $dx_a > dx_b$ **then**
    $res =$ lookup_AND_res($v_a, v_b, dx_a - dx_b$)
  else
    $res =$ lookup_AND_res($v_b, v_a, dx_b - dx_a$)
  endif

  if $res =$ not_found **then**
    $dx_{min} =$ min(dx_1, dx_2)
    $(d0, v0) =$ AND(succ($\mathbf{0}, dx_{min}$,(dx_a, v_a)),
          succ($\mathbf{0}, dx_{min}$,(dx_b, v_b)))
    $(d1, v1) =$ AND(succ($\mathbf{1}, dx_{min}$,(dx_a, v_a)),

```
              succ(1, dx_min, (dx_b, v_b)))
    if (d0, v0)=(d1, v1) then
      if  d0 = 0 then res = (d0, v0)
                 else res = (d0 + dx_min, v0)
      endif
    else
      v = lookup_or_create((d0, v0),(d1, v1))
      res = (dx_min, v)
    endif

    if dx_a > dx_b then
      save_AND_res(v_a, v_b, dx_a-dx_b, res)
    else
      save_AND_res(v_b, v_a, dx_b-dx_a, res)
    endif
  endif

  return res
end
```

Function succ is defined as follows:

$$\texttt{succ}(i, dx_{min}, (dx, v)) = \begin{cases} (dx - dx_{min}, v) & \text{if} \quad dx \neq dx_{min} \\ (d(v,i), s(v,i)) & \text{if} \quad dx = dx_{min} \end{cases}$$

In fact, the function succ is simply the restriction of each argument of AND with respect to the smaller variable between the labels of the two arguments. Figure 5 shows the results of succ when the arguments are $(5, v1)$ and $(2, v2)$. Since the smaller between the two labels is 2, for the restrictions of $(5, v1)$, we simply subtract 2 from 5 and obtain the relativized $(3, v1)$. For $(2, v2)$ itself, we simply follow the left and right branches accordingly.

lookup_or_create($(d0, v0)$,$(d1, v1)$) checks (using a hash table) whether there is a node v such that $d(v, 0) = d0$, $s(v, 0) = v0$, $d(v, 1) = d1$ and $s(v, 1) = v1$. If there is, return the node. If not, create such node (and put it in the hash table). Subroutine save_AND_res caches the results of AND operations while function lookup_AND_res retrieves them.

It is easy to see that the algorithm presented above is based on the Shannon expansion and it is similar to the standard algorithm presented in other BDD literature. The difference here is the scheme to relativize the labels as the execution moves down the graphs and to readjust, whenever necessary, the label of the results returned from the recursive calls.

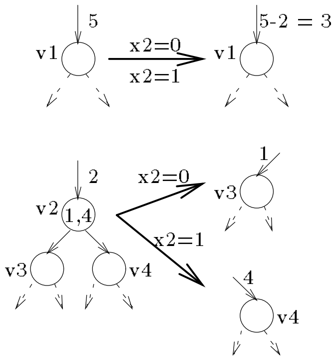

Figure 5: Computation of the function succ

For the complexity of the algorithm, we use, like [Br86], a hash table to avoid multiple evaluations of each tuple $\langle v_a, v_b, d_a - d_b \rangle$, that is, a pair of nodes and the difference of the labels. We also assume a good implementation of the hash tables on which operations, on average, can be done in constant time. Suppose m_a and m_b are the number of nodes of the arguments and n is the number of input variables. We know that there can be only $m_a m_b n$ unique tuples to be evaluated and therefore, the complexity of the algorithm is $O(m_a m_b n)$.

Although the algorithm seems to be of a higher complexity than that of OBDDs which is $O(m_f m_g)$ for the sizes of the arguments, m_f and m_g, it is not in any way a disadvantage. First, recall that the size of the graphs is bigger in OBDDs. Secondly, there is a one-to-one (but not always onto) mapping from each evaluation tuple $\langle v_a, v_b, d_a - d_b \rangle$ to an evaluation pair $\langle v_f, v_g \rangle$ where v_f and v_g are nodes in the OBDD arguments. This means that the number of recursive calls to a boolean operation procedure for $\uparrow\Delta$BDDs is less than or equal to the number of recursive calls to the corresponding procedure for OBDDs which represent the same boolean functions. In essence, switching to $\uparrow\Delta$BDDs will never introduce any unnecessary computation.

Another important operation which can be computed efficiently with ↑ΔBDDs is the general restriction operation. Formally, given a boolean function $f(x_1, \ldots, x_j, \ldots, x_n)$, the restriction operation $x_j \rightarrow c$, when $c \in \{0, 1\}$, yields a function $f(x_1, \ldots, c, \ldots, x_n)$ of $n-1$ arguments. This operation can be carried out in the same way as in the standard OBDDs, namely, by traversing the graph depth first. The index j of the variable, with respect to which we are computing the restriction, will be passed along as an argument to the restrict operation and as the computation moves down the graph, we keep subtracting j with the labels on the edges that we traverse through. Whenever we encounter a subgraph (d_v, v) such that $d_v > j$, we return (d_v, v) as the result. On the other hand, if $d_v = j$, then we follow one of the edges and return $(d(v, c), s(v, c))$. On the way up, we simply combine the result from each branch or readjust the label, just like in the algorithm shown above. Figure 6 depicts a sample computation of the restriction operation.

Note that ↑ΔBDDs coincides with a class of BDDs, called SBDDs with "variable shifter" attributed edges, which is proposed in [MIY90]. However, its potential was not fully recognized and no analysis result was given.

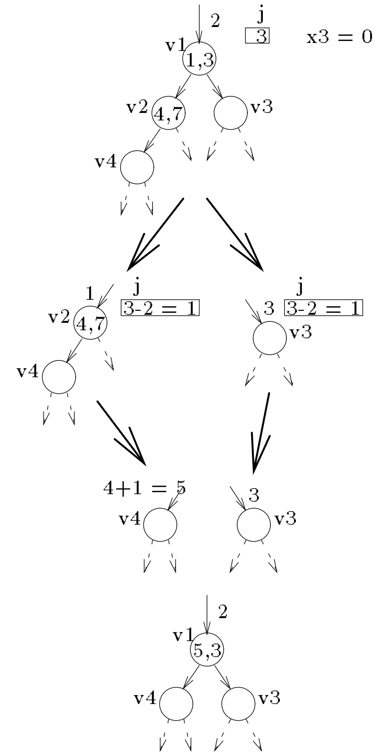

Figure 6: restriction operation

5 Delta Transformation

There are variations of the ↑ transformation which exploit some regularity in the graph in order to reduce the number of nodes. One such variation is the δ transformation.

As demonstrated in Figure 7, the δ transformation takes a structure with at least two integer labels per node, computes the difference between two of the labels, pushes one of the two labels up and keeps the difference. Figure 8 shows a full example of an application of δ to a ↑ΔBDD. Like the ↑ transformation, the δ transformation preserves the same properties: canonicity, efficient manipulation, etc.

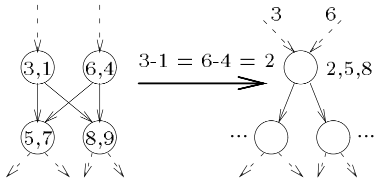

Figure 7: δ transformation

The δ transformation brings up a nice property of δ↑ΔBDDs which may be useful in some situations: all the nodes of OBDDs that have the same successors will always be collapsed into one

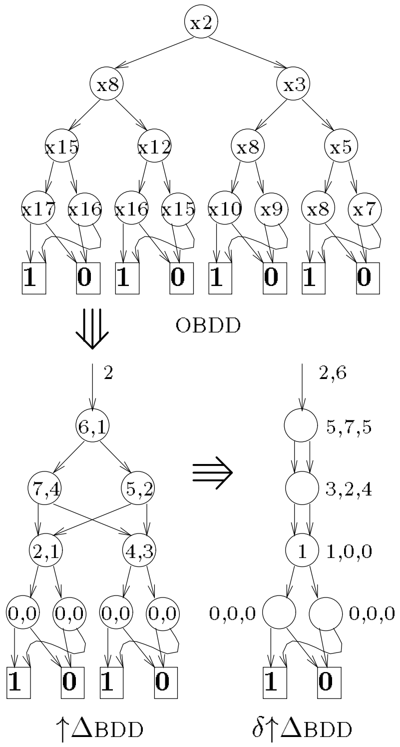

Figure 8: $\delta\uparrow\Delta$BDD vs OBDD

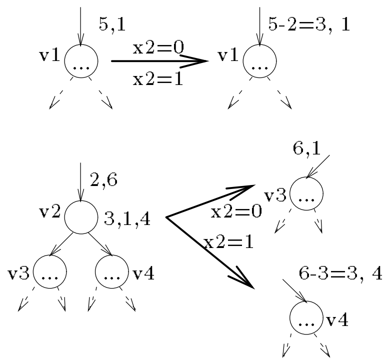

Figure 9: function succ of $\delta\uparrow\Delta$BDDs

node in $\delta\uparrow\Delta$BDDs.

It is easy to modify the algorithms for boolean operations on a class of BDDs, derived by δ transformation. The essential idea is to carry on another integer as we traverse down the graph and add (subtract) back the difference we keep as a label on each node to get back the real original label. Figure 9 demonstrates the key idea with the computations of the `succ` function of $\delta\uparrow\Delta$BDDs.

6 Conclusion

ΔBDDs, and $\uparrow$ as well as δ transformations, provide techniques to reduce the number of nodes, and thus the memory space. They exploit certain regularity found in many applications of BDDs, such as hardware circuit representation. ΔBDDs have the property of potentially exponential reduction in the number of nodes with respect to the size of the OBDD representation of the same functions. Its derived form $\uparrow\Delta$BDDs inherit such property as well as guarantee to have fewer or an equal number of nodes.

Even though, by using these techniques, there is a trade-off between the number of nodes and the number of labels per node, we argue that the gain is more than the loss. This also suggests the use of the transformations repeatedly up to a point where the memory space required to hold the extra labels offsets the memory space saved by the reduction of nodes. In some environments such as an implementation of BDDs package in a machine with long words, the unused bits can be fully utilized to keep the additional labels.

Acknowledgement

We thank Howard Wong-Toi, Tomas Uribe, Henny Sipma, and Nikolaj Bjorner, for carefully reading the drafts of this paper and for their helpful comments.

References

[Ak78] Akers, S. B. "Binary decision diagrams", *IEEE Transactions on Computers*, 1978. C-27, 6(June): 509–516.

[BRB90] Brace, K. S., Rudell, R. L., and Bryant, R. E. "Efficient implementation of a BDD package", *Proc. 27th ACM/IEEE Design Automation Conference*, 1990, June: 40-45.

[Br86] Bryant, R. E. "Graph-based algorithms for boolean function manipulation", *IEEE Transactions on Computers*, 1986. C-35, 6(Aug.): 677–691.

[Br92] Bryant, R. E. "Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams", *ACM Computing Surveys*, Sep. 1992. 24(3):293–318.

[MB88] Madre, J. C. and Billon, J. P. "Proving Circuit Correctness using Formal Comparison Between Expected and Extracted Behaviour", *Proc. 25th ACM/IEEE Design Automation Conference*, 1988, June : 205–210.

[MIY90] Minato, S., Ishiura, N. and Yajima, S. "Shared Binary Decision Diagram with Attributed Edges", *Proc. 27th ACM/IEEE Design Automation Conference*, 1990, June: 52-57.