Binary Decision Diagrams

20-CS-626-001 Formal Verification
Department of Computer Science
University of Cincinnati

1 Introduction

Binary Decision Diagrams (BDD) [1, 8] are a general, graphical representation for arbitrary Boolean functions. Various forms have been put into use, especially for solving VLSI design and verification problems. A canonical form ([3, 4]) has been shown to be quite useful for representing some particular, commonly occurring, Boolean functions. An important advantage of BDDs is that the complexity of binary and unary operations such as existential quantification, oring, anding, among others, is efficient with respect to the size of the BDD operands. Typically, a formula is given as a large collection of BDDs and operations such as those stated above are applied repeatedly to create a single BDD which expresses the models, if any, of the given formula. Intermediate BDDs are created in the process. The problem is that the size of intermediate BDDs may become extraordinarily and impractically large even if the final BDD is small. So, in some applications BDDs are useful and in some they are not.

A Binary Decision Diagram (BDD) is a rooted, directed acyclic graph. A BDD is used to compactly represent the truth table, and therefore complete functional description, of a Boolean function. Vertices of a BDD are called *terminal* if they have no outgoing edges and are called *internal* otherwise. There is one internal vertex, called the *root*, which has no incoming edge. There is at least one terminal vertex, labeled 1, and at most two terminal vertices, labeled 0 and 1. Internal vertices are labeled to represent the variables of the corresponding Boolean function. An internal vertex has exactly two outgoing edges, labeled 1 and 0. The vertices incident to edges outgoing from vertex v are called $then(v)$ and $else(v)$, respectively. Associated with any internal vertex v is an attribute called $index(v)$ which satisfies the properties $index(v) < \min\{index(then(v)), index(else(v))\}$ and $index(v) = index(w)$ if and only if vertices v and w have the same labeling (that is, correspond to the same variable). Thus, the *index* attribute imposes a linear ordering on the variables of a BDD. An example of a formula and one of its BDD representations is given in Figure 1.

Clearly, there is no unique BDD for a given formula. In fact, for the same formula, one BDD might be extraordinarily large and another might be rather compact. It is usually advantageous to use the smallest BDD possible. At least one canonical form of BDD, called *reduced ordered BDD*, does this [3, 4]. The idea is to order the variables of a formula and construct a BDD such that: 1) variables contained in a path from the root to any leaf respect that ordering; and 2) each vertex represents a unique Boolean function. Two Boolean functions are equivalent if their reduced ordered BDDs are isomorphic. A more detailed explanation is given in the next section.

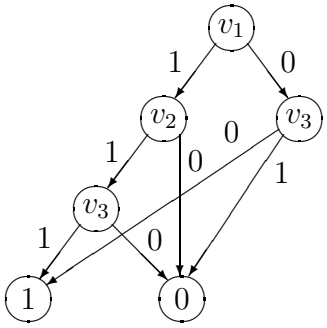


Figure 1: The formula $(v_1 \vee \neg v_3) \wedge (\neg v_1 \vee v_2) \wedge (\neg v_1 \vee \neg v_2 \vee v_3)$ represented as a BDD. The topmost vertex is the root. The two bottom vertices are terminal vertices. Edges are directed from upper vertices to lower vertices. Vertex labels (variable names) are shown inside the vertices. The 0 branch out of a vertex labeled v means v takes the value 0. The 1 branch out of a vertex labeled v means v takes the value 1. The *index* of a vertex is, in this case, the subscript of the variable labeling that vertex.

2 Binary Decision Diagrams

In this section the associated BDD data structure and efficient operations on that data structure are discussed. Attention is restricted to Reduced Ordered Binary Decision Diagrams (ROBDDs) due to its compact, efficient, canonical properties.

A ROBDD is a BDD such that: 1) There is no vertex v such that $then(v) = else(v)$; 2) The subgraphs of two distinct vertices v and w are not isomorphic. A ROBDD represents a Boolean function uniquely in the following way (symbol v will represent both a vertex of a ROBDD and a variable labeling a vertex). Define $f(v)$ recursively as follows:

1. If v is the terminal vertex labeled 0, then $f(v) = 0$;
2. If v is the terminal vertex labeled 1, then $f(v) = 1$;
3. Otherwise, $f(v) = (v \wedge f(then(v))) \vee (\neg v \wedge f(else(v)))$.

Then $f(root(v))$ is the function represented by the ROBDD. A Boolean function has different ROBDD representations, depending on the variable order imposed by *index*, but there is only one ROBDD for each ordering. Thus, ROBDDs are known as a canonical representation of Boolean functions. From now on we will use BDD to refer to a ROBDD.

A data structure representing a BDD consists of an array of `Node` objects (or nodes), each corresponding to a BDD vertex and each containing three elements: a variable label v, $then(v)$, and $else(v)$, where the latter two elements are BDD array indices. The first two nodes in the BDD array correspond to the 0 and 1 terminal vertices of a BDD. For both, $then(..)$ and $else(..)$ are

empty. We will use $terminal(1)$ and $terminal(0)$ to denote the BDD array locations of these nodes. All other nodes fill up the remainder of the BDD array in the order they are created. A node that is not in the BDD array can be created, added to the BDD array, and its array index returned in constant time. We denote this operation by $node = createNode(v, t, e)$, where $node$ is the array index of the newly created node, t is $then(v)$ and e is $else(v)$, both BDD array indices. A hashtable is maintained for the purpose of finding a node's array location given v, t, e. If no such node exists, $createNode$ is called to create and insert it. Otherwise, its array location is returned from the hashtable. We use $lookup(\langle v, t, e\rangle)$ to represent this operation. If $lookup$ returns `null` then the node does not exist, otherwise $lookup$ returns its BDD array location. We use $insert(\langle v, t, e\rangle, node)$ to represent the act of inserting the $node$ BDD array index into the hashtable at key $\langle v, t, e\rangle$. The procedure **findOrCreateNode** for returning the BDD array index of a node is shown in Figure 2.

The main BDD construction operation is to find and attach two descendent nodes ($then(v)$ and $else(v)$) to a parent node (v). The procedure **findOrCreateNode** is used to ensure that no two nodes in the final BDD data structure represent the same function. The procedure for building a BDD data structure is **buildBDD**, shown in Figure 2. It is assumed that variable indices match the value of $index$ applied to that variable (thus, $i = index(v_i)$). The complexity of **buildBDD** is proportional to the number of nodes that must be created. In all interesting applications, many BDDs are constructed. But they may all share the BDD data structure above. Thus, a node may belong to many BDDs.

The operations $\mathbf{reduce}_1$ and $\mathbf{reduce}_0$, shown in Figure 4 will be used to describe several important BDD operations in subsequent sections. Assuming v is the root of the BDD representing f, the operation $\mathbf{reduce}_1(v, f)$ returns f constrained by the assignment of 1 to variable v and $\mathbf{reduce}_0(v, f)$ returns f constrained by the assignment of 0 to the variable v.

Details on performing the common binary operations of $\wedge$ and $\vee$ on BDDs will be ignored here. The reader may refer to [2] for detailed descriptions. We only mention that, using a dynamic programming algorithm, the complexity of these operations is proportional to the product of the sizes of the operands and the size of the result of the operation can be that great as well. Therefore, using $\wedge$ alone, for example (as so many problems would require), could lead to intermediate structures that are too large to be of value. This problem is mitigated somewhat by operations of the kind discussed in the next four subsections, particular existential quantification.

The operations considered next are included not only because they assist BDD oriented solutions but mainly because they can assist search-oriented solutions when used properly. For example, if inputs are expressed as a collection of BDDs, then they may be preprocessed to reveal information that may be exploited later, during search. In particular, inferences may be determined and used to reduce input complexity. The next four sections emphasize this role.

Algorithm 1.

findOrCreateNode (v, t, e)
/* *Input: variable label* v, `Node` *object indices* t *and* e */
/* *Output: an array index of a* `Node` *object* $\langle v, t, e\rangle$ */
If $t == e$ then Return t.
Set $node \leftarrow lookup(\langle v, t, e\rangle)$.
If $node \neq$ `null` then Return $node$.
Set $node \leftarrow createNode(\langle v, t, e\rangle)$.
$insert(\langle v, t, e\rangle, node)$.
Return $node$.
□

Figure 2: Prodcedure for finding a node or creating and inserting it.

Algorithm 2.

buildBDD (f, i)
/* *Input: Boolean function* f, *index* i */
/* *Output: root* `Node` *of BDD representing* f */
If $f \Leftrightarrow 1$ return $terminal(1)$.
If $f \Leftrightarrow 0$ return $terminal(0)$.
Set $t \leftarrow$ **buildBDD**$(f|_{v_i=1}, i+1)$.
Set $e \leftarrow$ **buildBDD**$(f|_{v_i=0}, i+1)$.
Return **findOrCreateNode**(v_i, t, e).
□

Figure 3: Algorithm for building a BDD: invoked using **buildBDD**(f,1).

Algorithm 3.

reduce$_1$ (v, f)
/* *Input: variable* v, *BDD* f */
/* *Output: reduced BDD* */
If $root(f) == v$ then
Return $then(root(f))$.
Return f.
□

reduce$_0$ (v, f)
/* *Input: variable* v, *BDD* f */
/* *Output: reduced BDD* */
If $root(f) == v$ then
Return $else(root(f))$.
Return f.

Figure 4: Operations **reduce**$_1$ and **reduce**$_0$.

Algorithm 4.

exQuant (f, v)
/* *Input: BDD* f*, variable* v */
/* *Output: BDD* f *with* v *existentially quantified away* */
If $root(f) == v$ then Return $then(root(f)) \vee else(root(f))$.
If $index(v) > index(\text{root}(f))$ then Return 0. // If v is not in f do nothing
Set $h_{f_1} \leftarrow$ **exQuant**$(then(root(f)), v)$.
Let $h_{f_0} \leftarrow$ **exQuant**$(else(root(f)), v)$.
If $h_{f_0} == h_{f_1}$ then Return h_{f_1}.
Return **FindOrCreateNode**$(root(f), h_{f_1}, h_{f_0})$.
□

Figure 5: *Algorithm for existentially quantifying variable* v *away from BDD* f*. The* $\vee$ *denotes the "or" of BDDs.*

2.1 Existential Quantification

A Boolean function which can be written

$$f(v, \vec{x}) = (v \wedge h_1(\vec{x})) \vee (\neg v \wedge h_2(\vec{x}))$$

can be replaced by

$$f(\vec{x}) = h_1(\vec{x}) \vee h_2(\vec{x})$$

where $\vec{x}$ is a list of one or more variables. There is a solution to $f(\vec{x})$ if and only if there is a solution to $f(v, \vec{x})$ so it is sufficient to solve $f(\vec{x})$ to get a solution to $f(v, \vec{x})$. Obtaining $f(\vec{x})$ from $f(v, \vec{x})$ is known as *existentially quantifying* v *away from* $f(v, \vec{x})$. This operation is efficiently handled if $f(v, \vec{x})$ is represented by a BDD. However, since most interesting BDD problems are formulated as a conjunction of functions, and therefore as conjunctions of BDDs, existentially quantifying away a variable v succeeds easily only when just one of the input BDDs contains v. Thus, this operation is typically used together with other BDD operations for maximum effectiveness. The algorithm for existential quantification is shown in Figure 5.

If inferences can be revealed in preprocessing they can be applied immediately to reduce input size and therefore reduce search complexity. Although existential quantification can, by itself, uncover inferences (see, for example, Figure 6), those same inferences are revealed during BDD construction if inference lists for each node are built and maintained. Therefore, a more effective use of existential quantification is in support of other operations, such as strengthening (see Section 2.5), to uncover those inferences that cannot be found during BDD construction or in tandem with $\wedge$ to retard the growth of intermediate BDDs.

Existential quantification, if applied as a preprocessing step prior to search, can increase the number of choicepoints expanded per second but can increase the size of the search space. The increase in choicepoint speed is because existentially quantifying a variable away from the function has the same effect

as branching from a choicepoint in both directions. Then overhead is reduced by avoiding heuristic computations. However, search space size may increase since the elimination of a variable can cause subfunctions that had been linked only by that variable to become merged with the result that the distinction between the subfunctions becomes blurred. This is illustrated in Figure 7. The speedup can overcome the lost intelligence but it is sometimes better to turn it off.

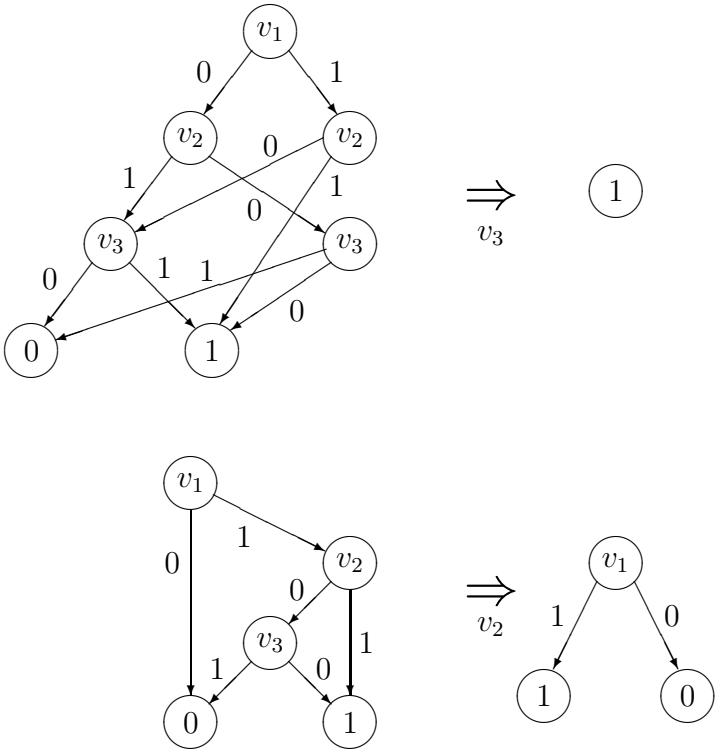


Figure 6: Two examples of existentially quantifying a variable away from a function. Functions are represented as BDDs on the left. Variable v_3 is existentially quantified away from the top BDD leaving 1, meaning that regardless of assignments given to variables v_1 and v_2 there is always an assignment to v_3 which satisfies the function. Variable v_2 is existentially quantified away from the bottom BDD leaving the inference $v_1 = 1$.

2.2 Reductions and Inferences

Consider the truth tables corresponding to two BDDs f and c over the union of variable sets of both f and c. Build a new BDD g with variable set no larger than the union of the variable sets of f and c and with a truth table such that on rows which c maps to 1 g maps to the same value that f maps to, and on other rows g maps to any value, independent of f. It should be clear

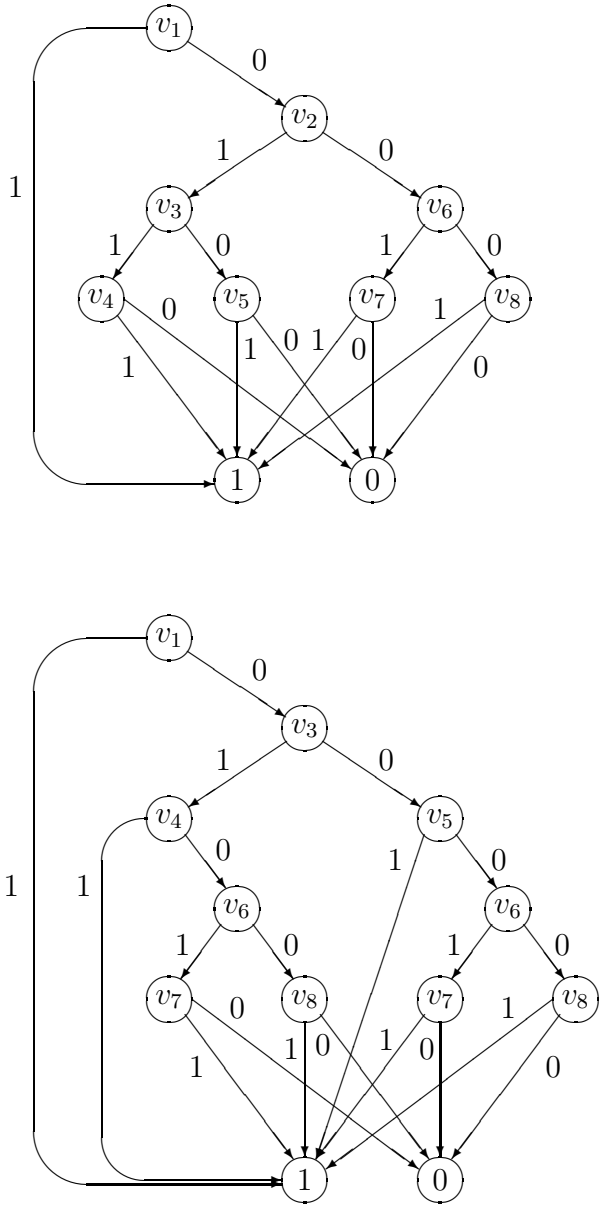


Figure 7: Existential quantification can cause blurring of functional relationships. The top function is seen to separate variables v_6, v_7, and v_8 from v_3, v_4, and v_5 if v_2 is chosen during search first. Existentially quantifying v_2 away from the top function before search results in the bottom function in which no such separation is immediately evident. Without existential quantification the assignment $v_1 = 0$, $v_2 = 1$, $v_3 = 1$ reveals the inference $v_4 = 1$. With existential quantification the assignment must be augmented with $v_7 = 0$ and $v_8 = 0$ (but v_2 is no longer necessary) to get the same inference.

that $f \wedge c$ and $g \wedge c$ are identical so g can replace f in a collection of BDDs without changing its solution space.

There are at least three reasons why this might be done. The superficial reason is that g can be made smaller than f. A more important reason is that inferences can be discovered. The third reason is that BDDs can be removed from the collection without loss. Consider, for example, BDDs representing functions

$$f = (v_1 \vee \neg v_2 \vee v_3) \wedge (\neg v_1 \vee v_2) \text{ and}$$
$$c = (v_1 \vee \neg v_2).$$

Let a truth table row be represented by a 0-1 vector which reflects assignments of variables indexed in increasing order from left to right. Let g have the same truth table as f except for row $\langle 011 \rangle$ which c maps to 0 and g maps to 1. Then $g = (v_1 \leftrightarrow v_2)$ and $f \wedge c$ is the same as $g \wedge c$ but g is smaller than f. As an example of discovering inferences consider

$$f = (v_1 \rightarrow v_2) \wedge (\neg v_1 \rightarrow (\neg v_3 \wedge v_4)) \text{ and}$$
$$c = (v_1 \vee v_3).$$

Let g have the same truth table as f except g maps rows $\langle 0001 \rangle$ and $\langle 0101 \rangle$ to 0, as does c. Then $g = (v_1) \wedge (v_2)$ which reveals two inferences. The BDDs for f, c, and g of this example are shown in Figure 9. The example showing BDD elimination is deferred to Theorem 1, Section 2.4.

Clearly, there are numerous strategies for creating g from f and c and replacing f with g. An obvious one is to have g and c map the same rows to 0. This strategy, which we will call zero-restrict, turns out to have weaknesses. Its obvious dual, which has g map to 1 all rows that c maps to 0, is no better. For example, applying zero-restrict to f and c of Figure 11 produces $g = \neg v_3 \wedge (v_1 \vee (\neg v_1 \wedge \neg v_2))$ instead of the inference $g = \neg v_3$ which is obtained from a more intelligent replacement strategy. The alternative approach, one of many possible ones, judiciously chooses some rows of g to map to 1 and others to map to 0 so that g's truth table reflects a logic pattern that generates inferences. The truth table of c has many 0 rows and this is exploited. Specifically, c maps rows $\langle 010 \rangle$, $\langle 011 \rangle$, $\langle 101 \rangle$, and $\langle 111 \rangle$ to 0. The more intelligent strategy lets g map rows $\langle 011 \rangle$ and $\langle 111 \rangle$ to 0 and rows $\langle 010 \rangle$ and $\langle 101 \rangle$ to 1. Then $g = \neg v_3$.

Improved replacement strategies might target particular truth table patterns, for example equivalences, or they might aim for inference discovery. Since there is more freedom to manipulate g if the truth table of c has many zeros, it is important to choose c as carefully as the replacement strategy. This is illustrated by the examples of Figure 10 and Figure 11 where, in the first case, no inference is generated but after f and c are swapped an inference is generated. The next two sections show two replacement strategies that are among the more commonly used.

2.3 Restrict

The original version of restrict is what we call zero-restrict above. That is, the original version of restrict is intended to remove paths to $terminal(1)$

Algorithm 5.

restrict (f, c)
/* *Input: BDD f, BDD c* */
/* *Output: BDD f restricted by c* */
If c or f is $terminal(1)$ or if f is $terminal(0)$ return f.
If $c == \neg f$ return $terminal(0)$.
If $c == f$ return $terminal(1)$.
// f and c have a non-trivial relationship
Set $v_f \leftarrow root(f)$. // v_f is a variable
Set $v_c \leftarrow root(c)$. // v_c is a variable
If $index(v_f) > index(v_c)$ return **restrict**(f, **exQuant**(c, v_c)).
If **reduce**$_0(v_f, c)$ is $terminal(0)$ then
    Return **restrict**(**reduce**$_1(v_f, f)$, **reduce**$_1(v_f, c)$).
If **reduce**$_1(v_f, c)$ is $terminal(0)$ then
    Return **restrict**(**reduce**$_0(v_f, f)$, **reduce**$_0(v_f, c)$).
Set $h_{f_1} \leftarrow$ **restrict**(**reduce**$_1(v_f, f)$, **reduce**$_1(v_f, c)$).
Set $h_{f_0} \leftarrow$ **restrict**(**reduce**$_0(v_f, f)$, **reduce**$_0(v_f, c)$).
If $h_{f_1} == h_{f_0}$ then Return h_{f_1}.
Return **findOrCreateNode**(v_f, h_{f_1}, h_{f_0}).
□

Figure 8: Algorithm for restricting a BDD f by a BDD c.

from f that are made irrelevant by c. The idea was introduced in [5]. In this monograph we present our own version which is implemented as Algorithm 5 of Figure 8. We use symbol to denote the restrict operator. Then $g = f \ \ c$ is the result of zero-restrict after all variables in c that are not in f are existentially quantified away from c. Figures 9 to 11 show examples that were referenced in the previous section.

Procedure **restrict** is similar to a procedure called generalized co-factor (**gcf**) or constrain (see Section 2.4 for a description). Both **restrict**(f, c) and **gcf**(f, c) agree with f *on interpretations where c is satisfied*, but are generally somehow simpler than f. Procedure **restrict** can be useful in preprocessing because the BDDs produced from it can never be larger than the BDDs they replace.

On the negative side, it can, in odd cases, cause a garbling of local information. Although **restrict** may reveal some of the inferences that strengthening would (see below), it can still cause the number of search choicepoints to increase. Both these issues are related: **restrict** can spread an inference that is evident in one BDD over multiple BDDs (see Figure 12 for an example).

$f = (v_1 \rightarrow v_2) \wedge (\neg v_1 \rightarrow (\neg v_3 \wedge v_4))$

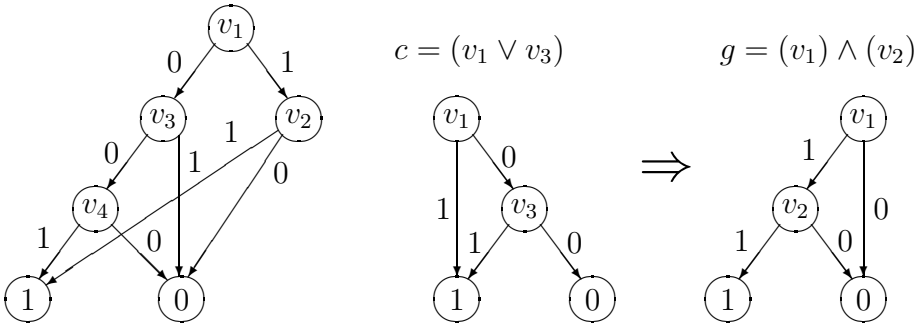


Figure 9: A call to **restrict**(f, c) returns the BDD g shown on the right. In this case inferences $v_1 = 1$ and $v_2 = 1$ are revealed. The symbol $\Rightarrow$ denotes the operation.

$f = (v_1 \vee \neg v_2) \wedge (\neg v_1 \vee \neg v_3)$ $\quad c = (v_2 \vee \neg v_3)$

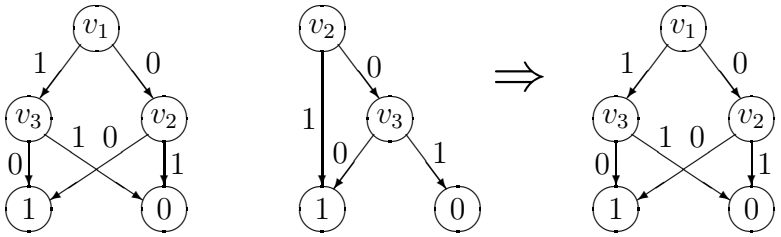


Figure 10: A call to **restrict**(f, c) results in no change.

$f = (v_2 \vee \neg v_3)$ $\quad c = (v_1 \vee \neg v_2) \wedge (\neg v_1 \vee \neg v_3)$ $\quad g = (\neg v_3)$

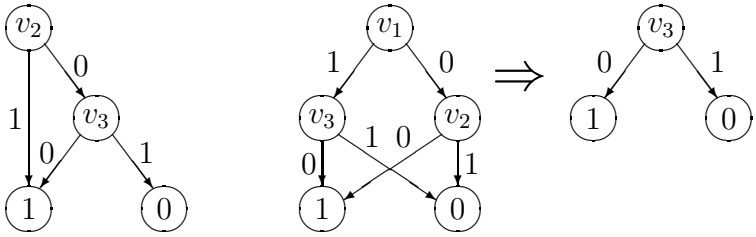


Figure 11: Reversing the roles of f and c in Figure 10, a call to **restrict**(f, c) results in the inference $g = \neg v_3$ as shown on the right. In this case, the large number of 0 truth table rows for c was exploited to advantage.

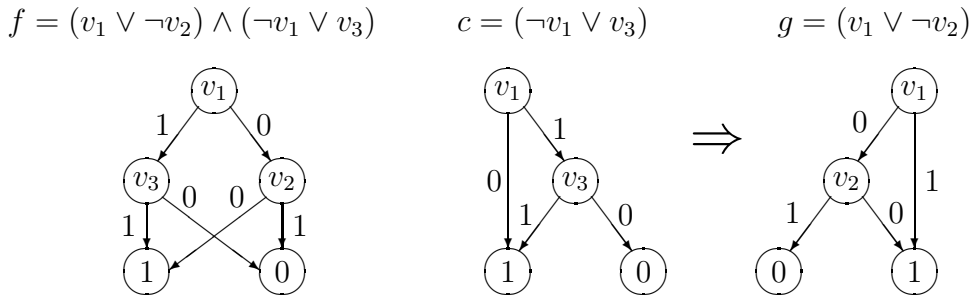


Figure 12: A call to **restrict**(f, c) spreads an inference that is evident in one BDD over multiple BDDs. If v_3 is assigned 0 in f then $v_1 = 0$ and $v_2 = 0$ are inferred. After replacing f with $g =$ **restrict**(f, c), to get the inference $v_2 = 0$ from the choice $v_3 = 0$, visit c to get $v_1 = 0$ and *then* g to get $v_2 = 0$. Thus, restrict can increase work if not used properly. In this case, restricting in the reverse direction leads to a better result.

2.4 Generalized Co-factor

The *generalized co-factor* operation, also known as *constrain*, is denoted here by $|$ and implemented as **gcf** in Figure 13 as Algorithm 6. It takes BDDs f and c as input and produces $g = f|c$ by *sibling substitution*. BDD g may be larger or smaller than f but, more importantly, systematic use of this operation can result in the elimination of BDDs from a collection. Unfortunately, by definition, the result of this operation depends on the underlying BDD variable ordering so it cannot be regarded as a logical operation. It was introduced in [6].

BDD g is a generalized co-factor of f and c if for any truth assignment t, $g(t)$ has the same value as $f(t')$ where t' is the "nearest" truth assignment to t that maps c to 1. The notion of "nearest" truth assignment depends on a permutation π of the numbers $1, 2, \ldots, n$ which states the variable ordering of the input BDDs. Represent a truth assignment to n variables as a vector in $\{0,1\}^n$ and, for truth assignment t, let t_i denote the i^{th} bit of the vector representing t. Then the distance between two truth assignments t' and t'' is defined as $\sum_{i=1}^{n} 2^{n-i}(t'_{\pi_i} \oplus t''_{\pi_i})$. One pair of assignments is nearer to each other than another pair if the distance between that pair is less. It should be evident that distances between pairs are unique for each pair.

For example, Figure 14 shows BDDs f and c under the variable ordering given by $\pi = \langle 1, 2, 3, 4\rangle$. For assignment vectors $\langle **01\rangle$, $\langle **10\rangle$, $\langle **11\rangle$ (where $*$ is a wildcard meaning 0 or 1), **gcf**(f, c), shown as the BDD at the bottom of Figure 14, agrees with f since those assignments cause c to evaluate to 1. The closest assignment to $\langle 0000\rangle$, $\langle 0100\rangle$, $\langle 1000\rangle$, and $\langle 1100\rangle$ causing c to evaluate to 1 is $\langle 0001\rangle$. $\langle 0101\rangle$, $\langle 1001\rangle$, and $\langle 1101\rangle$, respectively. On all these inputs **gcf**(f, c) has value 1, which the reader can check in Figure 14.

The following expresses the main property of $|$ that makes it useful.

Theorem 1. *Given BDDs $f_1, ..., f_k$, for any $1 \leq i \leq k$, $f_1 \wedge f_2 \wedge ... \wedge f_k$ is satisfiable if and only if $(f_1|f_i) \wedge ... \wedge (f_{i-1}|f_i) \wedge (f_{i+1}|f_i) \wedge ... \wedge (f_k|f_i)$ is*

Algorithm 6.

gcf (f, c)
/* *Input: BDD* f*, BDD* c */
/* *Output: greatest co-factor of* f *by* c */
If $f == terminal(0)$ or $c == terminal(0)$ return $terminal(0)$.
If $c == terminal(1)$ or $f == terminal(1)$ return f.
Set $v_m \leftarrow index^{-1}(\min\{index(root(c)), index(root(f))\})$.
// v_m is the top variable of f and c
If $\mathbf{reduce}_0(v_m, c) == terminal(0)$ then
Return $\mathbf{gcf}(\mathbf{reduce}_1(v_m, f), \mathbf{reduce}_1(v_m, c))$.
If $\mathbf{reduce}_1(v_m, c) == terminal(0)$ then
Return $\mathbf{gcf}(\mathbf{reduce}_0(v_c, f), \mathbf{reduce}_0(v_c, c))$.
Set $h_1 \leftarrow$ $\mathbf{gcf}(\mathbf{reduce}_1(v_m, f), \mathbf{reduce}_1(v_m, c))$.
Set $h_0 \leftarrow$ $\mathbf{gcf}(\mathbf{reduce}_0(v_m, f), \mathbf{reduce}_0(v_m, c))$.
If $h_1 == h_0$ then Return h_1.
Return $\mathbf{FindOrCreateNode}(v_m, h_1, h_0)$.
□

Figure 13: Algorithm for finding a greatest common co-factor of a BDD.

satisfiable. Moreover, any assignment satisfying the latter can be mapped to an assignment that satisfies $f_1 \wedge ... \wedge f_k$.

Proof. If we show

$$(f_1|f_i) \wedge \ldots \wedge (f_{i-1}|f_i) \wedge (f_{i+1}|f_i) \wedge \ldots \wedge (f_k|f_i) \tag{1}$$

is satisfiable if and only if

$$(f_1|f_i) \wedge \cdots \wedge (f_{i-1}|f_i) \wedge (f_{i+1}|f_i) \wedge \ldots \wedge (f_k|f_i) \wedge f_i \tag{2}$$

is satisfiable then, since (2) is equivalent to $f_1 \wedge \ldots \wedge f_k$, the first part of the theorem will be proved. Suppose (2) is satisfied by truth assignment t. That t represents a truth table row that f_i maps to 1. Clearly that assignment also satisfies (1). Suppose no assignment satisfies (2). Then all assignments for which f_i maps to 1 do not satisfy (1) since otherwise (2) would be satisified by any that do. We only need to consider truth assignments t which f_i maps to 0. Each $(f_j|f_i)$ in (1) and (2) maps to the same value that f_j maps the "nearest" truth assignment, say r, to t that satisfies f_i. But r cannot satisfy (2) because it cannot satisfy (1) by the argument above. Hence, there is no truth assignment falsifying f_i but satisfying (1) so the first part is proved.

For the second part, observe that any truth assignment that satisfies (1) and (2) also satifies $f_i \wedge \ldots \wedge f_k$ so we only need to consider assignments t that satisfy (1) but not (2). In that case, by construction of $(f_j|f_i)$, the assignment that is "nearest" to t and satisfies f_i also satisfies $(f_j|f_i)$. That assignment satisfies $f_1 \wedge \ldots \wedge f_k$. □

This means that, for the purposes of a solver, generalized co-factoring can be used to eliminate one of the BDDs among a given conjoined set of BDDs: the solver finds an assignment satisfying $\mathbf{gcf}(f_1, f_i) \wedge ... \wedge \mathbf{gcf}(f_k, f_i)$ and then extends the assignment to satisfy f_i, otherwise the solver reports that the instance has no solution. However, unlike **restrict**, generalized co-factoring cannot by itself reduce the number of variables in a given collection of BDDs. Other properties of the **gcf** operation, all of which are easy to show, are:

1. $f = c \wedge \mathbf{gcf}(f, c) \vee \neg c \wedge \mathbf{gcf}(f, \neg c)$.
2. $\mathbf{gcf}(\mathbf{gcf}(f, g), c) = \mathbf{gcf}(f, g \wedge c)$.
3. $\mathbf{gcf}(f \wedge g, c) = \mathbf{gcf}(f, c) \wedge \mathbf{gcf}(g, c)$.
4. $\mathbf{gcf}(f \wedge c, c) = \mathbf{gcf}(f, c)$.
5. $\mathbf{gcf}(f \wedge g, c) = \mathbf{gcf}(f, c) \wedge \mathbf{gcf}(g, c)$.
6. $\mathbf{gcf}(f \vee g, c) = \mathbf{gcf}(f, c) \vee \mathbf{gcf}(g, c)$.
7. $\mathbf{gcf}(f \vee \neg c, c) = \mathbf{gcf}(f, c)$.
8. $\mathbf{gcf}(\neg f, c) = \neg\, \mathbf{gcf}(f, c)$.
9. If c and f have no variables in common and c is satisfiable then $\mathbf{gcf}(f, c) = f$.

Care must be taken when co-factoring in "both" directions (exchanging f for c). For example, $f \wedge g \wedge h$ cannot be replaced by $(g|f) \wedge (f|g) \wedge h$ since the former may be unsatisfiable when the latter is satisfiable.

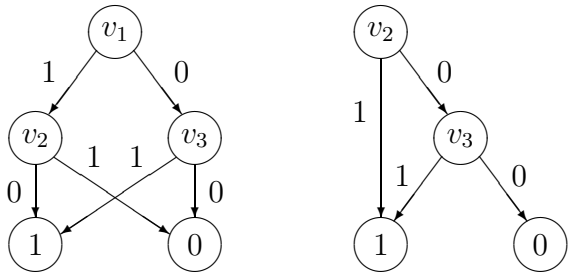
Examples of the application of **gcf** are shown in Figures 14 and 15. Figure 14 illustrates the possibility of increasing BDD size. Figure 15 presents the same example after swapping v_1 and v_3 under the same variable ordering and shows that the result produced by **gcf** is sensitive to variable ordering. Observe that the functions produced by **gcf** in both figures have different values under the assignment $v_1 = 1$, $v_2 = 1$, and $v_3 = 0$. Thus, the function returned by **gcf** depends on the variable ordering as well.

2.5 Strengthen

This binary operation on BDDs helps reveal inferences that are missed by **restrict** due to its sensitivity to variable ordering. Given two BDDs, b_1 and b_2, strengthening conjoins b_1 with the *projection* of b_2 onto the variables of b_1: that is, $b_1 \wedge \exists \vec{v} b_2$, where $\vec{v}$ is the set of variables appearing in b_2 but not in b_1. Strengthening each b_i against all other b_js sometimes reveals additional inferences or equivalences. Algorithm **strengthen** is shown in Figure 16. Figure 17 shows an example.

Strengthening provides a way to pass important information from one BDD to another without causing a size explosion. No size explosion can occur

$f = (v_1 \rightarrow \neg v_2) \vee (\neg v_1 \rightarrow v_3)$ $\quad$ $c = (v_2 \vee v_3)$



$gcf(f, c) = (v_1 \rightarrow \neg v_2) \vee (\neg v_1 \rightarrow (v_2 \rightarrow v_3))$

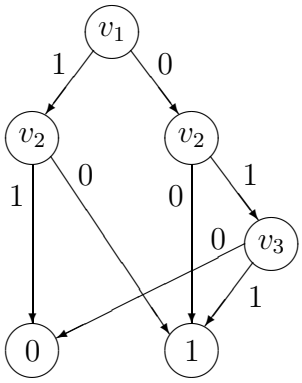


Figure 14: Generalized co-factor operation on f and c as shown. In this case the result is more complicated than f. The variable ordering is $v_1 < v_2 < v_3$.

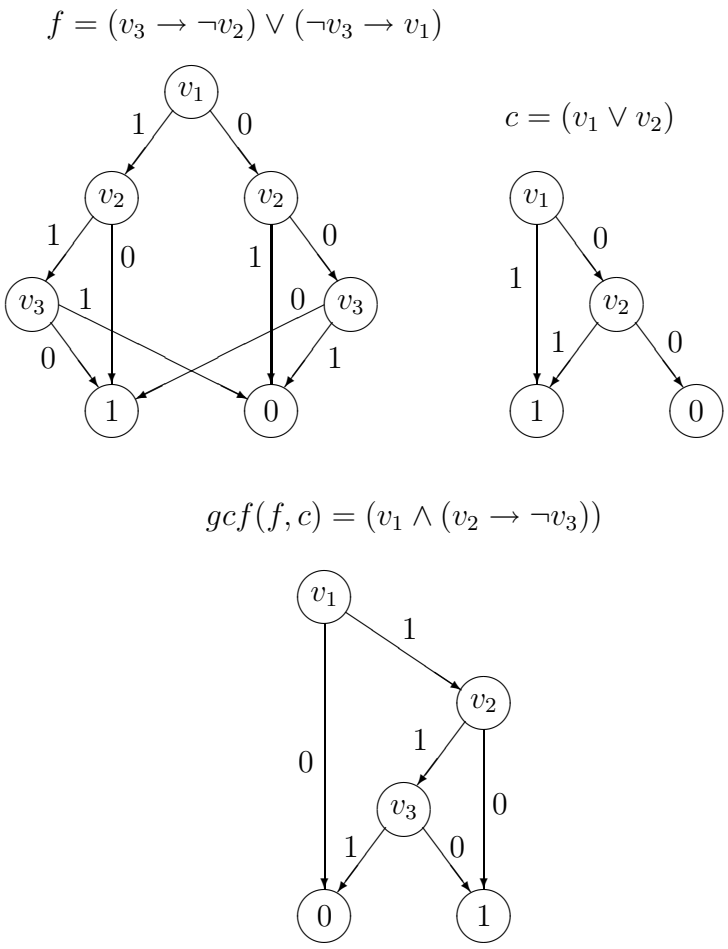


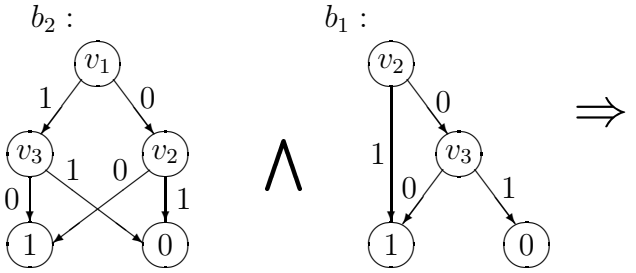
Figure 15: Generalized co-factor operation on the same f and c of Figure 14 and with the same variable ordering but with v_1 and v_3 swapped. In this case the result is less complicated than f and the assignment $\{v_1, v_2\}$ causes the output of **gcf** in this figure to have value 1 whereas the output of **gcf** in Figure 14 has value 0 under the same assignment.

Algorithm 7.

Strengthen (b_1, b_2)
/* *Input: BDD* b_1*, BDD* b_2 */
/* *Output: BDD* b_1 *strengthened by* b_2 */
Set $\vec{x} \leftarrow \{x : x \in b_2, x \notin b_1\}$.
Repeat the following for all $x \in \vec{x}$:
Set $b_2 \leftarrow$ **exQuant**(b_2, x).
Return $b_1 \wedge b_2$.
□

Figure 16: Algorithm for strengthening a BDD by another.

because, before b_1 is conjoined with b_2, all variables in b_2 that don't occur in b_1 are existentially quantified away. If an inference (of the form $v = 1$, $v = 0$, $v = w$, or $v = \neg w$) exists due to just two BDDs, then strengthening those BDDs against each other (pairwise) can *move* those inferences, even if originally spread across both BDDs, to one of the BDDs. Because **strengthen** shares information between BDDs, it can be thought of as sharing intelligence and *strengthening* the relationships between functions; the added intelligence in these strengthened functions can be exploited by a smart search heuristic. We have found that **strengthen** usually decreases the number of choicepoints when a particular search heuristic is employed, but sometimes it causes more choicepoints. We believe this is due to the delicate nature of some problems where duplicating information in the BDDs leads the heuristic astray.



Strengthening example: Existentially quantify v_1 away from b_2...

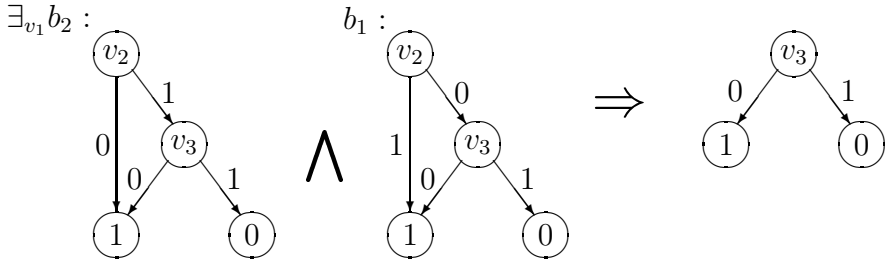


Figure 17: ...then conjoin the two BDDs. Inference $v_3 = 0$ is revealed.

Procedure **strengthen** may be applied to CNF formulas and in this case it is the same as applying Davis-Putnam resolution selectively on some of the clauses. When used on more complex functions it is clearer how to use it effectively as the clauses being resolved are grouped with some meaning. Evidence for this comes from Bounded Model Checking examples.

We close this section by mentioning that for some classes of problems resolution has polynomial complexity while strictly BDD manipulations require exponential time and for other classes of problems resolution has exponential complexity while BDD manipulations require polynomial time.

References

[1] S.B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, **C-27**:509–516, 1978.

[2] H. Andersen. An Introduction to Binary Decision Diagrams. Technical report, Department of Information Technology, Technical University of Denmark, 1998.

[3] R.E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Trans. on Computers*, **C-35**:677–691, 1986.

[4] R.E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, **24**:293–318, 1992.

[5] O. Coudert, C. Berthet and J.C. Madre. Verification of synchronous sequential machines based on symbolic execution. *Lecture Notes in Computer Science*, **407**:365–373, Springer, 1990.

[6] O. Coudert, and J.C. Madre. A unified framework for the formal verification of sequential circuits. In *Proceedings of the 1990 International Conference on Computer-Aided Design* (ICCAD '90), 126–129, IEEE Computer Society Press, Los Alamitos, CA, 1990.

[7] O. Coudert. On solving covering problems. In *Proceedings of the* 33^{rd} *Design Automation Conference*, 197–202, IEEE Computer Society Press, Los Alimitos, 1996.

[8] C.Y. Lee. Representation of switching circuits by binary decision programs. *Bell Systems Technical Journal*, **38**:985–999, 1959.