Morehead Electronic Journal of Applicable Mathematics
Issue 3 — CS-2002-02
Copyright ©2003

A CNF Analogue to Strengthening

Sean Weaver[1]
University of Cincinnati, Cincinnati, Ohio, 45215-0030

Abstract

We introduce a new BDD binary operation called strengthening and show that its effect can be completely characterized in terms of the well known clausal operations of resolution and subsumption.

1 Introduction

We are interested in large Boolean functions which are expressed as conjunctions (logical "and") of Boolean functions, each involving a small, fixed number k of input variables. Typically k is less than 20. The question of determining whether some assignment of values (*1* or *0* corresponding to *true* or *false*, respectively) to input variables causes a given large Boolean function to evaluate to *1* is called the Satisfiability problem (or SAT for short). Instances of SAT appear quite often in practice. For example, to determine whether a VLSI design matches its specification both the design and specification are expressed as Boolean functions D and S and then they are checked for equivalence, which means they are checked to see whether there is no assignment which causes both D and S to evaluate to different values. Both D and S can be quite large but typically are expressible as conjunctions of small functions where 1) each small function represents the equivalence of the function corresponding to an electronic logic "gate" and a newly created variable representing its output; 2) each logic gate has a small number of inputs; and 3) an input to one logic gate can be an output of another.

Large Boolean functions such as those described above are often expressed as collections of Binary Decision Diagrams. A Binary Decision Diagram is a rooted directed acyclic graph with nodes labeled by names of input variables, except for two special nodes labeled *0* and *1*. All nodes, except for the two special nodes, have two outgoing edges, one labeled *1* and one labeled *0*. The special nodes have no outgoing edges. A Binary Decision Diagram specifies the truth table for a corresponding Boolean function f: that is, the mapping from an assignment of input values to *0* or *1*. A path from root to the *1* node specifies a cylinder of assignments of values to inputs which causes f to evaluate to *1*:

[1]This work was done by Sean Weaver as part of an undergraduate senior project where Dr. John Franco was the advisor.

Keywords: Satisfiability, CNF, DNF, BDD, Resolution, Subsumption, Strengthening, Boolean

namely, for each node in the path, assign to the variable labeling the node the value indicated by the outward directed edge from that node on the path (all assignments of other variables in conjunction with that specified by the path causes f to evaluate to *1*). Similarly, a path from the root to the *0* node specifies a cyclinder of assignments which causes f to evaluate to *0*. We are concerned with a particular form of Binary Decision Diagrams called Reduced Ordered Binary Decision Diagrams [1]. In the following we will use the term BDD to mean Reduced Ordered Binary Decision Diagram. This form is described in detail in section 3.

To determine the satisfiability of a large Boolean function, it is common practice to first translate the function into Conjunctive Normal Form (CNF): that is, as a conjunction of disjunctions of literals. A literal may be positive (represented as a variable, say v) or negative (represented as the variable's complement, say $\neg v$). If v has value *1* then $\neg v$ has value *0* and if v has value *0* then $\neg v$ has value *1*. A disjunction of literals is called a *clause*: a clause has value *1* if and only if at least one of its literals has value *1*. The logical connective $\vee$ is used to delimit literals of a clause. As an example, $(v_1 \vee \neg v_2 \vee v_3)$ represents a clause. This clause has value *1* if v_2 has value *0* or either v_1 or v_3 have value *1*. A CNF expression is a conjunction of clauses: the expression has value *1* if and only if all its clauses have value *1*. The logical connective $\wedge$ is used to delimit clauses of an expression. As an example, $(v_1 \vee \neg v_2 \vee v_3) \wedge (\neg v_1 \vee v_3 \vee v_4) \wedge (\neg v_2 \vee \neg v_3 \vee \neg v_4)$ is a CNF expression which has value *1* if v_1 and v_3 have value *1* and v_2 has value *0* and has value *0* if all variables have value *1*. For the rest of the paper we will represent CNF expressions in the following way:

Definition 1 CNF Representation*: Express a CNF expression as a set of sets of literals.*

Example 2 *The CNF expression*

$$(v_1 \vee \neg v_2 \vee v_3) \wedge (\neg v_1 \vee v_4) \wedge (v_2 \vee \neg v_4)$$

is expressed as

$$\{\{v_1, \neg v_2, v_3\}, \{\neg v_1, v_4\}, \{v_2, \neg v_4\}\}.$$

$\square$

The Satisfiability problem is well known to be hard, in general. Solutions may be found significantly faster depending on how the given function is represented, whether as a collection of BDDs or a CNF expression, and what operations are applied to that representation. In (Franco et. al., 2002) a system, called SBSAT, for solving SAT given large Boolean functions expressed as a collection of BDDs is proposed. Unlike other systems (see, for example, [4, 5, 6, 7]), SBSAT does not translate the BDDs to CNF before searching for a solution because information that is important to reducing the search time may become garbled during the translation. Instead, the collection of BDDs is pre-processed using some new, efficient BDD operations, creating a reduced and simplified collection of BDDs to which search is applied. If the input is translated to a CNF expression, a number of primitive clausal operations could be applied to reduce the clause set. Among these are resolution and subsumption. The question we ask is whether the new BDD operations can be characterized easily in terms of clausal operations on CNF counterparts. The answer will help determine the power of SBSAT relative to existing systems. In this paper we study one of the new operations called strengthening. We find that strengthening a pair of BDDs with respect to each other is the same as resolving and subsuming clauses sets corresponding to the BDDs in a particular way. Details are given in subsequent sections.

2 Resolution and Subsumption

There are two principal operators used to manipulate a set of clauses: resolution and subsumption. Resolution is a binary operation and results in the generation of a clause that did not previously exist in the current clause set. Each generated clause is implied by the given expression. Therefore, if the empty clause is generated, the given expression can never have value *1*. On the other hand, if the empty clause cannot be generated, there is an assignment of variable values which causes the expression to evaluate to *1*. Two clauses are said to conflict in variable v if literal v is in one clause and literal $\neg v$ is in the other. Two clauses can be resolved if they conflict in exactly one variable. A resolution step may be performed on two clauses that can be resolved. The result is a clause called the *resolvent* which contains all the literals of both clauses except the conflicting ones.

Definition 3 Resolution Step*: Given two clauses c_1 and c_2, one of which contains literal v, the other literal $\neg v$, and such that there is no other literal that appears in one clause and whose complement appears in the other. The result of a* resolution step *on c_1 and c_2 with v as pivot is a clause containing all literals in c_1 and c_2 except v and $\neg v$. That is, $c_1 \cup c_2 \setminus \{v, \neg v\}$.*

For example, the clauses

$$\{v_1, \neg v_2, v_3\}, \{v_2, v_3, \neg v_4\}$$

can resolve on v_2 and their resolvent (the result of a single resolution step) is

$$\{v_1, v_3, \neg v_4\}.$$

Subsumption is a binary operation that results in the elimination of a clause from the clause set. The idea is to remove a clause which is implied by another. Specifically, if a subset of the literals of a clause c are exactly those literals of another clause, then c may be removed from the clause set.

We apply the Davis-Putnam Procedure [2], originally designed to determine satisfiability of a set of clauses, to simplify a set of clauses. One step of the

Davis-Putnam Procedure is called a Davis-Putnam step and is defined as follows:

Definition 4 Davis-Putnam Step*: Given a propositional expression ϕ_v in conjunctive normal form containing at least one clause with either literal v or literal $\neg v$. The result of a* Davis-Putnam step *on ϕ_v with respect to v is ϕ_v plus all clauses resulting from all possible resolution steps on pairs of clauses in ϕ_v and minus all clauses which contain v and all which contain $\neg v$. We use $DP_v(\phi_v)$ to denote the result of a Davis-Putnam step on expression ϕ_v with respect to v.*

□

From Definition 4

$$\begin{aligned} &DP_v(\phi_v) = \qquad (1)\\ &\quad \{c_1 \cup c_2 \setminus \{v, \neg v\} : c_1, c_2 \in \phi_v, v \in c_1, \neg v \in c_2, \nexists w \neq v : w, \neg w \in c_1 \cup c_2\}\\ &\quad \cup\, \phi_v \setminus \{c : c \in \phi_v, v \in c \text{ or } \neg v \in c\}. \end{aligned}$$

The implementation is done iteratively and it does not matter the order in which you choose the clauses with which you use to resolve.

Example 5

Given*:* $\{\underline{\{v_1, v_2\}}, \{v_1, v_3\}, \{v_1, \neg v_2, v_3\}, \underline{\{\neg v_1, v_2, v_4\}}\}$

Generate resolvent: $\{v_2, v_4\}$ *from the resolution of underlined clauses.*

Leaving*:* $\{\{v_1, v_2\}, \underline{\{v_1, v_3\}}, \{v_1, \neg v_2, v_3\}, \underline{\{\neg v_1, v_2, v_4\}}, \{v_2, v_4\}\}$

Generate resolvent: $\{v_2, v_3, v_4\}$ *from the resolution of underlined clauses.*

Leaving*:* $\{\{v_1, v_2\}, \{v_1, v_3\}, \{v_1, \neg v_2, v_3\}, \{\neg v_1, v_2, v_4\}, \{v_2, v_4\}, \{v_2, v_3, v_4\}\}$

All possible resolutions have been done involving v_1.

Removing all clauses that contain v_1.

Output*:* $\{\{v_2, v_4\}, \{v_2, v_3, v_4\}\}$

□

The Davis-Putnam Procedure is the result of applying Davis-Putnam steps iteratively using a list of variables. Each iterative step can be applied in any order and will always yield the same result. If the list of variables contains every variable occurring in the given function then the Davis-Putnam Procedure can be used to determine the satisfiability of the function. However, this approach is too inefficient to be useful. Instead we will apply the procedure to groups of selected subsets of clauses and variables. In fact, each clause subset we will be interested in expresses a small BDD. We can do this because there is a close relationship between BDDs and clause sets. First we describe BDDs in more detail and then state this relationship more precisely.

3 BDDs

Let $V = \{v_1, v_2, \ldots\}$ be a set of Boolean variables where indices have been assigned arbitrarily. We say variable v_i is of lower order than variable v_j if and only if $i < j$. A Boolean function is a mapping from assignments of values to variables of V to $\{0, 1\}$. As stated, a BDD specifies a Boolean function. BDDs are constructed by recursively using the form *if v then a else b* (referred to below as ITE) where v is a variable[2] from V and a and b are either Boolean functions (subfunctions) or the constant *1* (for *true*), or the constant *0* (for *false*) with the following restrictions: (i) variable v is of lower order than any of the variables in a or b; and (ii) a and b are not logically equivalent. Each node of a BDD corresponds either to an ITE, in which case it is labeled v, the conditional variable of the ITE expression, or one of the constants *1* or *0*, in which case it is labeled *1* or *0*, respectively. By (ii) and the fact that two logically equivalent BDDs are identical, there is exactly one node labeled *1* and exactly one labeled *0*. The root node corresponds to the highest level ITE in the recursion. Two edges directed away from a node are incident to the nodes representing the *then* and *else* subfunctions (a and b) of the ITE. The edge leading to subfunction a is labeled *1* and the other edge is labeled *0*. We speak of a path in a BDD as a traversal of edges from root to a leaf (a terminal node labeled *1* or *0*). By (i) no variable label is encountered more than once while traversing a path. As stated earlier, a path leading to *0* indicates a cylinder of assignments mapping to *0*. We can express this cylinder in terms of a set of literals: for each variable label v encountered on the path, add literal v to the set if the edge taken away from v's node is labeled *0* and add literal $\neg v$ if the edge is labeled *1*. Below we refer to such a set of literals obtained for a path p as a *lemma representing p*.

A BDD is a canonical representation of a function on a particular ordering of input variables. For every clause set there is a logically equivalent BDD, but for a particular BDD there may be many logically equivalent clause sets, one of which can be constructed as follows: for each path p terminating at the *0* node, construct the lemma representing p; add the clause containing exactly those literals to the clause set. The operation on a BDD P which yields all such clauses will be referred to below as *clause gathering on P*.

4 Strengthening

Strengthening is a binary operation on two Boolean functions. Strengthening makes use of the well-known operation called *existential quantification*: given a Boolean function f containing a variable v, v may be existentially quantified away from f by replacing f with $f|_{v=0} \vee f|_{v=1}$.

Definition 6 Existential Quantification*: Given a Boolean function f_v containing a variable v. The result of* Existentially quantifying v *away from* f_v *is the*

[2]Although v can be any Boolean function, for exposition purposes we restrict v to represent a variable only. This is done without loss of generality by allowing, if necessary, v to be a new variable (not contained in the original expression) which is equivalent to a Boolean expression.

function $f_v|_{v=0} \vee f_v|_{v=1}$. We use $\exists v f_v$ to denote the result of existentially quantifying v away from f_v. $\square$

In what follows, we always suppose the two functions to be strengthened are given as BDDs. Let P and Q denote two BDDs. To *strengthen* P with respect to Q means (*i*) existentially quantify away all variables in Q that are not in P forming a new BDD Q'', then (*ii*) conjoin P and Q'' using the logical *and* operator forming a new BDD P'. In other words we have:

Definition 7 Strengthening*: $P' = P \wedge \exists X Q$, where P and Q are BDDs and X is the set of variables occurring in Q but not in P.* $\square$

In SBSAT, strengthening is used during preprocessing to extract primitive inferences. A *primitive inference* is a single literal v or $\neg v$, or a single equivalence $v_i = v_j$ that is implied by one BDD. When a primitive inference is found it is immediately applied over the entire collection of input BDDs.

SBSAT applies strengthening in both directions. That is, given P and Q, SBSAT strengthens P with respect to Q and replaces P with the result then strengthens Q with respect to P and replaces Q with the result. When doing so in what follows we refer to the strengthening of P and Q. This pairwise process is iterated over the entire set of BDDs in the given input. This process allows SBSAT to collect many primitive inferences which were originally implied by two BDDs. In some cases strengthening even allows SBSAT to collect primitive inferences which were originally implied by multiple BDDS.

Because of the close relationship between BDDs and clauses noted above we can translate what is happening during the strengthening of P and Q to applications of the Davis-Putnam Procedure on clauses sets corresponding to P and Q. We use the term ϕ to mean a CNF expression. In what follows we use the symbol $\Leftrightarrow$ between different expressions of functions to mean logical equivalence.

Clause Sharing Method: Let ϕ_P and ϕ_Q be CNF expressions. Let X be the set of variables in ϕ_Q but not in ϕ_P. The *clause sharing method* of ϕ_P with respect to ϕ_Q is $\phi_{P'} = \phi_P \cup DP_X(\phi_Q)$. $\square$

Definition 8 Cross Product*: Given two set of sets representations ϕ_1 and ϕ_2 of CNF expressions, the* cross product *of ϕ_1 and ϕ_2, denoted by $\times$ is given as*

$$\phi_1 \times \phi_2 = \{c_1 \cup c_2 : c_1 \in \phi_1, c_2 \in \phi_2\}.$$

$\square$

Remark 9 *$\phi_1 \times \phi_2$ represents the CNF expression which is the logical "or" of the CNF expressions represented by ϕ_1 and ϕ_2.*

Lemma 10 *Let ϕ_v be a CNF expression containing at least one clause with literal v. Then*

$$\phi_v|_{v=0} = \{c \setminus \{v\} : c \in \phi_v, \neg v \notin c\}$$

and

$$\phi_v|_{v=1} = \{c \setminus \{\neg v\} : c \in \phi_v, v \notin c\}.$$

Proof: Consider only the first part; the second part follows from a symmetric argument. Given v has value 0, any clause containing the literal $\neg v$ always has value 1. Hence, removing such a clause from ϕ_v does not change the functionality of ϕ_v. Moreover, any clause containing the literal v has precisely the same functionality as that clause minus the literal v given v has value 0. Therefore, removing clauses containing $\neg v$ and literals containing v preserves the logical equivalence of ϕ_v, given v has value 0. □

Lemma 11 *Let ϕ be a CNF expression. Suppose $c \in \phi$ is a clause containing two complementary literals. Then $\phi \Leftrightarrow \phi \setminus \{c\}$.*

Proof: Clause c is tautologous. □

Lemma 12 *Let ϕ be a CNF expression. Suppose two clauses c_1 and c_2 are in ϕ and $c_1 \subset c_2$. Then $\phi \Leftrightarrow \phi \setminus \{c_2\}$.*

Proof: Clause c_2 has value 1 whenever clause c_1 does. □

The following is well known but we prove it here for expository purposes.

Lemma 13 *Given a CNF expression ϕ_v containing no tautologous clauses,*

$$\exists v \phi_v = DP_v(\phi_v).$$

The case where ϕ_v does not contain at least one clause with either literal v or literal $\neg v$ is trivial. Observe that the set of clauses represented by $\exists v \phi_v$ is identical to the set of clauses represented by $DP_v(\phi_v)$.

Proof: From Remark 9, Definition 6, and Lemma 10,

$$\exists v \phi_v = \{c_1 \setminus \{v\} : c_1 \in \phi_v, \neg v \notin c_1\} \times \{c_2 \setminus \{\neg v\} : c_2 \in \phi_v, v \notin c_2\}.$$

Then, by Definition 8,

$$\exists v \phi_v = \{c_1 \cup c_2 \setminus \{v, \neg v\} : c_1, c_2 \in \phi_v, \neg v \notin c_1, v \notin c_2\}.$$

By Lemma 11, all tautologies can be removed to get

$$\exists v \phi_v = \{c_1 \cup c_2 \setminus \{v, \neg v\} : c_1, c_2 \in \phi_v, \neg v \notin c_1, v \notin c_2, \nexists w \neq v : w, \neg w \in c_1 \cup c_2\}. \quad (2)$$

The right side of (2) can be split to get

$$\begin{aligned} &\exists v \phi_v = \quad (3)\\ &\quad \{c_1 \cup c_2 \setminus \{v, \neg v\} : c_1, c_2 \in \phi_v, v \in c_1, \neg v \in c_2, \nexists w \neq v : w, \neg w \in c_1 \cup c_2\}\\ &\quad \cup \{c_1 \cup c_2 \setminus \{v\} : c_1, c_2 \in \phi_v, v \in c_1, \neg v \notin c_2, \nexists w \neq v : w, \neg w \in c_1 \cup c_2\}\\ &\quad \cup \{c_1 \cup c_2 \setminus \{\neg v\} : c_1, c_2 \in \phi_v, v \notin c_1, \neg v \in c_2, \nexists w \neq v : w, \neg w \in c_1 \cup c_2\}\\ &\quad \cup \{c_1 \cup c_2 : c_1, c_2 \in \phi_v, v \notin c_1 \cup c_2, \neg v \notin c_1 \cup c_2, \nexists w \neq v : w, \neg w \in c_1 \cup c_2\}. \end{aligned}$$

By Lemma 12, $\{c_1 \cup c_2 \setminus \{v\} : v \in c_1, \neg v \notin c_2\}$ is removed from the right side of (3) by $\{c_2 \cup c_2 : \neg v \notin c_2\}$ (the last line of (3)) and $\{c_1 \cup c_2 \setminus \{\neg v\} : v \notin c_1, \neg v \in c_2\}$ is removed due to $\{c_1 \cup c_1 : v \notin c_1\}$ (also the last line of (3)). Thus (3) simplifies to

$$\begin{aligned}
&\exists v \phi_v = \qquad (4)\\
&\quad \{c_1 \cup c_2 \setminus \{v, \neg v\} : c_1, c_2 \in \phi_v, v \in c_1, \neg v \in c_2, \nexists w \neq v : w, \neg w \in c_1 \cup c_2\}\\
&\quad \cup \{c_1 \cup c_2 : c_1, c_2 \in \phi_v, v \notin c_1 \cup c_2, \neg v \notin c_1 \cup c_2, \nexists w \neq v : w, \neg w \in c_1 \cup c_2\}.
\end{aligned}$$

Also, by Lemma 12, $\{c_1 \cup c_2 : c_1 \neq c_2, v \notin c_1 \cup c_2, \neg v \notin c_1 \cup c_2\}$ in (4) can be removed due to $\{c_1 \cup c_1 : v \notin c_1, \neg v \notin c_1\}$ simplifying (4) to

$$\begin{aligned}
&\exists v \phi_v =\\
&\quad \{c_1 \cup c_2 \setminus \{v, \neg v\} : c_1, c_2 \in \phi_v, v \in c_1, \neg v \in c_2, \nexists w \neq v : w, \neg w \in c_1 \cup c_2\}\\
&\quad \cup \{c : c \in \phi_v, v \notin c, \neg v \notin c, \nexists w \neq v : w, \neg w \in c\}.
\end{aligned}$$

Because ϕ_v contains no tautological clauses by hypothesis,

$$\begin{aligned}
&\exists v \phi_v =\\
&\quad \{c_1 \cup c_2 \setminus \{v, \neg v\} : c_1, c_2 \in \phi_v, v \in c_1, \neg v \in c_2, \nexists w \neq v : w, \neg w \in c_1 \cup c_2\}\\
&\quad \cup \{c : c \in \phi_v, v \notin c, \neg v \notin c\}.
\end{aligned}$$

Rewriting the last line gives

$$\begin{aligned}
&\exists v \phi_v = \qquad (5)\\
&\quad \{c_1 \cup c_2 \setminus \{v, \neg v\} : c_1, c_2 \in \phi_v, v \in c_1, \neg v \in c_2, \nexists w \neq v : w, \neg w \in c_1 \cup c_2\}\\
&\quad \cup\, \phi_v \setminus \{c : c \in \phi_v, v \in c \text{ or } \neg v \in c\}.
\end{aligned}$$

The right side of (5) is identical to the right side of (1). The lemma follows. □

Remark 14 *Let P and Q be BDDs. Let ϕ_P and ϕ_Q be CNF expressions such that $P \Leftrightarrow \phi_P$ and $Q \Leftrightarrow \phi_Q$. Then $P \wedge Q \Leftrightarrow \phi_P \cup \phi_Q$.*

The following theorem is the main result. It describes strengthening in terms of the Davis-Putnam Procedure and conjunction.

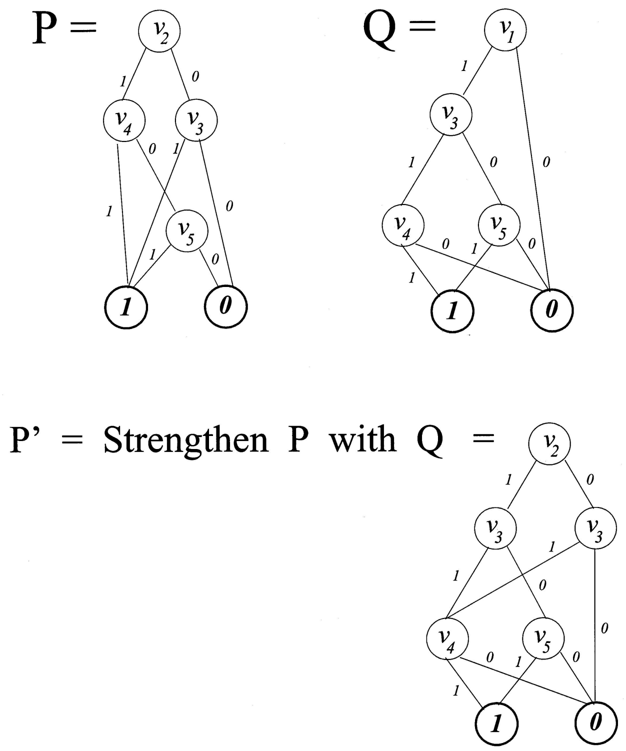
Theorem 15 *Let P and Q be BDDs. Let ϕ_P and ϕ_Q be CNF expressions such that $P \Leftrightarrow \phi_P$ and $Q \Leftrightarrow \phi_Q$. Let P' be the result of strengthening P with Q. Let $\phi_{P'}$ be the result of applying the clause sharing method to ϕ_P and ϕ_Q. Then $P' \Leftrightarrow \phi_{P'}$.*

Proof:

1. Let X be the set of variables in Q but not in P, note that X is also the set of variables in ϕ_Q but not in ϕ_P. Let $Q'' = \exists X Q$. Let $\phi_{Q''} = DP_X(\phi_Q)$. By Lemma 13, $Q'' \Leftrightarrow \phi_{Q''}$.

2. By definition of the clause sharing method, $\phi_{P'} = \phi_P \cup \phi_{Q''}$. By the definition of strengthening, $P' = P \wedge Q''$. Since $P \Leftrightarrow \phi_P$ and $Q'' \Leftrightarrow \phi_{Q''}$, then by Remark 14, $P' \Leftrightarrow \phi_P \cup \phi_{Q''}$ and therefore $P' \Leftrightarrow \phi_{P'}$. $\square$

In Theorm 15, not only is Q'' logically equivalent to $\phi_{Q''}$, but it is identical to the set of clauses resulting from the application of the clause gathering method on Q''. This is not true for step 2 of the proof, $\phi_{P'}$ is not always identical to the set of clauses resulting from the application of the clause gathering method on P', namely $\phi_{P''}$ (as shown in the CNF counterpart to Figure 1). However because $\phi_{P'}$ and $\phi_{P''}$ are logically equivalent, a series of resolutions and subsumptions can be done on either set to get a set of clauses that is identical to the other. Because BDDs are canonical if two BDDs represent the same function, meaning they are logically equivalent, they are also identical. It follows that if we have two functions represented by CNF expressions which are logically equivalent, their respective BDDs are identical and thus we can have a complete characterization of strengthening in CNF terms.



CNF counterpart to Figure 1:

Let ϕ_P and ϕ_Q be the results of clause gathering on P and Q respectively. Then,

$\phi_P = \{\{\neg v_2, v_4, v_5\}, \{v_2, v_3\}\}$
$\phi_Q = \{\{v_1\}, \{\neg v_1, v_3, v_5\}, \{\neg v_1, \neg v_3, v_4\}\}$
$\phi_{Q''} = DP_{v_1}(\phi_Q) = \{\{v_3, v_5\}, \{\neg v_3, v_4\}\}$
$\phi_{P'} = \phi_P \cup \phi_{Q''} = \{\{\neg v_2, v_4, v_5\}, \{v_2, v_3\}, \{v_3, v_5\}, \{\neg v_3, v_4\}\}.$

Let $\phi_{P''}$ be there result of clause gathering on P'. Then,

$\phi_{P''} = \{\{\neg v_2, \neg v_3, v_4\}, \{\neg v_2, v_3, v_5\}, \{v_2, \neg v_3, v_4\}, \{v_2, v_3\}\}$

Figure 1: Strengthening two BDDs P and Q

References

[1] Bryant, R. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677-691, 1986.

[2] M. Davis and H. Putnam. A computing procedure for quantification theory. JACM, July 1960.

[3] Dransfield, M., J. Franco, J.S. Schlipf, W.M. Vanfleet, J. Ward, and S. Weaver. A state-based, BDD-based Satisfiability Solver. Submitted to *Annals of Mathematics and Artificial Intelligence*.

[4] Freeman, J.W. Improvements to Propositional Satisfiability Search Algorithms. *Ph.D. dissertation in Computer and Information Science*, University of Pennsylvania, 1995.

[5] Marques-Silva, J.P., and K.A. Sakallah. GRASP - A New Search Algorithm for Satisfiability. Technical Report CSE-TR-292-96, Department of Electrical Engineering and Computer Science, University of Michigan, 1996.

[6] Moskewicz, M.W., C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Engineering a (Super?) Efficient SAT Solver. In *Proceedings of the 38th ACM/IEEE Design Automation Conference*, ACM, New York, 2001.

[7] Zhang, H. SATO: an Efficient Propositional Prover. In *Proceedings of the International Conference on Automated Deduction (CADE'97)*, 1249, Lecture Notes in Artificial Intelligence, 1997.