A Satisfiability Procedure for Quantified Boolean Formulae

David A. Plaisted

Computer Science Department, University of North Carolina
Chapel Hill, NC 27599-3175, USA

Armin Biere

ETH Zürich, Computer Systems Institute
8092 Zürich, Switzerland

Yunshan Zhu

Advanced Technology Group, Synopsys Inc.
Mountain View, CA 94040, USA

Abstract

We present a satisfiability tester QSAT for quantified Boolean formulae and a restriction QSAT_{CNF} of QSAT to unquantified conjunctive normal form formulae. QSAT makes use of procedures which replace subformulae of a formula by equivalent formulae. By a sequence of such replacements, the original formula can be simplified to **true** or **false**. It may also be necessary to transform the original formula to generate a subformula to replace. QSAT_{CNF} eliminates collections of variables from an unquantified clause form formula until all variables have been eliminated. QSAT and QSAT_{CNF} can be applied to hardware verification and symbolic model checking. Results of an implementation of QSAT_{CNF} are described, as well as some complexity results for QSAT and QSAT_{CNF}. QSAT runs in linear time on a class of quantified Boolean formulae related to symbolic model checking. We present the class of "long and thin" unquantified formulae and give evidence that this class is common in applications. We also give theoretical and empirical evidence that QSAT_{CNF} is often faster than Davis and Putnam-type satisfiability checkers and ordered binary decision diagrams (OBDDs) on this class of formulae. We give an example where QSAT_{CNF} is exponentially faster than BDDs.

Key words: QBF, Satisfiability, Davis & Putnam Procedure, BDDs, Cut Width, Circuit Verification, Model Checking.

QSAT is a new decision procedure for satisfiability of quantified Boolean formulae with potential applications to hardware verification and symbolic model checking. The philosophy of QSAT is to test satisfiability of a formula by repeatedly replacing subformulae by simpler equivalent subformulae. QSAT_{CNF} is an application of QSAT to unquantified clause form formulae, interpreted as quantified Boolean formulae by considering all free variables to be existentially quantified. QSAT and QSAT_{CNF} test satisfiability of a formula by successively eliminating variables from it, producing an equivalent formula, until all variables have been eliminated. The motivation for QSAT and QSAT_{CNF} is that satisfiability testers for unquantified formulae such as Davis and Putnam's method seem to be more efficient than BDDs (ordered BDDs, OBDDs) when there is not too much backtracking. BDDs can process large (unquantified) formulae because they make use of an ordering of the variables which breaks the processing down into smaller steps that are easier to perform. The idea of QSAT and QSAT_{CNF} is to import this BDD philosophy into satisfiability testing, by applying a modification of Davis and Putnam's method not to the whole formula at once, but piece by piece, where each application of the modified Davis and Putnam method to a piece of the formula simplifies the formula. This reduces the amount of backtracking. QSAT and QSAT_{CNF} can be built on top of any satisfiability tester for unquantified Boolean formulae, including Stålmarck's method [1].

It appears (see section 11) that BDDs are good for systems that are long and thin, such as adders. These are also systems for which QSAT_{CNF} should be efficient, because such systems can be broken into parts having a limited amount of communication between them. Each part corresponds to a subformula of the original formula. QSAT_{CNF} can simplify one such subformula, and then be applied recursively to the remaining formula. Thus it may be that QSAT_{CNF} is efficient on a large number of applications where BDDs are currently used. Similarly, QSAT should be efficient for quantified Boolean formulae that are long and thin, in a certain sense. BDDs can also be efficient on problems that are not long and thin, such as some versions of barrel shifters. QSAT_{CNF} may not be as fast on such problems.

The general method by which Boolean formulae are used for system testing is the following:

(1) A Boolean formula G is constructed from a system S and its specification, expressing that the system does not satisfy its specification. Methods for generating G are well known.
(2) The formula G is tested for satisfiability (consistency).
(3) If G is unsatisfiable, then S is correct. If G is satisfiable, then there is an

Email addresses: `plaisted@cs.unc.edu` (David A. Plaisted), `biere@inf.ethz.ch` (Armin Biere), `yunshan@synopsys.com` (Yunshan Zhu).

error in S, and the nature of the satisfiability of G can help to identify the error in S.

The system S can be a computer circuit or some system containing interconnected objects, which can be defined as a component or part of a physical system, such as a gate in a circuit. The formula G can be obtained by defining the formula A as a Boolean formula representing the system S and B as a Boolean formula representing the statement that S fails to satisfy its specification. Then G can be taken to be the conjunction $A \wedge B$ of these two formulae, expressing that S fails to satisfy its specification.

Binary decision diagrams (BDDs) [2] have been widely used in CAD applications such as logic synthesis, testing and formal verification. BDDs transform a circuit into a canonical form, depending on an ordering of the Boolean variables, and two circuits are equivalent if and only if they have the same canonical form. For many kinds of circuits, BDDs work very well, especially when a good ordering of the variables can be found. Equivalence checking [3,4] is important, because one can verify a new or optimized circuit by showing that it is equivalent to an old and trusted circuit. Commercial equivalence checkers can now verify circuits with millions of gates which are clearly out of reach for traditional simulation.

Satisfiability algorithms for Boolean formulae in clause form can also be used for hardware verification[5]. In this approach, the formula G above is in clause form, which is a special form of unquantified Boolean formula. An efficient method such as Davis and Putnam's method can then be applied to test if the formula G is satisfiable. Davis and Putnam's method was first described in the paper [6], though modern implementations differ in some ways. Most modern implementations use the method of DPLL [7], which eliminates variables by case analysis rather than ordered resolution. A recent, very efficient implementation of DPLL is described in [8]. [9] combined BDDs and satisfiability testers to solve verification problems. Another method for satisfiability testing of unquantified Boolean formulae, not necessarily in clause form, is disclosed in [1]. This method works breadth-first, trying all possible truth assignments to small subsets of the variables of a formula. From these assignments, information about dependencies between variables is obtained which can aid in determining satisfiability.

Automatic test pattern generation (ATPG) is another important problem in CAD. Given a combinational circuit, a stuck-at fault causes a wire to have a constant value. The task of ATPG is to generate input patterns that detect such stuck-at faults. It was well known that ATPG is equivalent to propositional satisfiability. Efficient SAT-based ATPG techniques have been developed[10].

Symbolic model checking [11,12,13] is concerned with verifying sequential systems. The use of BDDs for symbolic model checking was a breakthrough, because it permitted much larger systems to be verified than was possible before. BDDs permit the state of a system to be represented and manipulated efficiently, in many cases. However, the paper [14] gives some Boolean formulae obtained from symbolic model checking problems on which satisfiability algorithms such as DPLL and Stalmarck's method are more efficient than BDDs. Other examples are given in [5] in which the smallest BDD for a Boolean function is of exponential size, regardless of the variable ordering used. There is therefore also an interest in seeing how far satisfiability-based approaches can extend in symbolic model checking applications.

Boolean satisfiability has been extensively studied. See [15] for an excellent survey of a wide range of satisfiability techniques. However some important problems in hardware verification cannot be expressed with quantifier-free Boolean formulae. Computing fixed points in symbolic model checking is one such example. Therefore there is a need for satisfiability testers for quantified Boolean formulae. There has been some work in decision procedures for quantified boolean formulae [16,17]. Our technique differs from the previous work in that it modifies the quantified Boolean formula from the inside out, rather from the outside in. That is, QSAT can choose to process a quantifier other than one of the outermost quantifiers of the formula.

The same flexibility in the processing of quantifiers is inherent in QSAT_{CNF}. This flexibility enables QSAT_{CNF} to exploit structures in the hardware verification domain. In particular, QSAT_{CNF} is very efficient in handling long and thin circuits. Note that the propositional satisfiability problem without quantifiers is a special case of quantified Boolean formulae where all variables are assumed to be existentially or universally quantified. Since QSAT_{CNF} resembles the behaviors of BDDs, it complements the traditional DPLL style SAT solvers. An interesting future research direction is to combine the two approaches.

Many of the problems mentioned in [18] are "long and thin," meaning that they have small cut widths. Actually, the definition of cut width differs somewhat from one paper to another, but all such definitions capture approximately the same idea. We will show below that many of the problems from [19] are also long and thin, as well as several other benchmark problems we constructed. In fact all problems we tried from [19] have an average cut width of 19 or less. This suggests that the class of long and thin problems is fairly common in applications. It turns out that the worst-case time bound for QSAT_{CNF} on this class of long and thin problems is better than that of BDD's and Davis and Putnam's method by an exponential factor. We also have examples where the QSAT_{CNF} implementation is faster than BDD's and Davis and Putnam's method on this class of problems. This gives empirical and theoretical evidence that QSAT_{CNF} will be faster on many problems from this class.

The first author has obtained US patent 6,131,078 on QSAT.

1 Terminology

1.1 Boolean quantifiers and operators

$\exists$ is used for existential quantification and $\forall$ for universal quantification. $\wedge$ is used for logical conjunction (and), $\vee$ for logical disjunction (or), and $\neg$ for logical negation. The symbol $\leftrightarrow$ is used for equivalence and $+$ for exclusive or. The constants **true** and **false** are called truth values. The Boolean operators are defined on truth values in standard ways, so that $x \wedge y$ is **true** if and only if x is **true** and y is **true**, $x \vee y$ is **true** if and only if x is **true** or y is **true**, $\neg x$ is **true** if and only if x is **false**, $x \leftrightarrow y$ is **true** if and only if x and y have the same truth value, and $(x + y)$ is **true** if and only if x and y have different truth values. Capital letters (like X and Y) refer to sets or sequences of Boolean variables.

1.2 Formulae

A *quantified* Boolean formula is a formula built up from Boolean variables and the Boolean operators of conjunction, disjunction, negation, and other Boolean operators. Thus $(x \wedge (y \vee z))$ is a quantified Boolean formula. Boolean quantifiers are also allowed to occur in quantified Boolean formulae. Thus if B is a quantified Boolean formula and X is a set of Boolean variables, then $\exists X[B]$ and $\forall X[B]$ are also quantified Boolean formulae, where $\exists X$ is considered an existential quantifier and $\forall X$ is considered a universal quantifier. $\forall X[B]$ is often considered to abbreviate $\neg\exists X[\neg B]$. If X is the set or list $\{x, y, z\}$, then $\exists X[A]$ abbreviates the quantified Boolean formula $\exists x[\exists y[\exists z[A]]]$. An example of a quantified Boolean formula is $\exists x[x \wedge \neg\exists y[y \vee z]]$. Often the term "quantified" is dropped. A quantified Boolean formula without any occurrences of quantifiers is said to be *unquantified.* An occurrence of a variable x in a formula A is called free if this occurrence is not within the scope of a quantifier $\exists x$ or $\forall x$. Only the occurrence of z is free in the example formula. Variable occurrences that are not free are called bound. If A is a quantified Boolean formula and X is a set of variables, then $A[X]$ denotes a formula A that contains the free variables X. A formula B having the free variables X is often taken to abbreviate $\exists X[B]$ or $\forall X[B]$. A *literal* is a Boolean variable or its negation. If y is a variable, it is assumed that $\neg\neg y$ is identical to y. The literal $\neg y$ is the *complement* of y, and likewise y is the complement of $\neg y$. The literals y and $\neg y$ are said to be *complementary.* A *clause* is a disjunction of literals, as,

$x \vee \neg y \vee z$. A *set of clauses*, also termed a *conjunctive normal form* formula, is a conjunction of clauses, such as $C \wedge D \wedge E$, where C, D, and E are clauses.

1.3 Subformulae

Each Boolean formula A is a subformula of itself. Also, if A is of the form $B \otimes C$, where $\otimes$ is a Boolean operator, and D is a subformula of B or C, then D is also a subformula of A. Likewise, if D is a subformula of B, then D is a subformula of $\neg B$ and a subformula of $\Omega X[B]$ where Ω is $\exists$ or $\forall$.

1.4 Simplifications

For a Boolean formula A, $A|_y$ refers to A with all free occurrences of the Boolean variable y replaced by **true**, and the resulting formula simplified as many times as possible with respect to usual Boolean simplifications. These are the following: $B \wedge$ **true** simplifies to B, $B \wedge$ **false** simplifies to **false**, and other standard simplifications for eliminating **true** and **false** from Boolean expressions. Also, $\exists x[B]$ and $\forall x[B]$ simplify to B if there are no free occurrences of the variable x in B. Let $A|_{\neg y}$ refers to A with all free occurrences of the variable y replaced by **false**, and the resulting formula likewise simplified. There are also the additional simplifications $B \wedge p$ simplifies to $B|_p \wedge p$ and $p \wedge B$ simplifies to $p \wedge B|_p$ if p is a literal. Also, $B \vee p$ simplifies to $B|_{\neg p} \vee p$ and $p \vee B$ simplifies to $p \vee B|_{\neg p}$. If A is a Boolean formula and x is a Boolean variable, then $\exists x[A]$ is defined to be equivalent to the formula $A|_x \vee A|_{\neg x}$.

1.5 Interpretations

An interpretation is a set I of literals, often viewed as a conjunction of its elements, such that no pair of complementary literals occur in I. If I is an interpretation and A is a Boolean formula, then $A|_I$ is A with all occurrences of x replaced by **true**, for $x \in I$, and all occurrences of x replaced by **false**, for x such that $\neg x \in I$, with simplifications applied as before. The formula $A|_I$ is read "A relative to I." A formula B is satisfiable if there is an interpretation I such that $B|_I$ is **true**. Such an interpretation I is said to satisfy B. An interpretation I such that $B|_I$ is **false** is said to contradict or falsify B. A formula B is falsifiable if there is an interpretation I such that $B|_I$ is **false**. If B is not satisfiable, then B is unsatisfiable. A formula B is a tautology if for all I which assign truth values to all free variables of B, $B|_I$ is **true**. A formula B is a contradiction if for all I which assign truth values to all free variables of B, $B|_I$ is **false**. Two formulae A and B are equivalent ($A \equiv B$) if and only if for all interpretations I of their free variables, $A|_I$ is **true** if and only if $B|_I$ is **true**. A formula A is said to logically imply a formula B if and only if for all interpretations I of the free variables of A and B, if $A|_I$ is **true** then $B|_I$ is **true**. Also, a formula A logically implies B if and only if the formula $\neg A \vee B$ is a tautology.

1.6 Equivalences

As the standard Boolean equivalences which may be used for transforming formulae into equivalent formulae, the following may be taken, possibly with others added: $(A \otimes B) \equiv (B \otimes A)$ where $\otimes$ is $\wedge$, $\vee$, $\leftrightarrow$, or $+$, $(A \otimes B) \otimes C \equiv A \otimes (B \otimes C)$, where $\otimes$ is $\wedge$ $\vee$, $\leftrightarrow$, or $+$, $\neg\neg B \equiv B$, $\neg(A \wedge B) \equiv \neg A \vee \neg B$, $\neg(A \vee B) \equiv \neg A \wedge \neg B$, together with other equivalences for pushing negation inside connectives and distributing $\wedge$ over $\vee$ and defining other connectives in terms of negation, $\wedge$ and $\vee$. Also, $\Omega X[A \otimes B] \equiv (\Omega X[A]) \otimes B$ where $\otimes$ is $\wedge$ or $\vee$ and Ω is $\exists$ or $\forall$, and the variables X do not occur free in B, $\Omega X[B] \equiv B$ if the variables X do not occur free in B, $\forall X[B] \equiv \neg\exists X[\neg B]$, $\exists X[B] \equiv \neg\forall X[\neg B]$, $\exists X[\exists Y[B]] \equiv \exists Y[\exists X[B]]$, $\forall X[\forall Y[B]] \equiv \forall Y[\forall X[B]]$, $\exists X[A \vee B] \equiv (\exists X[A]) \vee (\exists X[B])$, and $\forall X[A \wedge B] \equiv (\forall X[A]) \wedge (\forall X[B])$. These may be used in either direction.

1.7 Duals

If a formula A has only the Boolean operators $\wedge$, $\vee$, and $\neg$ and quantifiers, then a formula B is called the dual of A if B is obtained from A by interchanging the Boolean operators $\wedge$ and $\vee$ and interchanging the quantifiers $\forall$ and $\exists$ and adding an additional negation to all the Boolean variables. Such a B is equivalent to $\neg A$. It is often the case that a method which applies to a formula A can also be applied to the dual B of A with small modifications. For example, A is satisfiable if and only if the dual B of A is not a tautology.

Formulae are constructed from circuits as follows. A signal on a wire is identified with a Boolean variable. Each gate is converted into a Boolean formula expressing the required relationship between its input and output signals. Thus an OR-gate with inputs x and y and output z would be converted into the formula $z \leftrightarrow (x \vee y)$. A formula representing the entire circuit is obtained as the conjunction (and) of all the formulae for its gates. This formula can be converted to clause form (if desired) by converting each of the gate formulae to clause form; standard methods for doing this are known. For example, the clause form for the formula $z \leftrightarrow (x \vee y)$ is $(\neg z \vee x \vee y) \wedge (\neg x \vee z) \wedge (\neg y \vee z)$.

2 High-level description of QSAT

Let the relation $\succ$ be defined so that $F[\exists Z[B]] \succ F[\exists X[B_1 \wedge B_2']]$, where F, B_1, B_2', X, and Z are as follows:

F is a quantified Boolean formula with no free variables. $\exists Z[B]$ is a subformula of F, so we write $F[\exists Z[B]]$ to indicate this. Z is a list of variables which consists of the variables in the lists X and Y in some order, so $\exists Z[B] \equiv \exists X[\exists Y[B]]$. The formula B is equivalent to $B_1 \wedge B_2$, where the variables Y do not occur free in B_1. The formula B_2' is equivalent to $\exists Y[B_2]$ but does not have the quantifiers $\exists Y$.

The relation $\succ$ is defined dually on formulae of the form $F[\forall X[\forall Y[B]]]$.

Theorem 1 *If $F \succ F'$ then $F \equiv F'$.*

Proof. $\exists Z[B] \equiv \exists X[\exists Y[B]] \equiv \exists X[\exists Y[B_1 \wedge B_2]] \equiv \exists X[B_1 \wedge \exists Y[B_2]] \equiv \exists X[B_1 \wedge B_2']$. q.e.d.

Also, if $F \succ F'$ then F' has some quantifiers eliminated that appear in F. Let the relation $\succ^*$ be defined by $F_1 \succ^* F_n$ if $F_i \succ F_{i+1}$ for all i, $1 \le i < n$.

The algorithm QSAT is as follows:

```
procedure qsat($F$);
   find $F'$ such that $F \succ^* F'$ and $F'$ has no quantifiers in it;
   if $F'$ is true then return true;
   else if $F'$ is false then return false;
   else return error; fi
end qsat;
```

Theorem 2 *If* QSAT*(F) returns* **true** *then F is a tautology and if* QSAT*(F) returns* **false** *then F is unsatisfiable. Also, for every formula F having no free variables, either $F \succ^*$* **true** *or $F \succ^*$* **false***.*

Proof. If QSAT(F) returns **true** then $F \succ^*$ **true**, so $F \equiv$ **true** and therefore F is a tautology. If QSAT(F) returns **false** then $F \succ^*$ **false**, so $F \equiv$ **false** and therefore F is unsatisfiable. For the rest, for any F containing at least one variable, one can find F' containing fewer variables than F such that $F \succ F'$. By repeated operations of this form, all variables will be eliminated, yielding a formula equivalent to **true** or **false**. q.e.d.

The difficult parts of QSAT are (1) expressing B as $B_1 \wedge B_2$ and (2) finding B_2' equivalent to $\exists Y[B_2]$ but without the quantifiers $\exists Y$ occurring in B_2'. We refer to such a selection of a subset Y of the variables of B as a *cut* of B. Finding B_2' given Y and B_1 is done by the procedures **simp** and **sat** which will be described below. Finding B_2' requires up to 2^n calls to a satisfiability procedure, where n is the number of variables in X that appear free in B_2. Therefore it is important to find a cut of B that makes n small. The quantity n will be called the *cost* of the cut. It is also important to find a cut so that Y does not contain too many variables, because the amount of backtracking increases with the size of Y.

Typically B is a conjunction of subformulae and B_2 will be chosen as all of these subformulae that contain the variables Y free. Thus finding a cut corresponds to choosing a set Y of variables to eliminate from B. The quantity n will also be called the cost of the set Y of variables. This quantity represents the connections between a portion of the circuit or system represented by B_2 and the rest of the circuit or system. Therefore, it is important to choose the set Y of variables to eliminate as a set that is not too large and is isolated from the rest of the system as much as possible. For long, thin systems, one can choose Y as some of the variables that appear at one end of the system, for example.

It is also possible to choose cuts based on an *ordering* of the variables Z. Recall that B is typically a conjunction of many subformulae. Two variables z and z' of Z are said to be *closely related* if they appear free in the same subformula of B. Recall that variables correspond to signals or parts of the system being verified, and often correspond to physical locations in this system. We can choose an ordering z_1, z_2, …, z_k of the variables of Z so that variables that are closely related appear near each other in this ordering. For example, if the system being verified is long and thin, then we can choose the ordering so that one progresses through the system from one end to the other as one progresses up the ordering. If one is verifying that two systems are equivalent, and both are long and thin, then the ordering can be chosen to progress in parallel through both systems, from one end to the other, as one progresses up the ordering. Once the ordering has been chosen, then the cuts can be chosen to eliminate all variables larger than some bound b from B, where b is chosen so that the cost of the cut is small and so that the number of variables being eliminated each time is not too large.

3 simp and sat procedures

QSAT makes use of two procedures, **simp** and **sat** which can be used to replace a quantified Boolean formula by a simpler equivalent formula, together with a number of refinements which are subsequently described. The procedures **simp** and **sat** are applied repeatedly in QSAT.

The procedure **simp** takes a Boolean formula A and produces another formula A' equivalent to A. Informally, the procedure **simp** gathers together a complete collection of interpretations I_i that make A **false**, and for each one it constructs a clause D_i expressing the negation of I_i. Then the conjunction (and) of the D_i is equivalent to A, because this conjunction expresses the fact that none of the I_i are **true**. Since all ways of making the formula A **false** have been excluded, the fact that A is **true** has been expressed. A' equivalent to A has been obtained, where A' is the conjunction of these D_i.

We now express the procedure **simp** algorithmically. This procedure takes a Boolean formula A and an interpretation I as arguments. It returns a set A' of clauses containing only the free variables of A such that if I is empty, then A' is logically equivalent to A. If A occurs in some other Boolean formula F, then let F' be F' with an occurrence of A replaced by A'. Then F' is equivalent to F. Thus satisfiability of F can be tested by finding some such subformula A, replacing it in F by A', and testing satisfiability of F'. Since F' is often simpler than F (any non-free variables in A have been eliminated, at least), it may be simpler to test satisfiability of F' than F. Of course, universal quantifiers in F can be replaced by existential quantifiers using the fact that $\forall X[C[X]]$ is equivalent to $\neg\exists X[\neg C[X]]$. This gives a decision procedure to decide if F is a tautology (it simplifies to **true**) or a contradiction (it simplifies to **false**). This procedure can also test if a formula F with free variables is satisfiable, as follows: let Y be the free variables in F. Then F is satisfiable if and only if $\exists Y[F]$ is a tautology, and whether $\exists Y[F]$ is a tautology can be tested by reducing it to **true** or **false** by successive simplifications.

The procedures **simp** and **sat** make use of two auxiliary procedures **unsat** and **taut**. The procedures **unsat** and **taut** can return **true**, **false**, or $\uparrow$ (unknown). These procedures must satisfy the following conditions:.

(1) If **unsat**(B) returns **true** then B is unsatisfiable.
(2) If B is unsatisfiable and has no free variables then **unsat**(B) returns **true**.
(3) If **unsat**(B) returns **false** then B is satisfiable.
(4) If B is satisfiable and has no free variables then **unsat**(B) returns **false**.
(5) If **taut**(B) returns **true** then B is a tautology.
(6) If B is a tautology and has no free variables then **taut**(B) returns **true**.

Intuitively, **unsat** is a procedure that tests whether a formula is unsatisfiable, but it can give up before an answer is computed and return $\uparrow$ (unknown) in some cases. Also, **taut** is a procedure for testing whether a formula is a tautology, and it can also give up before an answer is computed and return $\uparrow$

(unknown) in some cases. Both **unsat** and **taut** can be arbitrary procedures for testing unsatisfiability or tautology. The fact that any such procedures can be used makes it possible to implement **simp** and **sat** without having to program **unsat** and **taut** and makes it possible for very efficient existing implementations of tests for unsatisfiability and tautology to be used.

The procedure **simp** is defined as follows.

```
procedure simp($A, I$);
  if unsat($A|_I$) $=$ true then return $\neg d_1 \vee \neg d_2 \vee \ldots \vee \neg d_n$;
    where $D = \{d_1, d_2, \ldots, d_n\}$ is a subset of $I$ such that $A|_D$ is unsatisfiable
  else if taut($A|_I$) $=$ true then return true;
  else
    let $y$ be some free variable in $A$ such that neither $y$ nor $\neg y$ occurs in $I$;
    let $A_1$ be simp($A, I \cup \{y\}$);
    if $\neg y$ does not occur in any clause of $A_1|_I$ and $A_1|_I$ is unsatisfiable
      then return $A_1$; fi;
    let $A_2$ be simp($A, I \cup \{\neg y\}$);
    if $y$ does not occur in any clause of $A_2|_I$ and $A_2|_I$ is unsatisfiable
      then return $A_2$; fi;
    return $A_1 \wedge A_2$;
  fi
end simp;
```

The two conditional statements involving y and $\neg y$ are optimizations that may be omitted. Now, **simp**(A) is defined for a Boolean formula A to be **simp**$(A, \{\})$, that is, the result returned when **simp**(A, I) is called with I equal to the empty set. If A has no free variables, then **simp**(A) is either **true** or **false**. When **simp**(A, I) returns a formula A', then it is said that the bound variables of A have been eliminated from A to produce A'. The procedure **simp** can be called on a formula without bound variables.

Theorem 3 *The procedure* **simp**$(A, \{\})$ *returns a set S of clauses such that $S \equiv A$.*

Proof. We show that for any interpretation I of the free variables of A, $I \models A$ iff $I \models S$.

Suppose $I \not\models A$. Consider the set of $J \subseteq I$ such that **simp**(A, J) is called during the execution of **simp**$(A, \{\})$. Since $I \not\models A$, $A|_I =$ **false** hence $A|_J =$ **false** for some such J. For some such J, **unsat**$(A|_J)$ will return **true**. When **simp**(A, J) is called for this J, a clause D will be returned such that $J \not\models D$ and $I \not\models D$. Such a D will be a clause in S, so $I \not\models S$ as well.

Suppose $I \models A$. Consider again the set of $J \subseteq I$ such that **simp**(A, J) is called during the execution of **simp**$(A, \{\})$. Since $I \models A$, $A|_I =$ **true** hence for no such J, $A|_J =$ **false**. Thus no clause D will be returned such that $I \not\models D$. Since S has no such clauses D, $I \models S$ as well. q.e.d.

The procedure **sat** is a faster version of **simp** that applies when the free variables of A are existentially quantified. This quantification means looking for one interpretation that makes A **true**. In this case, the procedure can be stopped as soon as one such interpretation is found, and it is not necessary to continue looking for more.

The procedure **sat** tests if formulae A of the form $\exists Y[B]$ are tautologies, assuming that all free variables in B are mentioned in Y, and is defined as follows:.

```
procedure sat($\exists Y[B]$);
  if unsat($B$) = true then return false;
  else if unsat($B$) = false then return true;
  else
    let $y$ be some free variable in $B$;
    let $B_1$ be sat($\exists Y[B|_y]$);
    if $B_1$ is true then return true fi;
    let $B_2$ be sat($\exists Y[B|_{\neg y}]$);
    if $B_2$ is true then return true fi;
    return false;
  fi
end simp;
```

If B is satisfiable, then **sat**$(\exists Y[B])$ will return **true**, else **sat**$(\exists Y[B])$ will return **false**. Equivalently, if $\exists Y[B]$ is a tautology, then **sat**$(\exists Y[B])$ will return **true**, else **sat**$(\exists Y[B])$ will return **false**. The procedure **sat** has an advantage over **simp** in that **sat** may stop sooner, and therefore take less time. However, **sat** cannot be called in as many cases as **simp** can. The procedure **sat** can be called on a formula in which the list X or Y of variables may be empty.

The procedure **sat** can be extended to formulae of the form $\forall X[B]$ by calling **sat** on $\exists X[\neg B]$. If **sat**($\exists X[\neg B]$) returns **false**, then **sat**($\forall X[B]$) returns **true**; if **sat**($\exists X[\neg B]$) returns **true**, then **sat**($\forall X[B]$) returns **false**. This is justified by the fact that $\forall X[B]$ is equivalent to $\neg\exists X[\neg B]$. Equivalently, if B is not falsifiable (a tautology), then **sat**($\forall Y[B]$) will return **true**, else **sat**($\forall Y[B]$) will return **false**.

Theorem 4 *The procedure* **sat**(A) *returns* **true** *iff A is satisfiable, where A has no free variables and is of the form $\exists Y[B]$.*

Proof. Suppose A is satisfiable. Then there is an interpretation I of the variables Y such that $I \models B$. Eventually **sat**($\exists Y[B|_J]$) will be called on some $J \subseteq I$ such that $B|_J$ is **true**. For some such J, **unsat**$(B|_J)$ will return **false**. Therefore **sat**($\exists Y[B|_J]$) will return **true**, so **sat**(A) will return **true**.

Suppose A is unsatisfiable. Then for no interpretation I of the variables Y, $I \models B$. Thus **sat**($\exists Y[B|_J]$) will never be called on some $J \subseteq I$ such that $B|_J$ is **true**. This implies that **unsat** will never return **false**. Therefore **sat** will return **false**. q.e.d.

The tests **unsat** and **taut** (these tests are called within **sat** and **simp**) can be done using an arbitrary decision procedure for quantified Boolean formulae. These procedures permit the procedures **simp** and **sat** to avoid some testing of quantified Boolean formulae in some cases. The procedures **simp** and **sat** can be called in two modes, inner mode and free mode. In inner mode, it is assumed that whenever **simp**(A, I) or **sat**(A) is called, then A is of the form $\exists X[B(X,Y)]$ or $\forall X[B(X,Y)]$ where B contains no quantifiers. Every quantified Boolean formula will contain at least one such subformula, so inner mode is sufficient to handle any quantified Boolean formula. In inner mode, whenever **unsat**(A) or **taut**(A) is called, and A has no free variables, then A is of the form $\exists X[B(X)]$ or $\forall X[B(X)]$ where B is an unquantified Boolean formula, that is, it contains no quantifiers. The formula $\exists X[B(X)]$ is equivalent to **true** if and only if B is satisfiable, otherwise it is equivalent to **false**. The formula $\forall X[B(X)]$ is equivalent to **true** if and only if $\neg B(X)$ is unsatisfiable, otherwise it is equivalent to **false**. Thus in all cases, **unsat** and **taut** can be implemented using a satisfiability test for unquantified Boolean formulae if inner mode is used. Many methods are known for testing if unquantified Boolean formulae are satisfiable, including DPLL for formulae in clause form. These tests are typically applied to an entire formula at once to determine whether the whole formula is satisfiable. The method QSAT works in smaller steps, which can make the entire process significantly more efficient. In free mode, the formula $B(X,Y)$ may contain Boolean quantifiers. In this case, a procedure for satisfiability of arbitrary quantified Boolean formulae can be used. One possibility for this is to call **simp** or **sat** recursively on the formula B. Also, these successive calls to **unsat** and **taut** are permitted to have memory, in the sense that earlier calls may record information about A that can make later calls more efficient.

The part of the procedure **simp** that determines D can be done by **unsat**, as well; it suffices to return $\neg d_1 \vee \neg d_2 \vee \ldots \vee \neg d_n$ where $\{d_1, d_2, \ldots, d_n\} = I$ when $\mathbf{unsat}(A|_I)$ returns **true**. It is also possible to describe procedures that return smaller formulae, by noting which of the d_i are really necessary for showing that $A|_I$ is unsatisfiable. For correctness of the **simp** procedure, it is only necessary that some such D be returned when $\mathbf{unsat}(A|_I)$ returns **true**. For smaller I, if some such D is returned, it can make the procedure **simp** more efficient, but it is not necessary for correctness.

The overall procedure for simplifying a quantified Boolean formula F according to QSAT is to find a sub-formula A of F or a formula equivalent to F to which the procedure **simp** or **sat** can be applied. Such a formula A is then replaced by A', A' being $\mathbf{simp}(A)$ or $\mathbf{sat}(A)$ in F to obtain a simpler formula F' equivalent to F. This procedure can be repeated on F' in turn any number of times, until one obtains a formula that can be tested for satisfiability or tautology by some other means, or else one may obtain simply **true** or **false**.

An example of the operation of the procedure **simp** will now be given. Suppose that A is the formula $\exists z[(x \vee y \vee z) \wedge (x \vee y \vee \neg z) \wedge ((\neg x \vee \neg y \vee z) \wedge (\neg x \vee \neg y \vee \neg z)]$. Thus Y is the list $\{x, y\}$ of variables. When $\mathbf{simp}(A, I)$ is called with I equal to $\{\}$, the empty set, it is seen that that $A|_I$ is not unsatisfiable, nor is it a tautology. I interpreted as a conjunction is **true**, so that $A|_I$ is equivalent to A, which is neither a tautology nor unsatisfiable. The next step is to pick a variable in Y, say x, and call $\mathbf{simp}(A, \{x\})$. This in turn will find that $A|_x$ is not unsatisfiable, nor is $A|_x$ a tautology. The formula $A|_x$ is obtained by replacing x by **true** and simplifying; this yields $\exists z[(\mathbf{true} \vee y \vee z) \wedge (\mathbf{true} \vee y \vee \neg z) \wedge (\neg \mathbf{true} \vee \neg y \vee z) \wedge (\neg \mathbf{true} \vee \neg y \vee \neg z)]$, which simplifies to $\exists z[(\neg y \vee z) \wedge (\neg y \vee \neg z)]$. So the other variable in I, namely y is next picked, and $\mathbf{simp}(A, \{x, y\})$ is called. $A|_{\{x,y\}}$ is now found to be unsatisfiable. The next task is to find a subset D of $\{x, y\}$ such that $A|_D$ is unsatisfiable; in this case, the only subset that works is $\{x, y\}$. Thus $\{\neg x, \neg y\}$ (representing the clause $\neg x \vee \neg y$) is returned as the value of $\mathbf{simp}(A, \{x, y\})$. Now $\mathbf{simp}(A, \{x, \neg y\})$ is called. $A|_{\{x,\neg y\}}$ is found to be satisfiable. Also, $A|_{\{x,\neg y\}}$ is **true**, which is a tautology. Thus, **true** is returned. The call to $\mathbf{simp}(A, \{x\})$ then returns the conjunction of these results, which is $(\neg x \vee \neg y) \wedge \mathbf{true}$, or, $\neg x \vee \neg y$. Now $\mathbf{simp}(A, \{\neg x\})$ is called which in turn will call $\mathbf{simp}(A, \{\neg x, y\})$ and $\mathbf{simp}(A, \{\neg x, \neg y\})$. The former returns **true**, and the latter returns $\{x, y\}$, representing the clause $x \vee y$. Thus the call to $\mathbf{simp}(A, \{\neg x\})$ returns the conjunction of these results, which is $x \vee y$. Finally, the call to $\mathbf{simp}(A, \{\})$ returns the conjunction of $\neg x \vee \neg y$ and $x \vee y$, which is $(\neg x \vee \neg y) \wedge (x \vee y)$. It is indeed true that $(\neg x \vee \neg y) \wedge (x \vee y)$ is equivalent to $\exists z[(x \vee y \vee z) \wedge (x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee z) \wedge (\neg x \vee \neg y \vee \neg z)]$.

An optimization to QSAT is to speed up the satisfiability testing by noting when a subformula can be expressed as a Boolean combination of two other formulae not sharing free variables. Suppose the procedure **simp** or **sat** is testing a subformula of the form $\exists x[\exists y[B(x, z) \wedge C(y, z)]]$ where z is the only free variable. A partial interpretation I may assign a truth value to z, say, **true**. This formula A then becomes $\exists x[\exists y[B(x, \mathbf{true}) \wedge C(y, \mathbf{true})]]$. Such a formula will be given to the procedure **unsat**. However, this formula can first be rearranged in an equivalence-preserving way to obtain the formula $\exists x[B(x, \mathbf{true})] \wedge \exists y[C(y, \mathbf{true})]$. This formula consists of the two subformulae $\exists x[B(x, \mathbf{true})]$ and $\exists y[C(y, \mathbf{true})]$ which do not share free variables. Thus **unsat** can be more efficient by testing satisfiability of these two subformulae separately and combining the results.

Thus it is useful to detect when a formula can be processed more efficiently by expressing it in terms of two other formulae not sharing free variables. In most cases, the recognition of such decompositions must be done by the user. In the procedures **unsat** and **taut** called by **simp** and **sat**, it is possible to test satisfiability or tautology of formulae of the form $B_1 \otimes B_2$, where B_1 and B_2 have no free variables in common and $\otimes$ is either conjunction or disjunction, by testing satisfiability of B_1 and B_2 separately and combining the results. Thus $B_1 \wedge B_2$ is satisfiable if both B_i are, and $B_1 \vee B_2$ is satisfiable if either B_i is. This idea can be incorporated into the **simp** and **sat** procedures to improve their efficiency. It is possible for the user of the **sat** and **simp** procedures to specify how such partitioning should take place: If B is a conjunction or disjunction of many formulae C_i, then the order of these subformulae may have to be rearranged in order to permit such a partitioning to take place. For this, the user can give some guidance as to two sets of variables that do not occur together in any formula C_i, and this can be used to reorder the formula into the form $B_1 \otimes B_2$ where each B_i is a conjunction or disjunction of many C_i, and B_1 and B_2 do not share free variables. The same technique can be applied if a formula is $C_1 \otimes C_2 \otimes \ldots \otimes C_n$ where $\otimes$ is either exclusive or or equivalence, because these operators, like conjunction and disjunction, are associative and commutative. Thus the user can obtain a formula of the form $B_1 \otimes B_2$ where B_1 and B_2 do not share free variables.

4 Applications to Symbolic Model Checking

The procedures **simp** and **sat** can also be used to detect when fixpoints of repetitive systems have been attained. A *repetitive formula* is a formula of the form $A(X_1, X_2) \wedge A(X_2, X_3) \wedge \ldots \wedge A(X_{n-1}, X_n)$. Let us refer to this formula by $A^n(X_1, X_2, \ldots, X_n)$. The formula $A(X, Y)$ is often of the form $\exists Z[A'(X, Z, Y)]$ for some formula A'. Such formulae are often encountered in symbolic model checking. For symbolic model checking applications, it is of interest to know for which n the formula $B(X_1) \wedge A^n(X_1, X_2, \ldots, X_n) \wedge C(X_n)$ is unsatisfiable. It is often useful to know that this formula is unsatisfiable for all n. Thus a test is presented that can verify that the formula $B(X_1) \wedge A^n(X_1, X_2, \ldots, X_n) \wedge C(X_n)$ is unsatisfiable for all n.

Let $B^n(X_n)$ be the formula

$$\exists X_1[\exists X_2[\exists X_3[\ldots[\exists X_{n-1}[B(X_1) \wedge A^n(X_1, X_2, \ldots, X_n)]\ldots]$$

It is of interest to know whether for all n, $B^n(X_n) \wedge C(X_n)$ is unsatisfiable. This is equivalent to the question whether $(B^1(X) \vee B^2(X) \vee \ldots \vee B^n(X) \vee \ldots) \wedge C(X)$ is unsatisfiable. This can be done by finding the smallest n such that $B^{n+1}(X)$ logically implies $B^1(X) \vee B^2(X) \vee \ldots \vee B^n(X)$; such an n must exist by properties of these formulae. One can test whether $B^{n+1}(X)$ logically implies $B^1(X) \vee B^2(X) \vee \ldots \vee B^n(X)$ by testing whether the formula $\neg B^{n+1}(X) \vee (B^1(X) \vee B^2(X) \vee \ldots \vee B^n(X))$ is valid, that is, whether $\forall X[\neg B^{n+1}(X) \vee B^1(X) \vee B^2(X) \vee \ldots \vee B^n(X)]$ is a tautology. This is a quantified Boolean formula which can be simplified to **true** using above methods iff it is a tautology. When such an n has been found, then one can test whether $B^j(X) \wedge C(X)$ is satisfiable for any non-negative integer j not larger than n; if not, then one knows that for all n, $B^n(X) \wedge C(X)$ is unsatisfiable. Otherwise, for some n, $B^n(X) \wedge C(X)$ is satisfiable. One can test whether $B^j(X) \wedge C(X)$ is satisfiable by testing whether $\exists X[B^j(X) \wedge C(X)]$ is a tautology.

5 Complexity bound for QSAT

It is possible to bound the complexity of QSAT on a special class of quantified Boolean formulae.

Definition 5.1 *A quantified Boolean formula A is k* width bounded *if every subformula of A has at most k free variables.*

Theorem 5 *If* QSAT *is applied to a formula A that has no free variables and is k width bounded, and* QSAT *is called in inner mode, then the time taken is linear in the length of A and exponential in k.*

Proof. QSAT will perform a sequence of calls to **simp**, which will be called first on the innermost formulae of the form $\forall X[B]$ or $\exists X[B]$. Thus whenever **simp** is called on a formula, it will be of the form $\forall X[B]$ or $\exists X[B]$ where all quantifiers will already have been eliminated from B by previous calls to **simp**. Thus B will be of the form $(B_1\ op_1\ B_2\ op_2\ \cdots\ op_{n-1}\ B_n)$ where the B_i are unquantified formulae with at most k variables and op_i are binary Boolean connectives (if negations have been pushed to the bottom). Each B_i will either be an unquantified subformula of A or an unquantified conjunctive normal form formula obtained by previous calls to **simp**. Each call to **simp** will eliminate at least one variable and will return an unquantified conjunctive normal form formula having at most $k-1$ free variables. Also, each call will take time $\mathcal{O}(c^k)|B|$ for some constant c, where $|B|$ is the length of B, because of backtracking on the variables of B. The total time is bounded by $\sum_B \mathcal{O}(c^k)|B|$, or by $\sum_B \mathcal{O}(c^k)\sum_i |B_i|$ because $|B| \leq 2\sum_i |B_i|$. Let $\sum'_{B,i}$ denote the sum over the formulae B_i that have not been affected by prior calls to **simp**, and thus were subformulae of A, and let $\sum''_{B,i}$ denote the sum over the formulae B_i that are obtained by previous calls to **simp**. Note that these subformulae B_i of A referred to in $\sum'_{B,i}$ will be disjoint, because no occurrence of such a subformula will appear inside another such occurrence. Thus $\sum'_{B,i}|B_i| \leq |A|$. Also, we can assume that when **simp** returns a result, it is in conjunctive normal form with tautologous and duplicate clauses deleted. This implies that each such formula has at most 3^k clauses and each clause has at most k literals. Therefore each such formula B_i returned by **simp** satisfies $|B_i| \leq b^k$ for some constant b. Also, the total number of formulae B_i is bounded by $|A|$, because each such B_i corresponds to a subformula of A and contributes to at most one B in the sum. Thus $\sum_{B,i} 1 \leq |A|$. Therefore$\sum_B \mathcal{O}(c^k)\sum_i |B_i| = \mathcal{O}(c^k)\sum_{B,i}|B_i| \leq \mathcal{O}(c^k)(\sum'_{B,i}|B_i| + \sum''_{B,i}|B_i|) \leq \mathcal{O}(c^k)(|A| + \max(|B_i|)(\sum''_{B,i} 1))$ $= \mathcal{O}(c^k)(|A| + b^k|A|) \leq \mathcal{O}((bc)^k)|A|$. q.e.d.

For a fixed k, this time bound is linear in $|A|$. Such formulae arise naturally in the computation of fixed points for symbolic model checking.

Theorem 6 *Suppose the formulae A, A', B, and C are as in section 4. Suppose the formulae A', B, and C are unquantified Boolean formulae having at most k free variables. Then the formulae $\forall X[\neg B^{n+1}(X) \vee B^1(X) \vee B^2(X) \vee \ldots \vee B^n(X)]$ and $\exists X[B^j(X) \wedge C(X)]$ are k-width bounded quantified Boolean formulae, or may be made so by moving quantifiers in a linearly computable and equivalence-preserving manner.*

Proof. The only part that does not follow immediately from the definitions is that the formula $B^n(X_n)$, defined as

$$\exists X_1[\exists X_2[\exists X_3[\ldots[\exists X_{n-1}[B(X_1) \wedge A^n(X_1, X_2, \ldots, X_n)]\ldots],$$

has to be expressed instead as

$$\exists X_{n-1}[\ldots \exists X_2[\exists X_1[B(X_1) \wedge A(X_1, X_2)] \wedge A(X_2, X_3)] \wedge \ldots \wedge A(X_{n-1}, X_n)]$$

This manner of expressing the formula guarantees that all subformulae have at most k free variables. q.e.d.

The length of the formula B^i is linear in i, so the length of the formula $\forall X[\neg B^{n+1}(X) \vee B^1(X) \vee B^2(X) \vee \ldots \vee B^n(X)]$ is quadratic in n. The preceding theorem then shows that QSAT can decide the satisfiability of this formula in time quadratic in n. However, the formulae B^i share many subformulae. If this is taken into account, and whenever $\mathbf{simp}(F)$ is called, all occurrences of F are replaced by $\mathbf{simp}(F)$, then the time can be reduced to be linear in n. This also requires the formula $\forall X[\neg B^{n+1}(X) \vee B^1(X) \vee B^2(X) \vee \ldots \vee B^n(X)]$ to be represented economically so that common subformulae are only stored once.

6 Complexity Bounds for Qsat$_{\text{CNF}}$ and BDDs

We now present some complexity bounds for QSAT$_{\text{CNF}}$, as well as some results for BDDs. In harmony with our previous use of the term, we define a *cut* of a set S of clauses as a selection of a subset Y of the variables of S as bound variables and the remainder as free. If the variables of S are linearly ordered and the bound variables Y are selected as all those larger than a given variable x in S, then this cut is said to be a cut *at* x, and also an *ordered cut.* The *cost* of a cut Y of S is the number of variables x of S not in Y such that for some variable y in Y, and some clause C in S, x and y both appear in S. The clause C is said to *cross* such a cut Y. The cost of the cut Y corresponds to the number of variables in the set X mentioned above in the description of QSAT$_{\text{CNF}}$. The *average cost of a cut* of S relative to an ordering on variables is the average of the costs of the ordered cuts for this ordering. The *maximum cost of a cut* relative to a variable ordering is defined similarly. This maximum cost of a cut is also called the *cut width* or *width* of the circuit.

In [18], the complexity of automatic test pattern generation (ATPG) was studied. The ATPG problem was reduced to a propositional satisfiability problem. It was shown that the complexity of the SAT problem can be exponentially bounded with respect to the cut width of a circuit. It was also shown that many practical problems have small cut widths. Further, good variable orderings for these problems were found automatically by a heuristic for reducing the cut width. This suggests that for problems having small cut widths, finding good variable orderings is often easy, even without an understanding of the problem structure. The method of [18] is not restricted to conjunctive normal form formulae but requires expensive rewriting at each variable assignment.

Theorem 7 *The time taken by* QSAT$_{\text{CNF}}$ *on sets S of clauses having n linearly ordered variables and maximum costs of ordered cuts w and in which a bounded number d of variables are eliminated in each call to* **simp** *is $\mathcal{O}(n \cdot 2^w)$.*

Proof. The time is dominated by the calls to **taut** and **unsat**. Calls to **taut** are very fast. The number of calls to **unsat** is bounded by 2^c and each call takes time at most b^d for some constant b. The time on each cut is then at most $2^c \cdot b^d$. Each cut eliminates at least one variable, so at most n such cuts must be processed and the total time is at most $n \cdot 2^c \cdot b^d$. Since $c \leq w$ and b and d are bounded, the total time is $\mathcal{O}(n2^w)$. q.e.d.

Corollary 6.1 *The time taken by* Qsat$_{\text{CNF}}$ *on sets S of clauses having n linearly ordered variables and bounded maximum costs of ordered cuts and in which a bounded number d of variables are eliminated in each call to* **simp** *is $\mathcal{O}(n)$.*

Corollary 6.2 *The time taken by* Qsat$_{\text{CNF}}$ *on sets S of clauses having n linearly ordered variables and $\mathcal{O}(\log(n))$ maximum costs of ordered cuts and in which a bounded number d of variables are eliminated in each call to* **simp** *is polynomial.*

Typically b is much smaller than 2, because only one model needs to be found in the bound variable region (Y) but all models need to be found in the free variable region (X). Thus the cost of the cuts has a much larger influence on the running time of Qsat$_{\text{CNF}}$ than the number d of variables that are eliminated.

In general, long and thin circuits correspond to sets S of clauses with small costs of cuts, assuming that the circuit is laid out horizontally and variables are ordered left to right according to the positions of the corresponding wires in the circuit. BDDs with a good ordering typically do well on such circuits, but DPLL-type methods often do not. A vertical line through such a circuit corresponds to a cut, and the number of wires that cross the line is proportional to the number of clauses that cross the cut. In a long and thin circuit, this number of wires should be small, so the cost of the cut should be small. Sometimes the cost of a cut can be small even if many wires cross the vertical line. This corresponds to the case when many clauses cross a cut but only a few variables are mentioned in these clauses. Since adders and some other hardware circuits are long and thin, it is to be expected that Qsat$_{\text{CNF}}$ will be efficient on them if the variables are properly ordered. If the variables are improperly ordered, the costs of cuts can be very large, and Qsat$_{\text{CNF}}$ can be very slow.

A complexity analysis based on cut can also be applied to BDDs [20,12]. Given a set of clauses S, we define a graph G. Each node in G represents a variable in S. There is an edge between two nodes in G iff there is a clause in S containing the variables corresponding to the two nodes. The cut and the cost of the cut can defined based on the graph structure. If the set of clauses S are generated by a structural translation from a circuit, then G is essentially an undirected graph representation of the circuit with the exception that G contains edges between inputs(outputs) of a gate. If each gate in a circuit can only have constant fanin and fanout, adding the extra edges in G does not affect the complexity analysis. A circuit can also be represented as a directed graph, where each node represents a wire and each directed edge represents the connection from an input to an output of a gate. Based on this directed graph structure, [12] defined the concepts of forward width and reverse width of a circuit by distinguishing the directions of edges crossing a cut. It was shown that if a circuit computes function f, the size of OBDD that represents f is $\mathcal{O}(n2^{w_f 2^{w_r}})$, where n is the number of inputs of the circuit and w_f is the forward width and w_r is the reverse width of the circuit. For example, testing whether the output of a circuit can be **true** can be represented as a Boolean satisfiability problem. A naive solution is to build the OBDD for the circuit and see if there is a path to 1 in the OBDD. This approach is doubly exponential in the cut width of the circuit, while QSAT$_{\text{CNF}}$ is single exponential. Intuitively, building an OBDD that represents the functionality of the circuit is an overkill for the problem, as one assignment of input variables that sets the output to be **true** is sufficient.

Some examples are known [14] for which DPLL-type satisfiability algorithms are faster than BDDs. One can imagine stringing together a long sequence of such problems; such a sequence could not be solved by a BDD due to the difficulty of each problem, and it could not be solved by a DPLL-type algorithm, either, because such algorithms typically perform poorly on long, thin circuits. However, QSAT$_{\text{CNF}}$ could solve such a problem by solving the subproblems in sequence, one by one.

In fact, it is possible to give an explicit example for which QSAT$_{\text{CNF}}$ is exponentially faster than OBDDs.

Definition 6.3 *The* hidden weighted bit function *hwb is defined by*

$$hwb(x_1, \ldots, x_n) = x_{sum}$$

where the x_i are bits and $x_0 = 0$ and sum is the number of i such that x_i is 1.

Theorem 8 *Consider the problem of verifying that for all $x_1, \ldots, x_n$,*

$$(x_1 \vee x_2 \vee \ldots \vee x_n) \supset hwb(x_1, \ldots, x_n, 0) = hwb(1, x_1, \ldots, x_n)$$

where n is $2^k - 2$ for some k. Then this problem, with a suitable variable ordering, can be solved by QSAT$_{\text{CNF}}$ *in polynomial time, but for OBDDs with any variable ordering it takes exponential time.*

Proof. It is known ([5], page 78) that any OBDD for the hidden weighted bit function is of exponential size, regardless of the variable ordering. This implies that OBDDs will take exponential time to determine whether $(x_1 \vee x_2 \vee \ldots \vee x_n) \supset hwb(x_1, \ldots, x_n, 0) = hwb(1, x_1, \ldots, x_n)$. The reason for this is that by setting some x_i to 0, we can compute $hwb(x_1, \ldots, x_{n/2})$ from $hwb(x_1, \ldots, x_n, 0)$, but $hwb(x_1, \dots, x_{n/2})$ requires an exponential size OBDD. Therefore the OBDD for $hwb(x_1, \dots, x_n, 0)$ must be even bigger.

However, the most natural representation of the function $hwb(x_1, \dots, x_n, x_{n+1})$ by a set of clauses corresponding to a bounded fan-in circuit has cut width $\mathcal{O}(\log(n))$, as follows: First, one can compute the sum of the x_i by a tree of binary adders. This can be done from left to right, by adding x_1 and x_2, adding the result to x_3, et cetera. It can also be done from right to left, adding x_n and x_{n-1}, adding the result to x_{n-2}, et cetera. Another way is to compute the sum by a nearly balanced binary tree of adders. Any of these three possibilities will give the bound we desire. Next, the most straightforward way to select the correct variable x_{sum} is to use a barrel-shifter like structure. This involves a binary tree in which one bit of the sum is used at each node to determine whether to propogate the left or right child upwards. It is also necessary to represent the formula $(x_1 \vee x_2 \vee \dots \vee x_n)$. To do this with binary fan-in gates requires a tree of disjunctions. The disjunctions can be associated to the left or to the right, as the additions were, or computed in a nearly balanced binary tree structure.

A suitable ordering on these variables results in a logarithmic cut width, and this ordering can be found easily. First, each adder requires only $\mathcal{O}(\log(n))$ variables. The general way to order variables is assign each variable x a horizontal position $h(x)$ corresponding to a reasonable layout of the circuit and order the variables by their horizontal positions so that $x < y$ if $h(x) < h(y)$. The inputs can be ordered $x_1, x_2, ..., x_n$ from left to right, at equally spaced intervals. Then any variable that depends on two others can be given a position halfway between them. For example, if s_1 is the sum of x_1 and x_2 then s_1 would be halfway between x_1 and x_2, $h(s_1) = (h(x_1) + h(x_2))/2$. Other similar ordering schemes work just as well. This gives logarithmic cut width if the operations are associated to the left or right or in a nearly balanced binary tree structure, as suggested above. If a gate has a large fanout, then it has a large effect on the cut width, so it may be necessary to consider a variety of positions for the output variable of such a gate; in general, outputs of gates with large fanout should be made small in the ordering to reduce the cut width. One can also simply try all positions for such variables and pick the one that minimizes the cut width.

This approach gives logarithmic (or even constant) cut widths for all parts of the circuit. The only gates with large fanout are those corresponding to bits of the sum, and if these are ordered large in the ordering, then the cut width is logarithmic for these also. Thus the total cut width is $\mathcal{O}(\log(n))$ and by corollary 6.2, QSAT$_{\text{CNF}}$ can test satisfiability of a set of clauses representing the given problem in polynomial time. q.e.d.

This result should be seen as a theoretical result, that there exist circuits that are hard for BDD's but easy for $\text{Q}\textsc{sat}_{\text{CNF}}$. If the circuit for computing $hwb(x_1, \ldots, x_n, x_{n+1})$ were not natural for the Boolean function being computed, the preceding theorem might not be as interesting. However, the circuit seems to be the most straightforward method of computing *hwb*. As for the ordering, there are a number of good and fully automatic heuristics for ordering variables to reduce the cut width, such as those in [18], so it does not seem difficult to find an ordering as required by the theorem. For a human having understanding of the structure of the circuit, the ordering is also very natural. Finally, the use of mathematically defined functions such as *hwb* to obtain benchmarks is standard practice, as for example the use of the pigeonhole problems, known to be unsatisfiable, to obtain benchmarks for propositional satisfiability checkers.

The same bound applies also to the original version of Davis and Putnam's method [6] which uses ordered resolution. In fact, many of the complexity bounds and comments about choosing variable orderings in this paper apply equally well to ordered resolution. However, DPLL[7] is often much faster than ordered resolution, so one would expect $\text{Q}\textsc{sat}_{\text{CNF}}$ to be faster in many cases as well.

BDDs can also be used for satisfiability testing in other ways that may be faster. Let B be the Boolean expression $(x_1 \vee x_2 \vee \ldots \vee x_n) \supset hwb(x_1, \ldots, x_n, 0) = hwb(1, x_1, \ldots, x_n)$. We can test if B is satisfiable by building a BDD for B and testing if the BDD is 0 since an unsatisfiable formula is identically false and has a BDD of 0. Since B is unsatisfiable, any BDD for B will be 0, which is very small (constant size). However, in order to construct this BDD, intermediate BDDs need to be built that may be larger. The complexity of constructing this BDD may depend on the manner in which these intermediate BDDs are built. It is also possible to incorporate the intermediate signals in a circuit for computing *hwb* into B when testing its satisfiability; this might lead to a faster method.

7 Qsat$_{\text{CNF}}$ implementation

Recall that an unquantified Boolean formula S can be viewed as the quantified Boolean formula $\exists Z[S]$ where Z includes all the free variables of S. Applying $\text{Q}\textsc{sat}$ to $\exists Z[S]$ gives a procedure to test whether the unquantified Boolean formula S is satisfiable. If $\text{Q}\textsc{sat}(\exists Z[S])$ returns **true** then S is satisfiable, else S is unsatisfiable. Suppose S is a clause form (conjunctive normal form) formula. Let Y be a nonempty subset of the variables Z and let X be the remaining variables. Since S is a conjunction of clauses, S may be written as $S_1 \wedge S_2$ where S_2 is a conjunction of the clauses mentioning variables in Y and S_1 is a conjunction of the remaining clauses in S. Then $\exists Z[S] \equiv \exists X[\exists Y[S_1 \wedge S_2]]$

$\equiv \exists X[S_1 \wedge \exists Y[S_2]] \equiv \exists X[S_1 \wedge \mathbf{simp}(\exists Y[S_2])]$. Let the conjunctive normal form formula S_2' be $\mathbf{simp}(\exists Y[S_2])$. Then the formula $S_1 \wedge S_2'$ is a conjunctive normal form formula that has fewer free variables than S. Therefore the same procedure can be applied to this formula in turn, until all variables are eliminated. In this way we obtain the restriction $\text{Q}\textsc{sat}_{\text{CNF}}$ of Qsat to clause form formulae S. The satisfiability problem for quantified Boolean formulae is PSPACE-complete, but that for quantifier-free conjunctive normal form formulae is NP-complete, so the latter is probably much easier. Note that the full **simp** procedure is not needed to implement $\text{Q}\textsc{sat}_{\text{CNF}}$. Instead, one only needs the restriction $\mathbf{simp}_{\text{CNF}}$ of **simp** to conjunctive normal form formulae. In fact, only $\mathbf{simp}_{\text{CNF}}$ was implemented, and not the full **simp** procedure.

To further illustrate the connection between Qsatand $\text{Q}\textsc{sat}_{\text{CNF}}$, we show how Qsat could be implemented using only $\mathbf{simp}_{\text{CNF}}$. If A is an arbitrary quantifier-free formula, then $\mathbf{simp}(\exists Z[A]) \equiv \mathbf{simp}_{\text{CNF}}(\exists Z[\exists W[A']])$ where A' is a conjunctive normal form formula and $\exists W[A']$ is equivalent to A. So for formulae of the form $\exists Z[A]$ where A is quantifier-free, **simp** can be implemented on top of $\mathbf{simp}_{\text{CNF}}$ using a conjunctive normal form translator that has the option of adding extra existentially quantified variables W (as in the structure-preserving translation [21]). The structure-preserving translation permits any Boolean formula to be put in conjunctive normal form in linear time. Also, $\mathbf{simp}_{\text{CNF}}$ can be extended to formulae of the form $\forall Z[A]$ by duality where A is quantifier-free. If d indicates the dual, then $\mathbf{simp}_{\text{CNF}}(\forall Z[A]) = (\mathbf{simp}_{\text{CNF}}(\exists Z[A^d]))^d$. These techniques permit **simp** to be implemented on top of $\mathbf{simp}_{\text{CNF}}$. In this way, Qsat could be implemented on top of $\mathbf{simp}_{\text{CNF}}$ by always choosing to apply **simp** to a subformula of the form $\exists Z[A]$ or $\forall Z[A]$, where A is quantifier-free, as in inner mode.

A version of $\text{Q}\textsc{sat}_{\text{CNF}}$ was implemented by Bill Yakowenko [22] in C, and on certain problems it did better than SATO. This implementation used fast data structures to select good cuts rapidly. However, it did not have SATO's sophisticated methods for solving the basic satisfiability problem.

The version of $\text{Q}\textsc{sat}_{\text{CNF}}$ we implemented and tested assumes that the variables Z are linearly ordered, and that the variables Y are chosen as the d largest variables in this ordering, for some d. With notation as above, with S as a set of clauses, we obtain the following procedure:

```
procedure $\text{QSAT}_{\text{CNF}}(S)$;
   if $S$ contains the empty clause then return false
   else if $S$ has no variables then return true;
   let $n$ be the number of variables in $S$;
   choose a number $d$ of variables to eliminate from $S$,
      where $0 < d \leq n$;
   let $Y$ be the set of the $d$ largest variables in $S$;
   let $T$ be the set of clauses in $S$ mentioning variables in $Y$;
   let $X$ be the variables in $T$ not included in $Y$;
   return $\text{QSAT}_{\text{CNF}}(S \backslash T \cup \textbf{simp}(\exists Y[T]))$;
end $\text{QSAT}_{\text{CNF}}$;
```

The procedure **simp** eliminates the variables Y from T and returns a set T' of clauses such that $T' \equiv \exists Y[T]$. The clauses in T' are called *output lemmas*, or just lemmas, of QSAT_{CNF} in the following description. These lemmas mention only variables in X and are written on intermediate output files during the execution of QSAT_{CNF}. Recall that **simp** performs a Davis and Putnam (DPLL[7])-like backtracking search of partial interpretations of the free variables X in T, generating all models in the process, and repeatedly calls **unsat** on subproblems generated during this search. **unsat** is a Davis and Putnam (DPLL) procedure that stops at the first model. SATO has been modified so that both **simp** and **unsat** are performed by one call to SATO. This modified SATO procedure explores the X variables first, then the Y variables, in a Davis and Putnam-type search. During the exploration of the X variables, all models must be generated, but during the exploration of the Y variables, only one model need be generated. The calls to **unsat** correspond to the portion of the execution of SATO that is spent searching the bound variables Y of T. Lemmas generated during this part of the search are not output, but are called *internal lemmas* of QSAT_{CNF}.

QSAT_{CNF} was implemented by the first author in C on top of the significantly modified version of SATO3.2. QSAT_{CNF} calls this modified version of SATO to implement the procedure **simp**, and uses file i/o to communicate the output lemmas between calls to **simp**. Each such call is called a *round* of QSAT_{CNF}. The implementation only works for clause form formulae, even though QSAT itself is defined more generally.

One wants to make the backtracking during the search of the X variables efficient by a purity test or something similar. It turns out that this is not sound. However, if at a given point in the search, neither a variable x in X

nor its negation appears in any active clauses, then only one choice for x need be tried. This can significantly improve efficiency. In the Y region, a purity test as usual in DPLL can be used.

As an example of the unsoundness of the purity rule, suppose that S is $p \wedge (\neg p \vee \neg q) \wedge (q \vee r) \wedge \neg r$. Suppose the variable r is eliminated. Then $\text{QSAT}_{\text{CNF}}(S \backslash T \cup \mathbf{simp}(\exists Y[T]))$ is called, where T is $(q \vee r) \wedge \neg r$ and $S \backslash T$ is $p \wedge (\neg p \vee \neg q)$. Note that q is pure in T, but one cannot delete the clause containing q from T, because this will result in $\mathbf{simp}(\exists Y[T])$ returning **true** and $\text{QSAT}_{\text{CNF}}(S)$ returning **true**. If no purity rule is applied, $\mathbf{simp}(\exists Y[T])$ returns q and $\text{QSAT}_{\text{CNF}}(S)$ returns **false**. However, q is not pure in S, so a modified purity rule could be applied that considered the whole clause set S and not just T.

The DPLL method used by SATO has been modified in the following ways, among others, in the QSAT_{CNF} implementation:

When splitting on a variable, QSAT_{CNF} chooses the variable and the truth value to maximize the number of satisfied clauses, subject to the restriction that variables in X have to be chosen first. This is similar to GRASP [23]. This strategy tends to find models sooner and also tends to generate fewer internal lemmas. This means that larger lemmas can be tolerated; in fact, QSAT_{CNF}'s default bound on lemma size is 100.

The output lemmas generated by the QSAT_{CNF} implementation may have unnecessary variables in them, because QSAT_{CNF} does not check that all literals in these lemmas are really needed. Eliminating these extra variables could significantly increase the efficiency of the QSAT_{CNF} implementation.

The QSAT_{CNF} implementation computes the average and maximum cost of a cut right away, so one can know quickly whether the problem is too hard. A good idea is simply to call a standard satisfiability tester in such a case.

One technique that can make QSAT_{CNF} more efficient is to reformulate the problem by splitting clauses; this entails replacing a clause $C_1 \cup C_2$ by the two clauses $C_1 \cup \{x\}$ and $C_2 \cup \{\neg x\}$ where x is a new variable, all variables in C_1 are smaller in the ordering than all variables in C_2, and x is ordered in between C_1 and C_2. Splitting can reduce the cost of cuts because each of the two resulting clauses will cross fewer cuts, and the number of variables in each clause is less.

8 Methods of reordering variables

Because the costs of cuts can depend on the ordering of the variables, we implemented a number of reordering routines which will now be described. In this section we consider general methods that can be applied without any special knowledge of the structure of the problem. In the next section we consider more specialized methods that can be applied under human guidance when the structure of the problem is known. It is probably best initially to use methods that take advantage of the structure of the problem, and then try to improve the ordering using general heuristics.

Consider the variables V to be ordered by a function $f\colon V \rightarrow \{1, \ldots, |V|\}$ mapping each variable x onto an integer $f(x)$, so that $x < y$ if $f(x) < f(y)$. We assume that some ordering of the variables is given initially. Define the *inverse* of an ordering to be the ordering in which the order of the variables is reversed.

One reordering method reduces the average cost of a cut by interchanges of variables. Such interchanges are continued until there is no further reduction in the average cost of a cut.

Another *lexicographic* method reduces the cost of the maximum cut, namely, the cut at the maximum variable, as much as possible, then does this for the cut at the next largest variable, and so on. This routine is very fast, and may be expressed by the following algorithm:

```
for($i$ = Max_atom; $i > 1$; $i$--){
   for($j = i$; $j > 0$; $j$--)
      $C_j$ = cost of cut at $i$ if $i$ and $j$ are exchanged;
   pick $j$ such that $C_j$ is minimal;
   exchange $i$ and $j$
}
```

The ordering heuristic used in all our tests is to try four orderings and pick the one that minimizes the maximum cost of a cut. These four orderings are:

(1) The original ordering.
(2) The inverse of the original ordering.
(3) The lexicographic ordering, applied to variables in the original ordering.
(4) The lexicographic ordering, applied to the variables in the inverse of their original ordering.

Note that the maximum cost of a cut in an ordering may differ considerably from the maximum cost of a cut in the inverse ordering. Also, ties in the lexicographic ordering are broken based on the original ordering of the variables, so the last two possibilities above may give different results.

If one has a good ordering on the input variables, this can be used to order all the variables in a problem. Such orderings are often needed in any case for BDD's. For example, the remaining variables can be ordered according to the ordering on the maximum or minimum input variable on which they depend. This is sometimes good for long, thin circuits like adders. Another good method is to use an ordering as illustrated by the hidden weighted bit function in section 6, in which variables are ordered by real number positions and the position associated with the output of a gate is the average of the positions of the inputs.

The effectiveness of such orderings depends on the manner in which the formula is expressed. Often a formula is translated to clause form using a "structure-preserving" translation in which new Boolean variables are introduced for subformulae of the original formula. When this is done, it is important to associate multiple conjunctions and disjunctions properly to make this ordering yield a smaller cost of cut.

For example, suppose we have a formula $A_1 \wedge A_2 \wedge \ldots \wedge A_n$ where the maximum input variable that A_i depends on is i. Then we should express this formula as $((A_1 \wedge A_2) \wedge A_3) \wedge \ldots$ and introduce new variables B_1, B_2, and B_3, such that

$$B_1 \equiv (A_1 \wedge A_2)$$

$$B_2 \equiv (B_1 \wedge A_3)$$

$$B_3 \equiv (B_2 \wedge A_4)$$

$$\ldots$$

since then the maximum input variable that B_i depends on is $i+1$. If this formula is expressed by associating to the right, by

$$B_1 \equiv (A_{n-1} \wedge A_n)$$

$$B_2 \equiv (A_{n-2} \wedge B_1)$$

$$\ldots$$

then all B_i depend on A_n, so ordering by the maximum input variable that the B_i depend on, does not order the B_i well. A survey of ordering methods for BDDs is given in section 2.5.2 of [5]. These orderings roughly correspond to the orderings implemented in $\text{Q}\textsc{sat}_{\text{CNF}}$, except that we did not implement the "sifting" approach which may give better results.

9 Choosing an ordering for particular examples

We now give some ordering techniques that depend on a detailed knowledge of the structure of the problem. Such methods are however fairly straightforward for a human to apply. We illustrate these techniques on the benchmarks on which $\text{Q}\textsc{sat}_{\text{CNF}}$ was run. For simplicity in discussing these examples, we assume that the Boolean variables are integers, so that we write i for the variable x_i; this is the convention used for DIMACS input.

We first discuss the maxmin example. This example expresses that

$$\max(a, b) \geq \min(a, b),$$

where a and b are n-bit binary numbers and max and min are defined by

$$\max(x, y) \equiv \texttt{if } x \geq y \texttt{ then } x \texttt{ else } y$$

$$\min(x, y) \equiv \texttt{if } x \geq y \texttt{ then } y \texttt{ else } x$$

To encode this in clause form, we let $a[i]$, $b[i]$, $max[i]$, and $min[i]$ be Boolean variables representing the i-th bits of a, b, $\max(a, b)$, and $\min(a, b)$, respectively. We also let $ge(a, b, i)$ be a variable signifying whether the i low-order bits of a are greater than or equal to the i low-order bits of b, and similarly for $ge(max, min, i)$. Finally, $ge(max, min)$ is a variable signifying whether $\max(a, b) \geq \min(a, b)$, and $ge(a, b)$ is a variable signifying whether $a \geq b$. Thus $ge(max, min) \equiv ge(max, min, n)$ and $ge(a, b) \equiv ge(a, b, n)$.

Now, the variables $ge(a, b, i)$, $ge(max, min, i)$, $\max(a, b, i)$, and $\min(a, b, i)$ can be defined in terms of each other and the variables $a[i]$, $b[i]$, as well as all these quantities with i replaced by $i-1$. In addition, $ge(max, min)$ and $ge(a, b)$ need to be used in defining these variables.

The simplest way to order these variables is to order them by i. Since there are six quantities, we could let $a[i]$ be $6 \cdot i$, $b[i]$ be $6 \cdot i + 1$, et cetera. The problem with this is that $ge(max, min)$ and $ge(a, b)$ would then have large values. Since $ge(max, min)$ and $ge(a, b)$ are related to so many other quantities, it is better to make them small. In this way, we finally obtain the following ordering:

$ge(max, min)$: 1	$ge(a, b, i)$: $6 \cdot i - 1$
$ge(a, b)$: 2	$max[i]$: $6 \cdot i$
$a[i]$: $6 \cdot i - 3$	$min[i]$: $6 \cdot i + 1$
$b[i]$: $6 \cdot i - 2$	$ge(max, min, i)$: $6 \cdot i + 2$

This is the ordering used in the "dpmaxmin" examples in the tables below. With this ordering, the dpmaxmin example has a very small average cost of a cut, which is 4 even for large n. Other orderings, however, have a much larger cost. The "maxmin" examples below are formalized with a different ordering and a structure-preserving translation. The maxmin examples can be solved quickly by BDDs.

We next consider the barrel shifter. This example has an n-bit register S holding a quantity indicating how far another register X is to be rotated. This is formalized in terms of time steps; at the first time step, the register X is either rotated one bit or zero bits, depending on whether $S[1]$ is 1 or 0. At the second time step, the register X is rotated either two bits or zero bits, depending on whether $S[2]$ is 1 or 0, and so forth.

The theorem to be proved is that the binary value of S gives the amount that the register X is rotated. This can be expressed as one statement, but it is much better to consider one bit of X at a time and verify that it gets shifted by the proper amount. Even better is to verify for each value of S separately, 2^n values in all, that the specified bit of X gets shifted the proper amount. With this encoding, this problem is easily solved by $\text{Q}\textsc{sat}_{\text{CNF}}$, but the problems are so trivial that they are not included in the following tables of runtimes. The version of this problem with all values of S considered at once for one bit of X is problem "barrel" below. For these problems, it is best to order the variables so that the bits of S occur early in the ordering, because they are related to so many other variables. The other variables, representing bits of X at various times, can be ordered in a natural manner, left to right.

We now discuss how to order clause sets having the structure of a nearly-balanced binary tree. This can be expressed by the set $\{\{x_i, \neg x_{2\cdot i}, \neg x_{2\cdot i+1}\} \mid 1 \leq i \leq n\}$ of clauses. If the variables are ordered $x_1 < x_2 < x_3 \ldots$, then the average cost of an (ordered) cut will be linear. This is because each clause $\{x_i, \neg x_{2\cdot i}, \neg x_{2\cdot i+1}\}$ contributes $i + 1$ to the sum of the costs of the cuts. (The cost is one for cuts from $i + 1$ to $2 \cdot i$ and two for the cut at $2 \cdot i + 1$.) If the variables are ordered $x_1 > x_2 > x_3 \ldots$ then the average cost of a cut is about twice as large, because then the given clause contributes $2 \cdot i + 1$ to the sum of the costs of the cuts. However, if the tree is ordered prefix, postfix, or infix, regarding x_i as a node and $x_{2 \cdot i}$ and $x_{2 \cdot i+1}$ as its children, then the average cost of a cut is $\mathcal{O}(\log(n))$. Each cut is crossed by at most $\log(n)$ clauses in the tree. There are about $\log(n)$ levels to the tree, and the probability of a clause at a given level crossing a cut is about one half. Therefore the average cost of a cut is $\mathcal{O}(\log(n))$. For these orderings, the average cost of a cut is about twice as large for prefix as for postfix, with the cost for infix being in the middle. We note that a factor of two in the average cost of a cut can make a tremendous difference in the efficiency of $\text{Q}\textsc{sat}_{\text{CNF}}$.

We next illustrate the importance of the variable ordering with these clause sets: $\{\{x_1, x_i, x_{i+1}\} \mid 1 \leq i \leq n\}$. To reduce the costs of the cuts, the variable x_1 should come early in the ordering. If the variables are ordered $x_1 < x_2 < x_3 < \ldots$ then the average cost of a cut is constant. If the variables are ordered $x_1 > x_2 > x_3 \ldots$ the average cost of a cut is linear in n. Even if the variables are ordered $x_1 > x_2 > x_3 \ldots$, the average cost of a cut can be made constant by introducing the clause form of the formulae $x_1 \equiv y_1$ and $y_i \equiv y_{i+1}$ for $1 \leq i < n$. This is similar to splitting clauses. Then instead of the clause $\{x_1, x_i, x_{i+1}\}$ the clause $\{y_i, x_i, x_{i+1}\}$ may be used, with the variable ordering $x_1 > y_1 > x_2 > y_2 > x_3 > y_3 \ldots$. This preserves the meaning of the formula but reduces the average cost of a cut to a constant. This is one advantage of $\text{Q}\textsc{sat}_{\text{CNF}}$, namely, the costs of cuts can be reduced by introducing new variables. This is not so easy with BDDs. Of course, it is best to keep closely related variables near each other in the ordering if possible. For example, the clauses $\{\{x_i, x_{i+1}, x_{i+2}\} \mid 1 \leq i \leq n\}$ have a constant average cost of cut if the variables are ordered $x_1 < x_2 < x_3 \ldots$ or $x_1 > x_2 > x_3 \ldots$ but the cost can be much larger if other orderings are used. The general rule is to keep related variables close together, but variables that are closely related to many others should occur early in the ordering or should be handled by splitting or by introducing equivalences as above.

It will be clear from the test examples below that the choice of an ordering can have a dramatic effect on the average and maximum cost of a cut. It is probably best in general to give a good variable ordering based on the topology and layout of circuit, then perhaps optimize it using a reordering routine.

10 Choosing cuts

Even after a variable ordering is chosen, it is still necessary to choose which cuts to process in $\text{Q}\textsc{sat}_{\text{CNF}}$. In many examples, some of the cuts will have larger costs than others, and $\text{Q}\textsc{sat}_{\text{CNF}}$ can be much faster if the cuts to be processed by **simp** are chosen well. It is not obvious how to do this, however. For this purpose, we introduce a mathematical model of the time taken to process a cut and use it to derive a systematic method of choosing cuts that works well in practice.

Once we have a method of estimating the time t taken to process a cut, then the cut is chosen which minimizes the ratio t/d of this time t to d, where d is the number of variables eliminated (the variables considered to be bound in the call to **simp**). This cut is chosen so that the average time required to eliminate each variable is as small as possible.

The estimate used for the time t taken to process a cut is $2^{d/(2b)}2^{c/2}$ where c is the cost of the cut and d is the number of (bound) variables eliminated. Thus a large value of b (called the *best cut ratio*) means that eliminating bound variables is easy compared to handling free variables, so it's best to eliminate many variables at a time and choose large values of d. The default value for b is 30, which typically works well. We also require that at least $b/2$ variables be eliminated each round. It's reasonable that eliminating bound variables (Y) is easy compared to eliminating the free variables (X) because in the calls to **unsat** it is only necessary to find one model of Y to know whether the problem is satisfiable but in the free variable region it is necessary to find all models of X, which often takes much longer. Also, the procedure **taut** called by **simp** is very fast for clause form formulae, because it is only necessary to test if every clause is a tautology.

The best cut ratio b can be adjusted, too, based on the performance of QSAT_{CNF}. If the average cost of a cut is small but QSAT_{CNF} is taking a long time per round, then it must be that d is too large, so in this case b should be reduced. If each round is very fast, then probably too few variables are being eliminated each round, and b should be increased. If the average cost of a cut is large (say 50 or over), the only hope is to do the whole problem at once; this can be done by making the best cut ratio very large, say 1000, which essentially calls a DPLL algorithm once on the whole problem.

The estimate of the time taken on a cut could be improved by considering not only the cost of the cut but also the number of clauses of various sizes that cross the cut. This might give a more accurate estimate, and thereby improve the choice of cuts and further reduce the execution time of QSAT_{CNF}.

11 Test results

We ran QSAT_{CNF} on several benchmarks and have some test results. The benchmarks were chosen as problems typical of well-known hardware verification problems for which BDD's perform well. Three such problems were chosen, together with the problems from the IFIP problem set [19] which have been proposed as illustrative of the efficiency of BDD's.

The general way to convert a problem to conjunctive normal form is as follows: Suppose one has a circuit C to compute a Boolean function f. Corresponding to this circuit there is a Boolean expression $B_C(X, Y, Z)$ where X are the input variables, Y are the internal variables, and Z are the output variables of the circuit. Then $B_C(X, Y, Z) \supset (Z = f(X))$. The formula $(B_C(X, Y, Z) \wedge B_D(X, W, V)) \supset Z = V$ is valid iff C and D are equivalent circuits. To show that circuits C and D are equivalent, one negates the formula $(B_C(X, Y, Z) \wedge B_D(X, W, V)) \supset Z = V$, converts it to conjunctive normal form, and shows that the resulting formula is unsatisfiable. A structure-preserving translation is used to avoid an exponential increase in size due to the conjunctive normal form translation. Also, n-ary conjunctions and disjunctions are expressed in terms of binary conjunctions and disjunctions in order to produce smaller clauses. The ordering of the variables in the resulting formula is defined systematically in terms of a natural ordering on the input variables for the problems "cmpadd," "maxmin," and "barrel"; in general, variables that depend on larger input variables end up larger in the ordering.

In the following tables, for each problem, the maximum and average cost of a cut is given, both with variables renumbered to reduce these costs and for the original problem. The number of Boolean variables and the run times for QSAT$_{\text{CNF}}$, the time taken by QSAT$_{\text{CNF}}$ after renumbering the variables, and the run times for GRASP, SATO, and BDD's (2 versions) are also given. BDD$_1$ indicates the SMV tool[12] with BDD's constructed using MTBDD's (multiterminal BDD's) as intermediates, which is the default. This can be time consuming, so BDD$_2$ gives a version of SMV with another faster method (SMVFlatten) for creating the Boolean function from which BDD's are constructed. In the tables, QSAT indicates QSAT$_{\text{CNF}}$. For the IFIP problems, only QSAT$_{\text{CNF}}$ was run, and the g (GROW) parameter was varied in some cases. This flag controls the maximum bound on lemma size. In these tables, (nr) indicates that no variable reordering was done, and NA indicates that a problem was not attempted. All problems were run on a Pentium II 450 running Linux, except that BDD's were run on SMV on a Pentium III (Coppermine) 730 Mhz, running Linux. Default parameters for QSAT$_{\text{CNF}}$ were used on all problems, including the default reordering routine, except that in some cases it was specified that no reordering should be done and in some cases the GROW parameter was varied. Default parameters were also used for GRASP, SATO, and BDD's. Sometimes the cut costs are not affected by renumbering (reordering) the variables, because if the renumbering is not giving small cuts, the renumbering heuristic gives up quickly to save time and the original variable ordering is used instead.

problem	max cut	avg cut	max cut	avg cut	vbls
	renumbered		original		
cmpadd8-src2	10	6	19	10	289
cmpadd16-src2	12	7	22	13	629
cmpadd32-src2	14	9	25	15	1345
cmpadd64-src2	18	11	28	18	2845
cmpadd64-src4	18	11	20	14	2841
maxmin16-mpc	56	31	114	64	666
maxmin20-mpc	67	35	142	79	838
maxmin24-mpc	76	40	170	95	1010
maxmin28-mpc	93	48	198	100	1182
maxmin29.cnf	344	200	344	200	4554
dpmaxmin10	6	4	6	4	62
dpmaxmin30	6	4	6	4	182
dpmaxmin50	6	4	6	4	302
dpmaxmin100	6	4	6	4	602
dpmaxmin200	6	4	6	4	1202
dpmaxmin300	6	4	6	4	1802
dpmaxmin400	6	4	6	4	2402
dpmaxmin500	6	4	6	4	3002
barrel8-sc	26	17	218	126	309
barrel16-sc	56	36	755	425	1006
barrel16-sc	56	36	755	425	1550
barrel32-sc	2336	1288	2336	1288	4639
barrel32-sc	2336	1288	2336	1288	4639
barrel64-sc					

Table 1
Characteristics for several long thin problems.

problem	QSAT		GRASP	SATO	BDD$_1$	BDD$_2$
	total	a. r.				
	times in seconds					
cmpadd8-src2	.23	0.19	0.19	0.05	0.01	0.01
cmpadd16-src2	1.57	1.38	2.05	1.09	0.01	0.02
cmpadd32-src2	2.35	1.58	58.76	NA	0.02	0.03
cmpadd64-src2	5.90	2.70	640.32	NA	0.09	0.06
cmpadd64-src4	4.23	2.41	NA	NA	NA	NA
maxmin16-mpc	1.01	0.81	0.12	14.30	0.01	0.01
maxmin20-mpc	1.45	1.14	0.21	88.78	0.01	0.01
maxmin24-mpc	3.45	3.04	0.26	NA	0.02	0.02
maxmin28-mpc	4.76	4.24	0.46	NA	0.02	0.02
maxmin29.cnf	7.04	4.04	1.65	NA	NA	NA
dpmaxmin10	0.01	0.01	0.01	0.01	0.01	0.01
dpmaxmin30	0.17	0.15	0.19	0.49	0.02	0.03
dpmaxmin50	0.25	0.19	0.67	5.18	0.06	0.07
dpmaxmin100	0.45	0.34	5.57	74.95	0.22	0.28
dpmaxmin200	0.87	0.54	56.66	NA	1.24	1.61
dpmaxmin300	1.40	0.67	210.59	NA	4.27	5.54
dpmaxmin400	2.14	0.90	505.97	NA	10.95	13.6
dpmaxmin500	3.04	1.17	NA	NA	22.81	28.8
barrel8-sc	0.11	0.02	0.02	0.01	0.01	0.01
barrel16-sc	1.12	0.21	0.18	0.11	0.03	0.07
barrel16-sc	0.18(nr)					
barrel32-sc	6.73	2.23	1.53	2.90	0.27	0.53
barrel32-sc	2.10(nr)					
barrel64-sc	NA	NA	NA	NA	81.67	3.91

Table 2
Results for several long thin problems.

problem	max cut	avg cut	max cut	avg cut	vbls	QSAT g=250	QSAT g=100	QSAT g=10
	renumbered		original			times in seconds		
add4.11.clause	12	7	53	33	163	0.29	NA	NA
add4.10.clause	15	10	49	30	150	0.30	NA	NA
add4.9.clause	16	10	45	28	137	0.27	NA	NA
add4.8.clause	12	8	41	25	124	0.21	NA	NA
add4.7.clause	13	8	37	23	109	0.10	NA	NA
add4.6.clause	13	8	33	20	96	0.13	NA	NA
The rest are all under 0.5 seconds, too.								
addsub.14.clause	10	6	44	24	113	0.13	NA	NA
addsub.13.clause etc. all under 0.5 seconds, too								
mul7.9.clause	17	10	24	14	64	0.07	NA	0.04(nr)
mul7.8.clause	19	11	32	18	94	0.85	NA	0.13(nr)
mul7.7.clause	22	13	42	24	130	16.05	4.05(nr)	0.55(nr)
mul7.6.clause	23	15	50	28	164	NA	NA	1.66(nr)
mul7.5.clause	22	16	56	31	194	NA	NA	3.45(nr)
mul7.4.clause	25	17	60	33	218	NA	181.4(nr)	5.13(nr)
mul7.3.clause	25	17	63	35	236	NA	NA	4.99(nr)
mul7.2.clause	23	17	65	36	248	NA	NA	2.90(nr)
mul7.1.clause	24	17	66	37	254	8.98	NA	1.67(nr)
mul7.0.clause	24	17	66	37	254	0.99	NA	0.55(nr)
mul8.0.clause	27	19	84	47	338	10.45	NA	2.02(nr)
mul8.1.clause	27	19	84	47	338	> 48 sec	NA	9.63(nr)
mul8.2.clause	26	19	83	46	332	NA	NA	17.93(nr)
rip6.all.clause	9	5	24	13	43	0.21	NA	NA
rip8.all.clause	10	6	31	17	55	0.18	NA	NA
add3.all.clause	21	14	42	25	135	1.67	NA	NA
mul4.all.clause	23	15	27	18	96	0.17	NA	NA
bf1355-315.cnf	269	204	269	204	2287	1.21	NA	NA
ssa0432-001.cnf	47	30	108	72	435	0.15	NA	NA

Table 3
Results for IFIP and other problems

The cmpadd8-src2, cmpadd16-src2, . . . problems involve showing that a ripple-carry adder with some number of bits (8, 16, . . .) is equivalent to a carry look-ahead adder. The cmpadd8-src4, cmpadd16-src4, . . . problems are similar but with a different variable ordering. Both series of problems use a structure-preserving translation so that the maximum number of variables per clause is three. The test results for these problems are given in Table 1 and Table 2. It is interesting that cmpadd64 with a good renumbering can have an average cost of cut of only 11; some of our other renumberings reduced this value to 8. This suggests that problems on which BDDs are fast are typically very thin with a proper variable ordering. Qsat$_{\text{CNF}}$ has a much slower rate of growth than SATO or GRASP on these problems, and takes only a few seconds, but is still slower than BDD's.

The "maxmin" problems have been described above. The "mpc" versions have large costs of cuts, but many of them are solved quickly anyway because Qsat$_{\text{CNF}}$ essentially calls SATO once on the whole problem or a large portion of it. SATO is slower here largely because its default value for the GROW parameter is small.

The "dpmaxmin" problems are the maxmin problems with the ordering given in section 9. These problems have a maximum clause size of four. The cuts have very small cost, indicating the importance of the problem formulation and the ordering. The test results for the maxmin and dpmaxmin problems can also be found in Table 1 and Table 2. Here again Qsat$_{\text{CNF}}$ far outperforms SATO and GRASP, and gives small solution times. In this case, Qsat$_{\text{CNF}}$ is even faster than BDD's for very large problems..

The "barrel-sc" problems formalize a barrel shifter and specify that one bit of the X register is shifted properly for all values of the S register. These examples are solved fast by Qsat$_{\text{CNF}}$, GRASP, and SATO, though the cut costs before renumbering are very large. This is because Qsat$_{\text{CNF}}$ simply calls SATO on the whole problem, and the basic Davis and Putnam procedure works well on these problems. Though BDD's perform well, they begin to blow up for the 64 bit case, at least when MTBDD's are used as intermediates.. Note how dramatically the reordering heuristic reduces the cut costs.

The add4, addsub, mul7, mul8, and rip problems are all taken from the IFIP benchmarks [19]. The clause form of these problems was provided by Stickel and Uribe. Without knowing a good ordering on the input variables, it was difficult to order the variables, but the problems are generally easy nonetheless. Many of these problems have small cut widths using our renumbering. Sometimes, as in the mul7 and mul8 problems, the performance depends highly on the lemma size bound g. However, this does not reveal anything particular to $\text{Q}\textsc{sat}_{\text{CNF}}$, but rather general properties of the Davis and Putnam algorithm, because SATO is called once on essentially the whole problem, due to the large costs of the cuts. These test results are found in Table 3. These tests show that for a satisfiability checker to perform well on problems suited to BDD's, sometimes it is necessary to choose the g parameter carefully. Perhaps choosing g small when cut costs are small is a good general heuristic.

We also tried a couple of randomly chosen problems from the DIMACS set [24]. These had large cut costs, but in both cases $\text{Q}\textsc{sat}_{\text{CNF}}$ essentially called SATO on the whole problem and in this way solved both problems quickly. Note however that the "ssa" problem with reordering of variables, has small cut costs. These problems are also given in Table 3.

The renumberings used for the IFIP and DIMACS examples were often done without considering a good ordering on the input variables of the circuit. This information could lead to better variable orderings and better performance for $\text{Q}\textsc{sat}_{\text{CNF}}$. Since BDDs make use of variable orderings, this information could easily be supplied to $\text{Q}\textsc{sat}_{\text{CNF}}$ as well.

In general, $\text{Q}\textsc{sat}_{\text{CNF}}$ appears to be sufficiently fast on all examples tried where BDDs are fast. The run times were at most a few seconds. Thus DPLL type methods (including $\text{Q}\textsc{sat}_{\text{CNF}}$) appear to be competitive with BDDs on a few problems typical of those encountered in hardware verification. $\text{Q}\textsc{sat}_{\text{CNF}}$ is often much faster than SATO and GRASP on long thin problems, and sometimes even faster than BDD's. It is also of interest to note that many of these problems have small cut costs, especially when the variables are reordered, indicating that methods specialized for small cut costs may have significant applicability. Futhermore, even our simple heuristic for minimizing cut costs was able to find orderings giving small cut costs in a few seconds at most; this suggests that finding a good variable ordering is not a significant difficulty. Of course, much better variable ordering routines may exist, such as that used in [18], further reducing the costs of the cuts and increasing the applicability of $\text{Q}\textsc{sat}_{\text{CNF}}$.

12 Other methods

We now discuss the augmented sum method presented by Truemper in Chapter 11 of his book [25]. This method is expressed in matrix terms, but solves a satisfiability problem essentially by eliminating variables and clauses from a set of clauses and adding a set of constraints to express the effect of the eliminated variables and clauses. This gives a reduced problem which can be solved directly or in the same way, by additional augmented sum solutions.

The differences between the augmented sum method and QSAT$_{\text{CNF}}$ are the following:

(1) The augmented sum method eliminates both clauses and variables, while QSAT$_{\text{CNF}}$ eliminates only variables. Hence we will consider the version of the augmented sum method that eliminates only variables.
(2) The augmented sum method eliminates all clauses containing eliminated variables and adds new variables corresponding to the subsets of clauses consisting of eliminated variables. One such new variable is added per clause. In contrast, QSAT$_{\text{CNF}}$ adds no new variables.
(3) The augmented sum method adds new clauses relating the new variables to the remaining variables. QSAT$_{\text{CNF}}$ adds new clauses but they only mention remaining (non-eliminated) variables.

Since the reduced problem in the augmented sum method might actually have more clauses than the original problem, the method is not applied to cuts that would lead to this result. However, QSAT$_{\text{CNF}}$ may be applied even when the number of clauses increases dramatically during the elimination of some variables. There are some other differences between the two methods relating to the manner in which backtracking is done and the manner in which new clauses are expressed. However, the basic philosophy of the two methods is similar.

There is another method which can be polynomial on thin systems, namely, the strategy0 option of SATO. We now discuss this option and prove that if lemmas are generated properly, it runs in polynomial time on log width sets of clauses. As in theorem 8, this shows that this option can be exponentially faster than OBDDs on some problems. However, in practice, this setting is sometimes exponential for SATO, and we discuss the reasons for this. The strategy0 option is the one in which variables in DPLL are chosen for splitting in their numerical order. Recall that the *(cut) width* of a set S of clauses is the maximum cost of an ordered cut of S.

For this analysis we give a simplified version of the DPLL[7] algorithm, but the same analysis applies in general to DPLL. Recall that DPLL is the method in which case analysis is used to eliminate variables that cannot be removed by other methods; in the original Davis and Putnam paper [6] resolution was used for this purpose. In our simplified version of the DPLL algorithm, we use a LIFO stack s to hold a sequence of literals representing the current interpretation. Whenever DPLL is called, a literal is pushed onto s, and whenever DPLL returns, a literal is popped off s. The sequence of calls to DPLL can be thought of as constructing a binary tree of partial interpretations and searching it depth-first. A stack s can also be thought of as a node in this binary tree. At the top level, DPLL is called with an empty stack.

Let $I(s)$ be the (partial) interpretation making all literals on the stack true. Thus a variable x is satisfied by $I(s)$ if x is on s, x is falsified by $I(s)$ if $-x$ is on s, and neither is true otherwise. We say that a set S of clauses is *falsified by* $I(s)$ if for some clause C in S, every literal L in C appears negated on s. This is also written "$S|_{I(s)}$ contains the empty clause." If this is true we call s a *conflict node* for S. We also say that $I(s)$ falsifies C in this case. When this happens, DPLL backtracks and tries another possibility. We call *push*$(x$,$s)$ and *push*$(\neg x, s)$ *children* of s if neither x nor $\neg x$ appear on s. We also call *push*(x, s) and *push*$(\neg x, s)$ *siblings* of each other.

We assume there is a lemma generation procedure *lemma*(s, S) which if $S|_{I(s)}$ is unsatisfiable returns a clause such that:

```
if $S$ is falsified by $I(s)$
  then $lemma(s, S)$ is a clause $C$ in $S$ that is falsified by $I(s)$, else
if $x$ is not in $lemma(push(x, s), S)$
  then $lemma(s, S) = lemma(push(x, s), S)$ else
if $\neg x$ is not in $lemma(push(\neg x, s), S)$
  then $lemma(s, S) = lemma(push(\neg x, s)$,S) else
$lemma(s, S) = (lemma(push(x, s), S)$ - $\{x\}) \cup (lemma(push(\neg x, s), S)$ - $\{\neg x\})$.
```

We note that

(1) *lemma*(s, S) is a logical consequence of S
(2) *lemma*(s, S) is falsified by $I(s)$
(3) for every literal L in *lemma*(s, S) there is a clause C in S containing L such that the variable in *top*(s) or some larger variable appears in C.
(4) if S is falsified by $I(s)$ then *lemma*(s, S) is a clause C in S

Note also that the only time a new lemma is derived is when S is not falsified by $I(s)$ and x is in *lemma*$($*push*(x, s),$S)$ and $\neg x$ is in *lemma*$($*push*$(\neg x, s))$.

We also assume that the variables are linearly ordered and that S is of width w with respect to this ordering, that is, if a clause C in S contains both the i^{th} and j^{th} variable in this ordering, then $|i - j| \leq w$. It follows from the third item above that *lemma*(s, S) has at most w literals in it. The procedure DPLL(s, S) is as follows:

```
procedure DPLL($s, S$) [[ test if $S|_{I(s)}$ is satisfiable ]]
  if $S|_{I(s)}$ contains the empty clause then return false else
  if(all variables in $S$ appear on $s$) then return true else {
    let $x$ be the smallest variable in $S$ that does not appear on $s$;
    if DPLL($push(x, s)$,$S$) then return true else
    if $x$ is not in $lemma(push(x, s)$,$S$) then return false else
    if DPLL($push(\neg x, s)$,$S$) then return true else
      {if $\neg x$ is in $lemma(push(\neg x, s)$ then $S \leftarrow S \cup$ {$lemma(s, S)$ };
       return false};
  }
end DPLL;
```

Note that a lemma can be added to S at most once. This is because after a lemma C is added to S, if $I(s)$ falsifies C, then DPLL(s, S) will determine that $S|_{I(s)}$ contains the empty clause and in this case no new lemmas are added to S. Furthermore, whenever a lemma C is added to S, it must be the case that $I(s)$ falsifies C.

Theorem 9 *If S is of width w then the run time for DPLL(λ,S) with lemma generation on S as described above is $\mathcal{O}(nc^w)$ where n is the number of variables in S and c is a constant and λ is the empty stack.*

Proof. Let us say that (s, S) is *top-free* if *lemma*(s, S) does not contain the complement of *top*(s). Now, if $S|_{I(s)}$ contains the empty clause, then (s, S) is not top-free because in this case every literal in *lemma*(s, S) appears complemented on s, and if (s, S) were top-free then DPLL would have backtracked before reaching s. Furthermore, if (*push*(x, s),S) and (*push*$(\neg x, s)$,S) are both not top-free, then *lemma*(s, S) is a new lemma that is added to S. Therefore if *push*(x, s) and *push*$(\neg x, s)$ are both conflict nodes for S, then *lemma*(s, S) is a new lemma.

It follows that if S has width w, then the number of s for which *push*(x, s) and *push*$(\neg x, s)$ are both conflict nodes for S, is bounded by 3^w. If we think of the DPLL search as a binary tree, then *push*(x, s) and *push*$(\neg x, s)$ are sibling nodes that are both leaves of the tree. One can show that the number of nodes in a binary tree of height h having at most k such sibling leaf nodes is $\mathcal{O}(hk)$. Therefore the number of calls to DPLL is $\mathcal{O}(n3^w)$ where n is the number of variables. q.e.d.

Corollary 12.1 *If the set of clauses has $\mathcal{O}(\log(n))$ width then DPLL with lemma generation as described runs in polynomial time.*

To obtain this result it is only necessary to generate new lemmas when both $S|_{I(push(x,s))}$ and $S|_{I(push(\neg x,s))}$ contain the empty clause. Some of our test results (not included) showed that SATO is not polynomial with the strategy0 option; this suggests that SATO's lemma mechanism differs from that described above. Therefore modifying SATO to achieve the polynomial time bound could already introduce a slowdown into SATO because of the extra work to generate this kind of lemma, and because of the additional lemmas that would be generated. This modified version of DPLL would have all lemmas from all levels stored together, further increasing the number of lemmas. $\text{Q}\textsc{sat}_{\text{CNF}}$ often does 30 or more variable elimination rounds. Thus DPLL with the strategy0 flag and lemma generation as indicated above would have perhaps 30 times as many lemmas as $\text{Q}\textsc{sat}_{\text{CNF}}$, probably a lot more because it would also be necessary to modify SATO's lemma mechanism. SATO has to go through all lemmas and clauses containing a given variable a number of times on every step, so these extra lemmas might slow it down. Also, all these lemmas could make this version of DPLL use a lot more storage. These extra lemmas might make the cache behavior much worse as well; $\text{Q}\textsc{sat}_{\text{CNF}}$ only works on a small number of variables at a time, and might have better cache performance. Still, the strategy0 option deserves looking into.

We now discuss some other related papers. A discussion of the efficiency of the original version of Davis and Putnam's method (which essentially performs ordered resolution) can be found in [26]. The authors show that this method is efficient for long, thin circuits and give some complexity bounds. In fact, the complexity bounds given for $\text{Q}\textsc{sat}_{\text{CNF}}$ also apply to ordered resolution (with 2^w replaced by a^w for some constant a). The authors also show how to construct a model of the set of clauses if it is satisfiable. The authors give a number of heuristics for choosing a variable ordering in order to make ordered resolution efficient. In addition, they consider two combinations of Davis and Putnam's method with resolution. The first one simply bounds the size of the resolvents that are kept, and is incomplete. The second one involves finding a cut set, a set of variables that disconnects the problem into two parts, and considering truth assignments to the cut variables. For each such assignment, resolution can be done on the two remaining parts, or they can be split again. This combination is complete. Neither approach is the same as $\text{Q}\textsc{sat}_{\text{CNF}}$, although $\text{Q}\textsc{sat}_{\text{CNF}}$ does have some similarities to ordered resolution.

The paper [27] presents a satisfiability tester for unquantified Boolean formulae similar to Stalmarck's method. This method is not similar to $\text{Q}\textsc{sat}_{\text{CNF}}$ either, but the paper is interesting and gives some cases where satisfiability testers far outperform BDDs. Qsat$_{\mathrm{CNF}}$ would probably do very well on such problems compared to BDDs, too.

Another approach to handling quantified Boolean formulae is the Q-resolution of Kleine-Buening [28]. This is a version of resolution that operates on clauses whose literals can be quantified Boolean formulae. It permits the removal of universally quantified Boolean variables during the resolution operation. This would not apply to Qsat on clause form formulae because all variables are either existentially quantified (the variables Y) or free variables (X) whose universal quantifier is outside the scope of the resolution operations. Also, Qsat has a global approach to choosing which resolutions are necessary.

Another variant of BDDs are the *zero-suppressed* OBDDs (ZBDDs) [29] in which a value of zero is assumed as the default and the characteristic function of the non-zero set is described. In [30], ZBDDs are used to represent very large sets of clauses. It is shown that the original Davis and Putnam method (ordered resolution) can sometimes be very efficient when ZBDDs are used to represent the set of clauses obtained after each group of ordered resolutions on a maximal variable. This method can work very well when the original set of clauses has a simple structure, permitting sets of ordered resolvents to be represented by small ZBDDs. On such sets, this approach can far outperform the DPLL method.

The paper [31] gives comparisons between DPLL and BDDs on the IFIP benchmarks. However, most of these problems are very easy for Qsat$_{\mathrm{CNF}}$.

13 Discussion

Some theoretical results show that Qsat is efficient on a class of quantified Boolean formulae related to symbolic model checking. Both theoretical and empirical results show the superiority of Qsat$_{\mathrm{CNF}}$ over DPLL and BDD's on some problems. The theoretical results show that Qsat$_{\mathrm{CNF}}$ has a smaller worst case bound than DPLL and BDD's on "long and thin" problems. An example was given on which BDD's are exponentially slower than Qsat$_{\mathrm{CNF}}$. The empirical results show that the Qsat$_{\mathrm{CNF}}$ implementation is often dramatically faster than GRASP and SATO on long, thin problems, and sometimes faster than BDD's. It is also well known that DPLL itself is often much faster than BDD's on problems with large cut widths; Qsat$_{\mathrm{CNF}}$ would have a similar behavior on these problems because it would just call DPLL once on the whole problem.

The fact that many of the problems examined have small cut costs suggests that such problems may be common. The fact that even our variable reordering routine was often able to find good orderings suggests that in many cases, finding a good variable ordering is not a problem. Even in the paper [18], many of the problems considered had a small cut width and finding a good variable ordering was not difficult. Furthermore, if one has a knowledge of the overall structure of a problem, one can often devise a significantly better variable ordering.

One advantage of QSAT is that one has some *a priori* measure of how well QSAT will perform on a quantified Boolean formula, given by the minimum k for which the formula is k width bounded. This means that one can attempt to preprocess the formula to reduce k and make QSAT more efficient, or not use QSAT on unsuitable formulae. In the same way, the cut width of a clause form formula gives an *a priori* measure of the efficiency of QSAT_{CNF}. One need not even use QSAT_{CNF} on formulae or variable orderings that are unsuitable, and the suitability of a formula or variable ordering can be precalculated without human guidance. By the same reasoning, one can generate many variable orderings by different techniques and pick the "best" one systematically. Perhaps it is not so simple to compute a suitability measure for BDD's in advance, to help in the choice of formulae or variable orderings to use.

A question that remains is how "thin" does a formula have to be for QSAT and QSAT_{CNF} to be efficient. The "dpmaxmin" examples are extreme, in that the average cut width is only 4, and there are thousands of variables. Even if QSAT_{CNF} is superior on such problems, there may not be many problems having such an extreme structure. On the other hand, the QSAT_{CNF} implementation makes heavy use of file i/o to communicate between the rounds. If this were eliminated, and the implementation optimized in other ways, it might be competitive even on problems whose cut width was not so small.

Undoubtedly there will continue to be improvements in the basic DPLL procedure, and some of these new procedures may be as fast or faster than QSAT_{CNF} on long and thin formulae. However, QSAT_{CNF} can be implemented on top of any DPLL procedure, and so it can be made more efficient at the same time. Furthermore, assuming these faster DPLL procedures still have an exponential worst-case bound, QSAT_{CNF} will still have a better worst case bound on long and thin formulae.

14 Conclusions

The QSAT algorithm for testing satisfiability of quantified Boolean formulae has been presented and analyzed on a class of quantified Boolean formulae related to symbolic model checking. A specialization QSAT_{CNF} of this algorithm to clause form formulae has been given and analyzed theoretically. This method eliminates variables from a set of clauses using a DPLL - like method [7]. A theoretical result showing that QSAT_{CNF} can be exponentially faster than BDDs is also given. The same result applies to ordered resolution [6], but QSAT_{CNF} is much faster than ordered resolution on some problems because it is based on DPLL which is often much faster than ordered resolution. The strategy0 option of SATO is analyzed theoretically and shown to be polynomial on log width circuits if lemma generation is done properly. In fact, the complexity bounds for QSAT_{CNF} also apply to the strategy0 option of SATO if lemma generation is done properly. This implies that the strategy0 option of SATO can be exponentially faster than BDD's on some examples, if lemma generation is done properly. However, modifying SATO in this way might incur a time or space penalty.

Though the results of this paper are mainly theoretical, test results of an implementation suggest that QSAT_{CNF} may be fast enough to be practical on clause form formulae obtained from problems for which BDDs are fast. QSAT_{CNF} is often dramatically faster than GRASP and SATO on the problems tested, and sometimes even faster than BDD's. There may be many problems on which QSAT_{CNF} is superior to both DPLL and BDD's, because most tests were done on problems that are ideal for BDDs. Some suggestions for improving the QSAT_{CNF} implementation are given; a better implementation may be significantly faster.

A number of related previous works are discussed, and none appear to be identical to QSAT_{CNF}. The closest is the augmented sum method presented by Truemper [25]. Applications of QSAT and QSAT_{CNF} to symbolic model checking are given, and it is possible that these procedures could also be of practical use on problems from this domain.

References

[1] G. Stålmarck, M. Säflund, Modeling and verifying systems and software in propositional logic, in: B. K. Daniels (Ed.), Safety of Computer Control Systems (SAFECOMP'90), Pergamon Press, 1990, pp. 31–36.

[2] R. E. Bryant, Graph-based algorithms for Boolean function manipulation, IEEE Transactions on Computers 35 (8) (1986) 677–691.

[3] C. Berman, L. Trevillyan, Functional comparison of logic designs for VLSI circuits, in: Proc. Intl. Conf. on Computer Aided Design, 1989, pp. 456–459.

[4] A. Kuehlmann, A. Srinivasan, D. LaPotin, Verity - a formal verification program for custom CMOS circuits, IBM Journal of Research and Development 39 (1995) 149–165.

[5] T. Kropf, Introduction to Formal Hardware Verification, Springer-Verlag, 1999.

[6] M. Davis, H. Putnam, A computing procedure for quantification theory, Journal of the Association for Computing Machinery 7 (1960) 201–215.

[7] M. Davis, G. Logemann, D. Loveland, A machine program for theorem-proving, Commun. ACM 5 (1962) 394–397.

[8] H. Zhang, SATO: An efficient propositional prover, in: International Conference on Automated Deduction (CADE'97), no. 1249 in LNAI, Springer-Verlag, 1997, pp. 272–275.

[9] A. Gupta, P. Ashar, Integrating a Boolean satisfiability checker and BDDs for combinational verification, in: Proc. of VLSI Design 98, 1998, pp. 222–225.

[10] P. R. Stephan, R. K. Brayton, A. L. Sangiovanni-Vincentelli, Combinational test generation using satisfiability, Tech. Rep. M92/112, Departement of Electrical Engineering and Computer Science, University of California at Berkley (October 1992).

[11] J. R. Burch, E. M. Clarke, K. L. McMillan, Symbolic model checking: 10^{20} states and beyond, Information and Computation 98 (1992) 142–170.

[12] K. L. McMillan, Symbolic Model Checking: An Approach to the State Explosion Problem, Kluwer Academic Publishers, 1993.

[13] E. M. Clarke, Jr., O. Grumberg, D. A. Peled, Model Checking, MIT Press, 1999.

[14] A. Biere, A. Cimatti, E. M. Clarke, Y. Zhu, Symbolic model checking without BDDs, in: TACAS'99, no. 1579 in LNCS, Springer-Verlag, 1999.

[15] J. Gu, P. Purdon, J. Franco, B. Wah, Algorithms for the satisfiability (SAT) problem : A survey, DIMACS series in Discrete Mathematics and Theoretical Computer Science 35 (1997) 19–151.

[16] M. Cadoli, A. Giovanardi, M. Schaerf, An algorithm to evaluate quantified Boolean formulae, in: Proc. of AAAI-98, 1998.

[17] J. Rintanen, Improvements to the evaluation of quantified Boolean formulae, in: T. Dean (Ed.), Proceedings of the 16th International Joint Conference on Artificial Intelligence, Morgan Kaufmann, 1999, pp. 1192–1197.

[18] M. R. Prasad, P. Chong, K. Keutzer, Why is ATPG easy?, in: Proceedings of the 36th Design Automation Conference (DAC99), 1999, pp. 22–28.

[19] L. J. Claesen, Efficient tautology checking algorithms, in: Formal VLSI Correctness Verification – VLSI Design Methods, Vol. II, Elsevier, 1990, p. Chapter 2.

[20] C. Berman, Circuit width, register allocation and ordered binary decision diagrams, IEEE Trans. CAD 10 (8) (1991) 1059–1066.

[21] D. Plaisted, S. Greenbaum, A structure-preserving clause form translation, Journal of Symbolic Computation 2 (1986) 293–304.

[22] W. J. Yakowenko, Propositional theorem proving by semantic tree trimming for hardware verification, Ph.D. thesis, University of North Carolina at Chapel Hill (July 1999).

[23] J. P. M. Silva, Search algorithms for satisfiability problems in combinational switching circuits, Ph.D. Dissertation, EECS Department, University of Michigan .

[24] DIMACS challenge benchmarks, ftp://ftp.rutgers.dimacs.edu/challenges/sat.

[25] K. Truemper, Effective Logic Computation, Wiley and Sons, 1998.

[26] I. Rish, R. Dechter, Resolution versus search: Two strategies for SAT, Journal of Automated Reasoning 24:1-2 (2000) 225–275.

[27] J. F. Groote, J. P. Warners, The propositional formula checker Heer Hugo, Journal of Automated Reasoning 24:1-2 (2000) 101–125.

[28] H. K. Buening, M. Karpinski, A. Fluegel, Resolution for quantified Boolean formulas, Information and Computation 117 (1995) 12–18.

[29] S. Minato, Zero-suppressed BDD's for set manipulation in combinatorial problems, in: 30th ACM/IEEE Design Automation Conference, 1993, pp. 272–277.

[30] P. Chatalic, L. Simon, ZRes: the old Davis-Putnam procedure meets ZBDD, in: International Conference on Automated Deduction (CADE'00), no. 1831 in LNAI, Springer-Verlag, 2000, pp. 449–454.

[31] T. E. Uribe, M. Stickel, Ordered binary decision diagrams and the Davis-Putnam procedure, in: J. P. Jouannaud (Ed.), Proceedings of the 1st International Conference on Constraints in Computational Logics, no. 845 in Lecture Notes in Computer Science, Springer-Verlag, 1994.