Efficient SAT Solving for Non-Clausal Formulas Using DPLL, Graphs, and Watched Cuts*

Himanshu Jain†
Verification Group
Synopsys, Inc.

Edmund M. Clarke
School of Computer Science
Carnegie Mellon University

ABSTRACT

Boolean satisfiability (SAT) solvers are used heavily in hardware and software verification tools for checking satisfiability of Boolean formulas. Most state-of-the-art SAT solvers are based on the Davis-Putnam-Logemann-Loveland (DPLL) algorithm and require the input formula to be in conjunctive normal form (CNF). We present a new SAT solver that operates on the negation normal form (NNF) of the given Boolean formulas/circuits. The NNF of a formula is usually more succinct than the CNF of the formula in terms of the number of variables. Our algorithm applies the DPLL algorithm to the graph-based representations of NNF formulas. We adapt the idea of the two-watched-literal scheme from CNF SAT solvers in order to efficiently carry out Boolean Constraint Propagation (BCP), a key task in the DPLL algorithm. We evaluate the new solver on a large collection of Boolean circuit benchmarks obtained from formal verification problems. The new solver outperforms the top solvers of the SAT 2007 competition and SAT-Race 2008 in terms of run time on a large majority of the benchmarks.

Categories and Subject Descriptors: J.6 [Computer Aided Engineering]: [Computer-Aided Design]

General Terms: Algorithms, Design, Verification

Keywords: Boolean Satisfiability, Verification, DPLL, NNF

1. INTRODUCTION

The problem of propositional (Boolean) satisfiability (SAT) is to decide whether a given propositional formula is satisfiable. This problem is of central importance in hardware and software verification, logic synthesis, automatic test generation, and artificial intelligence. Most state-of-the-art SAT procedures [3, 4, 5, 17, 18, 13, 11, 19, 8] are variations of the Davis-Putnam-Logemann-Loveland (DPLL) algorithm and require the input formula to be in conjunctive normal form (CNF). Typical formulas arising in practice are not necessarily in CNF. We refer to propositional formulas not in CNF form as *non-clausal* formulas (Boolean circuits). As argued by Thiffault et al. [20] converting a non-clausal formula to CNF introduces overhead in form of a large number of new variables and may destroy the initial structure of the formula, which can be crucial in efficient satisfiability checking.

Suppose we are given a non-clausal formula ϕ. In order to check the satisfiability of ϕ using a CNF based SAT solver, ϕ needs to be converted to CNF. Let us assume for now that ϕ is in *negation normal form (NNF)*. This means that ϕ contains only "$\wedge$" ("AND"), "$\vee$" ("OR"), "$\neg$" ("NOT") operators and the scope of each "$\neg$" is a propositional variable. There are two ways of converting ϕ to a CNF formula. The first way of converting ϕ to a CNF formula is by introduction of new variables [21]. This produces a CNF formula ϕ' that is equi-satisfiable to ϕ and is linear in the size of ϕ. This is most common and practical way of converting ϕ to a CNF formula.

The second method is to expand ϕ using the distributive properties of $\wedge, \vee$ in order to obtain a CNF formula. Let us denote the CNF formula obtained by expansion of ϕ as $CNF(\phi)$. For example let ϕ be $(a \wedge \neg b) \vee (c \wedge (d \vee \neg f))$. Then $CNF(\phi)$ is $(a \vee c) \wedge (a \vee d \vee \neg f) \wedge (\neg b \vee c) \wedge (\neg b \vee d \vee \neg f)$. Note that $CNF(\phi)$ and ϕ contain the same set of variables and are logically equivalent. This method of obtaining $CNF(\phi)$ from ϕ is impractical because the size of $CNF(\phi)$ can be exponential in the size of ϕ.

In this paper we present a new SAT solver that checks the satisfiability of ϕ by applying the DPLL algorithm to the *hpgraph* [7, 15] of ϕ. Each path in the hpgraph of ϕ, starting from a root node and ending at a leaf node, corresponds to a clause in $CNF(\phi)$. That is, the hpgraph of ϕ implicitly encodes $CNF(\phi)$. Fig. 1(a) shows the hpgraph of the formula $(a \wedge \neg b) \vee (c \wedge (d \vee \neg f))$. The size of the hpgraph of ϕ is linear in the size of ϕ. Thus, the hpgraph compactly represents $CNF(\phi)$. By using the hpgraph we avoid explicitly listing out the clauses in $CNF(\phi)$, which can require exponential time and space in the size of ϕ.

Let us denote the disjunctive normal form (DNF) formula obtained by converting ϕ to DNF by expanding out ϕ as $DNF(\phi)$. Let ϕ be $(a \wedge \neg b) \vee (c \wedge (d \vee \neg f))$. Then $DNF(\phi)$ is $(a \wedge \neg b) \vee (c \wedge d) \vee (c \wedge \neg f)$. Once again explicitly listing out $DNF(\phi)$ can be exponential in the size of ϕ. Our solver utilizes the *vpgraph* [7, 15] of ϕ. The vpgraph of ϕ implicitly encodes $DNF(\phi)$ and is linear in the size of ϕ. The vpgraph for our example is shown in Fig. 1(b).

1.1 Our Contributions

- We present a new SAT solver that checks the satisfiability of a NNF formula ϕ by applying the DPLL algorithm to the hpgraph of ϕ. Our solver also utilizes the vpgraph of ϕ in certain steps of SAT solving. If the input formula/circuit is not in NNF it is

*This research was sponsored by the Semiconductor Research Corporation (SRC), the Gigascale Systems Research Center (GSRC), the Office of Naval Research (ONR), the Naval Research Laboratory (NRL), the Army Research Office (ARO) and General Motors Lab at CMU.

†The author did this research as a graduate student at CMU.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
DAC 2009, July 26 - 31, 2009, San Francisco, California, USA
Copyright 2009 ACM 978-1-60558-497-3 -6/08/0006 ...$5.00.

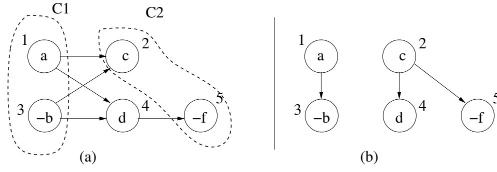

Figure 1: Let ϕ be $(a \wedge \neg b) \vee (c \wedge (d \vee \neg f))$. (a) The hpgraph of ϕ. Two node cuts C_1, C_2 are shown. (b) The vpgraph of ϕ. We show the negation of a variable by using a minus sign.

first converted to an equi-satisfiable NNF formula as explained in Section 2.

- The most crucial component of our DPLL SAT solver is an efficient Boolean Constraint Propagation (BCP) algorithm on the hpgraph. We describe an algorithm for performing BCP on hpgraph that *adapts* the *two-watched-literal scheme* [18] found in CNF SAT solvers. In particular, a "watch" in an hpgraph corresponds to a *node cut* in the hpgraph. By maintaining two node cuts for each connected component in the hpgraph we achieve the same effect as the two watched-literal scheme found in the CNF SAT solvers. Fig. 1(a) shows two node cuts C_1, C_2 (possible watches) for a hpgraph. Two node cuts allow watching two nodes (literals) on each path (clause) in a hpgraph component. The two-watched-literal scheme used in CNF SAT solvers is a special case of our algorithm (when the hpgraph represents a CNF formula). As in CNF SAT solvers *non-chronological backtracking* is cheap as the node cuts are not updated when backtracking. (Section 3)
- We show how to update the node cuts (watches) in the hpgraph efficiently by using the vpgraph of the given formula. We show that a *minimal* cut in a hpgraph corresponds to a path in the corresponding *vpgraph*. Thus, finding a small node cut in an hpgraph corresponds to finding a path in the corresponding vpgraph. For example, notice that paths $\langle 1,3 \rangle, \langle 2,5 \rangle$ in the vpgraph shown in Fig 1(b) correspond to cuts C_1, C_2, respectively, in the hpgraph shown in Fig 1(a). (Section 4)
- We have carefully implemented these ideas in a non-clausal SAT solver. We evaluate the solver on 2541 non-clausal benchmarks obtained from publicly available sources. Our solver outperforms the top SAT solvers of the SAT 2007 competition and SAT-Race 2008 in terms of runtime. (Section 5)

Related Work: Most state-of-the-art SAT procedures are based on DPLL search and require the input formula to be in CNF [3, 4, 5, 17, 18, 13, 11, 19, 8]. There has been work on applying DPLL directly to circuit [12, 16, 20] representations. In [12] a hybrid SAT solver is described where the original formula is processed in circuit form, and learned clauses are processed separately in CNF. The circuit-based BCP is implemented by means of a lookup table. In [20] a watched literal scheme is proposed for efficient BCP on a given circuit. Unlike existing circuit SAT solvers our SAT solver does not operate on the circuit representation directly. In our approach a given formula/circuit is converted to an equi-satisfiable NNF formula. The NNF formula is then represented in the form of two graphs called the vpgraph and hpgraph. These graphs are used in our SAT algorithms. Jain et al. [15] use a vpgraph/hpgraph in a *General Matings* based SAT solver. This work uses the vpgraph/hpgraph in a DPLL-based SAT solver. In our unreported experiments we found the solver in [15] to have poor performance on the benchmarks we consider in this paper.

A technical report version of this paper with proofs can be found in the chapters 2 and 4 of [14].

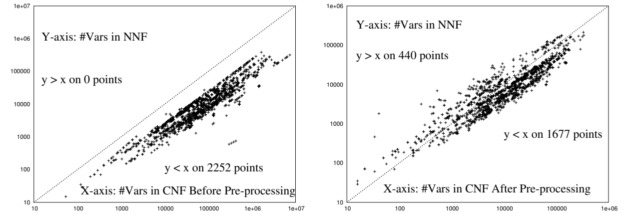

Figure 2: Number of variables comparison.

2. PRELIMINARIES

A Boolean formula is in *negation normal form (NNF)* iff it contains only the Boolean connectives "$\wedge$" ("AND"), "$\vee$" ("OR") and "$\neg$" ("NOT"), and the scope of each occurrence of "$\neg$" is a Boolean variable. We also require that there is no *structure sharing* in a NNF formula, that is, output from a gate acts as input to at most one gate. A NNF formula is tree-like, while a circuit can be DAG-like.

Conversion of Boolean Circuits to NNF: Most formulas obtained in practice are Boolean circuits. In our work Boolean circuits are converted to NNF formulas in two stages. The first stage re-writes other operators (such as xor, iff, implies, if-then-else) in terms of $\wedge$, $\vee$, $\neg$ operators. In order to avoid a blowup in the size of the resulting formula we allow sharing of sub-formulas. Thus, the first stage produces a formula containing $\wedge, \vee, \neg$ gates, possibly with structure sharing. The second stage gets rid of the structure sharing in order to obtain a NNF formula. This is done by introduction of new variables. We introduce a new variable for each gate that has a fanout *greater* than one. Once the sharing is removed the negations can be pushed to the variables using DeMorgans laws. Observe that the conversion of a Boolean circuit to a NNF formula can be done in linear time in the size of the Boolean circuit. We compare the number of variables in the NNF and CNF representations of a collection of industrial Boolean circuits in Fig. 2. The NNF forms have $5-10$ times fewer variables than CNF. Modern CNF SAT solvers use pre-processing techniques in order to eliminate variables and clauses from the input CNF formula [9]. Fewer variables reduce overhead during BCP and can make the decision heuristics more effective. Even after pre-processing of CNF formulas the NNF forms have fewer variables than CNF on 80% of our benchmarks. The fewer variables in the NNF form (without any pre-processing) motivates the need for exploring SAT solving techniques that operate on NNF directly. We provide experimental evidence that shows the utility of the NNF forms in SAT solvers.

In the subsequent sections we assume that the input Boolean circuit has been converted to a NNF formula. Given a NNF formula ϕ our SAT algorithms check the satisfiability of ϕ without introducing any more new variables. We use the work by Andrews [7] and Jain et al. [15] in order to represent the NNF formula in form of two graphs the *hpgraph* and *vpgraph*.

Hpgraph: Given a NNF formula ϕ, the hpgraph $G_h(\phi)$ is defined as a tuple (V, R, L, E, Lit), where V is the set of nodes corresponding to all occurrences of literals in ϕ, $R \subseteq V$ is a set of root nodes, $L \subseteq V$ is a set of leaf nodes, $E \subseteq V \times V$ is the set of edges, and $Lit(n)$ denotes the literal associated with node $n \in V$. A root node $n \in R$ has no incoming edges and a leaf node $n \in L$ has no outgoing edges.

The hpgraph for a NNF formula $\phi := (a \wedge \neg b) \vee (c \wedge (d \vee \neg f))$ is shown in Fig. 1(a). We have $V = \{1,2,3,4,5\}$, $R = \{1,3\}$, $L = \{2,5\}$, $E = \{(1,2),(1,4),(3,2),(3,4),(4,5)\}$ and for each $n \in V$, $Lit(n)$ is shown inside the node labeled n.

Vpgraph: Given a NNF formula ϕ, the vpgraph $G_v(\phi)$ is also defined as a tuple (V', R', L', E', Lit'). The vpgraph for the formula ϕ is shown in Fig. 1(b). We have $V' = \{1,2,3,4,5\}$, $R' = \{1,2\}$,

$L' = \{3,4,5\}$, $E' = \{(1,3),(2,4),(2,5)\}$ and for each $n \in V$, $Lit(n)$ is shown inside the node labeled n.

Jain et al. [14, 15] present a recursive (linear time) procedure for obtaining the hpgraph and vpgraph from a given NNF formula. We briefly review the recursive steps involved in the creation of an hpgraph: 1) the hpgraph for a literal l is a new node n with $Lit(n) = l$, 2) the hpgraph for $\phi_1 \wedge \phi_2$ is the union of the hpgraphs of ϕ_1 and ϕ_2, and 3) the hpgraph for $\phi_1 \vee \phi_2$ is obtained by connecting leaves in the hpgraph of ϕ_1 to the roots in the hpgraph of ϕ_2.

For this paper we only need to understand the key properties of the hpgraph and vpgraph. A path $\pi = \langle n_0, \ldots, n_k \rangle$ in $G_h(\phi)$ or $G_v(\phi)$ is said to be a **rl-path** iff it starts at a root node and ends at a leaf node. In Fig. 1(a), $\langle 1,2 \rangle$, $\langle 1,4,5 \rangle$, $\langle 3,2 \rangle$, $\langle 3,4,5 \rangle$ are rl-paths.

Key Property of the Hpgraph and Vpgraph: Each rl-path π in the hpgraph $G_h(\phi)$ corresponds to a clause obtained by collecting the literals occurring on π. For example, the rl-path $\langle 1,4,5 \rangle$ in Fig. 1(a) corresponds to the clause $a \vee d \vee \neg f$. The collection of all clauses occurring in the hpgraph gives a CNF form for the given formula. Each rl-path in the vpgraph $G_v(\phi)$ corresponds to a cube (term). For example, the rl-path $\langle 2,4 \rangle$ in Fig. 1(b) corresponds to the term $c \wedge d$. The collection of all terms occurring in the vpgraph gives a DNF form of the given formula.

THEOREM 1. *Let $G_h(\phi)$ be the hpgraph and $G_v(\phi)$ be the vpgraph of ϕ. Let π denote a rl-path and n denote a node on π.*
(a) ϕ is equivalent to the CNF formula $\bigwedge_{\pi \in G_h(\phi)} \bigvee_{n \in \pi} Lit(n)$.
(b) ϕ is equivalent to the DNF formula $\bigvee_{\pi \in G_v(\phi)} \bigwedge_{n \in \pi} Lit(n)$.

The clauses (terms) corresponding to rl-paths in hpgraph (vpgraph) can be redundant (that is, contain a literal and its negation).
Using Hpgraph inside a SAT Solver: Given an assignment σ to a subset of variables in ϕ, we say that there is a **conflict** iff σ falsifies ϕ. A literal l is an **implied (unit)** literal iff l must be set to true in order to obtain a satisfying assignment under σ. We say an assignment *falsifies* a node n in $G_h(\phi)$ or $G_v(\phi)$ iff the assignment falsifies $Lit(n)$. We use the hpgraph of ϕ in order to detect conflicts and implied literals. Recall that each rl-path in the hpgraph corresponds to a clause. The following corollary states that if each node of a rl-path is false then there is a conflict.

COROLLARY 1. *Given an assignment σ to variables in ϕ the following are equivalent: 1) σ falsifies ϕ. 2) there exists a rl-path π in $G_h(\phi)$ such that σ falsifies every node on π.*

Consider the hpgraph in Fig. 1(a). The assignment $\sigma := \{a = 0, c = 0\}$ falsifies every node on rl-path $\langle 1,2 \rangle$. Thus, σ falsifies ϕ.

COROLLARY 2. *Let σ be an assignment to variables in ϕ. If there is a rl-path π in $G_h(\phi)$ and a node $m \in \pi$ such that σ falsifies every node in π except m, and $Lit(m)$ is not assigned in σ, then $Lit(m)$ is an implied literal.*

The implied literal detected using the above corollary will be termed as an *h-implied* literal. We will refer to the node m in the above corollary as an *h-implied* node. Consider the hpgraph shown in Fig. 1(a) and an assignment $\sigma := \{a = 0, d = 0\}$. σ falsifies all but node 5 on the rl-path $\langle 1,4,5 \rangle$ in the hpgraph. It follows that $Lit(5) = \neg f$ is an implied literal (due to clause $a \vee d \vee \neg f$). That is, f must be set to zero under the current assignment.

A main difference between the existing DPLL SAT solvers and our solver is in the Boolean constraint propagation (BCP). In our solver the BCP algorithm uses the hpgraph of a given formula.

3. BCP ON THE HPGRAPH

Let V denote the set of variables in a given formula ϕ. Given an assignment σ of truth values to a set of variables $W \subseteq V$, the Boolean constraint propagation (BCP) algorithm detects two cases. (1) It reports if σ falsifies ϕ (conflict). (b) If there is no conflict, the BCP algorithm provides a set of implied (unit) literals. Before we describe the BCP algorithm on a hpgraph, we briefly review the BCP algorithm used in modern CNF SAT solvers.

3.1 Review of BCP in CNF SAT solvers

Most modern CNF SAT solvers use the *two-watched-literal scheme* [18] in order to obtain an efficient BCP algorithm. Suppose we are given a CNF formula ϕ. Let D be a clause in ϕ. In the two-watched-literal scheme two *watches* are associated with D. A *watch* is simply a literal l occurring in D. Before the search (DPLL algorithm) starts any two literals in D can be designated as its watches. Let l_1, l_2 be the watches corresponding to D. Four cases arise depending upon the status of l_1, l_2 given the current assignment σ.
Case A: Watches l_1, l_2 are either true or unassigned. In this case there cannot be any conflict or an implied literal due to D. The clause D is not even examined during BCP.

The clause D is examined only when one of its watches becomes false. Without loss of generality assume that l_2 becomes false in the remaining three cases.
Case B: If l_1 is already true, then D is already satisfied. In this case nothing needs to be done even though l_2 is false.

If l_1 is not true, the solver tries to replace the falsified watch (l_2) by another watch that is not false. If there is a literal l_3 in D that is not false and $l_3 \neq l_1$, then l_2 is replaced by l_3. However, such a literal (l_3) may not exist in the remaining two cases.
Case C (conflict): All the literals in D are false. In this case D is false under the current assignment.
Case D (implied literal): l_1 is unassigned but all other literals in D are false. In this case, l_1 is reported as an implied literal.

The main benefit of the two-watched-literal scheme is that it reduces the number of times the solver examines the clauses in a given CNF formula. This is crucial for obtaining efficient solvers that can handle CNF formulas with millions of clauses. Another advantage is that the *non-chronological backtracking* is cheap. This is because the watched literals do not need to be updated during backtracking. We now describe how the two-watched-literal scheme found in CNF SAT solvers can be adapted to obtain an efficient algorithm for BCP on a hpgraph.

3.2 Generalizing Two-Watched-Literal Scheme to Two-Watched-Cut Scheme for Hpgraph

Let ϕ be a NNF formula. We are given an assignment σ to a subset of variables occurring in ϕ. The BCP algorithm uses the hpgraph $G_h(\phi)$ of ϕ to detect conflicts and implied literals. Given $G = (V, E, R, L, Lit)$ we say that $C \subseteq V$ is a **rl-cut** in G iff removal of all nodes in C from G disconnects all rl-paths in G. For example, $\{1,3,4\}, \{2,3,4\}, \{5,6,8\}, \{5,7,8\}$ are some of the rl-cuts in the hpgraph shown in Fig. 3. The node set $\{2,7,8\}$ is not an rl-cut as it does not disconnect the rl-paths $\langle 3,5 \rangle, \langle 4,5 \rangle$. An rl-cut contains at least one node from each rl-path. More precisely let C be a rl-cut in $G_h(\phi)$. For every rl-path π in $G_h(\phi)$ there exists a node n such that $n \in \pi$ and $n \in C$. Two rl-cuts C_1, C_2 are said to be **node-disjoint** if $C_1 \cap C_2 = \emptyset$. For example the rl-cuts $\{1,3,4\}, \{5,7,8\}$ in Fig. 3 are node-disjoint.
Watches in a Hpgraph: Each rl-path in a hpgraph corresponds to a clause. Let the clause corresponding to an rl-path π be D. In order to apply the two-watched-literal scheme found in CNF SAT solvers we want to *watch* two nodes n_1, n_2 on π. This in turn amounts

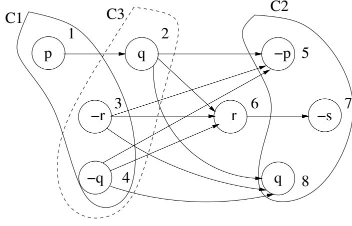

Figure 3: An hpgraph. Three rl-cuts $C_1 := \{1,3,4\}, C_2 := \{5,7,8\}, C_3 := \{2,3,4\}$ are shown.

to watching two literals $Lit(n_1), Lit(n_2)$ in D. However, there are usually exponentially many paths (clauses) in a hpgraph. So it is expensive to maintain watches for each rl-path (clause) explicitly.

This intuition leads us to define a *watch* in a hpgraph as a *rl-cut* in the hpgraph. By taking a rl-cut as a watch we make sure that at least one node on every rl-path is present in our watch. This in turn corresponds to watching a literal on each clause in the hpgraph. For example, the rl-cut $C := \{1,3,4\}$ is a possible watch for the hpgraph in Fig. 3. Note that C contains at least one node from each rl-path in Fig. 3. Watching node 3 on rl-paths $\langle 3,5\rangle, \langle 3,6,7\rangle, \langle 3,8\rangle$, amounts to watching literal $Lit(3) = \neg r$ on clauses $\neg r \vee \neg p, \neg r \vee r \vee \neg s, \neg r \vee q$, respectively.

In order to get the effect of the two-watched-literal scheme we watch two rl-cuts in the hpgraph. By maintaining two rl-cuts for a hpgraph we are able to watch two nodes (literals) on each rl-path (clause) in the hpgraph. For example the rl-cuts $C_1 := \{1,3,4\}$ and $C_2 := \{5,7,8\}$ are two possible watched cuts for the hpgraph in Fig. 3. For the rl-path $\langle 1,2,6,7\rangle$, the rl-cuts C_1, C_2 allow us to watch nodes 1, 7 (literals $p, \neg s$).

Suppose we are given $G = (V,E,R,L,Lit)$, a partial assignment σ, and a rl-cut $W \subseteq V$. We say that W is **acceptable** iff there is no node $m \in W$ such that $Lit(m)$ is false in σ. We say that W is **satisfied** iff for all $m \in W$ $Lit(m)$ is true in σ. For example, given the hpgraph in Fig. 3 and $\sigma := \{q = 1\}$ the rl-cuts $\{5,6,8\}, \{5,7,8\}$ are acceptable. The rl-cuts $\{2,3,4\}, \{1,3,4\}$ are not acceptable given σ (due to node 4).

3.3 BCP on Hpgraph Using Two Watched Cuts

For a given hpgraph $G_h(\phi) = (V,E,R,L,Lit)$ we maintain two rl-cuts C_1, C_2 (watches). Before the DPLL algorithm starts C_1, C_2 can be initialized to any two rl-cuts in $G_h(\phi)$ that are node-disjoint. As the search progresses the algorithm tries to maintain the invariant that at least one of C_1, C_2 is acceptable. The algorithm also tries to maintain C_1, C_2 as node-disjoint as possible. This is useful for detecting implied literals. Recall, that implied literals detected using the hpgraph (Corollary 2) are called h-implied literals. We describe the various cases that may arise during the BCP on a hpgraph below. Each of the cases below generalize the cases that occur in the two-watched-literal scheme for CNF SAT solvers.

Case A: Both rl-cuts (watches) C_1, C_2 are node-disjoint and acceptable. Then there can be no conflict or h-implied literals due to the current assignment. This is because each clause in the hpgraph contains two literals that are not false. In this case there is no need to look at any other part of the hpgraph. In Fig. 3 let $C_1 := \{1,3,4\}, C_2 := \{5,7,8\}$ and $\sigma := \{r = 0\}$. Observe that both C_1, C_2 are acceptable rl-cuts and are node-disjoint.

Suppose one of the rl-cuts say C_2 is no longer acceptable. Then we have the following cases.

Case B: For each node $n \in C_1$, $Lit(n)$ is already true, that is, C_1 is *satisfied*. In this case there cannot be any conflict or an h-implied literal in the hpgraph. Intuitively, every clause present in the hpgraph is satisfied. The algorithm leaves C_2 unchanged in this case.

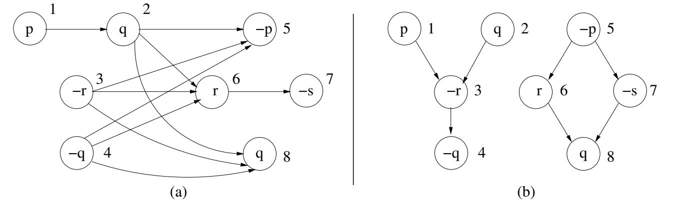

Figure 4: (a) Hpgraph for formula $(((p \vee q) \wedge \neg r \wedge \neg q) \vee (\neg p \wedge (r \vee \neg s) \wedge q))$. (b) The corresponding vpgraph.

In Fig. 3 let $C_1 := \{1,3,4\}, C_2 := \{5,7,8\}$ and $\sigma := \{p = 1, r = 0, q = 0\}$. Observe that C_2 is not acceptable as $Lit(5), Lit(8)$ are false under σ. However, C_1 is satisfied. So C_2 is not updated.

If the previous cases do not apply, then the algorithm tries to find a replacement rl-cut for C_2. When searching for a replacement to C_2, the algorithm tries to find a rl-cut that is as different from C_1 as possible. Intuitively, this is similar to why we keep two *distinct* watched literals in a clause in the CNF two-watched-literal scheme. If a replacement cut C_3 is found such that C_3 is acceptable and $C_3 \cap C_1 = \emptyset$, then C_2 is replaced by C_3. Otherwise, we have the following two cases.

Case C (conflict): There is no acceptable rl-cut in the hpgraph. In this case the current assignment σ falsifies the given formula. In Fig. 3 let $C_1 := \{1,3,4\}, C_2 := \{5,7,8\}$ and $\sigma := \{p = 1, r = 1\}$. In this case neither C_1 nor C_2 is acceptable and there is no possible replacement for them. This is expected as the clause $(\neg r \vee \neg p)$ corresponding to the rl-path $\langle 3,5\rangle$ is false.

Case D (implications): There is an acceptable rl-cut C_3 but C_3 is not node-disjoint from C_1. In this case, for every $n \in C_1 \cap C_3$ the corresponding literal $Lit(n)$ is an h-implied literal (assuming $Lit(n)$ is not already true). If $C_3 \neq C_1$, then C_2 is replaced by rl-cut C_3.

In Fig. 3 let $C_1 := \{1,3,4\}, C_2 := \{5,7,8\}$ and $\sigma := \{p = 1\}$. Observe that C_2 is not acceptable as $Lit(5)$ is false under σ. Also note that case B does not hold as C_1 is not satisfied. Thus, we seek a replacement for C_2. Note that any new acceptable rl-cut must include nodes 3,4 since as they are the only possible nodes that can be watched on the paths $\langle 3,5\rangle, \langle 4,5\rangle$, respectively. Thus, a possible rl-cut C_3 is $\{2,3,4\}$. Both C_1, C_3 contain nodes 3,4. It can be seen that $Lit(3), Lit(4)$ are precisely the h-implied literals. $Lit(3) = \neg r$ is h-implied due to the rl-path $\langle 3,5\rangle$, which corresponds to the clause $\neg r \vee \neg p$. Similarly, $Lit(4) = \neg q$ is h-implied due to the rl-path $\langle 4,5\rangle$, which corresponds to the clause $\neg q \vee \neg p$. Since h-implied literals are also implied literals it follows that $\neg r, \neg q$ are implied literals given the current assignment.

4. MINIMAL RL-CUTS IN HPGRAPH

In section 3.3 we adapted the CNF two-watched-literal scheme to hpgraphs by using two rl-cuts in the hpgraph $G_h(\phi)$. We now describe how rl-cuts are obtained and updated efficiently during BCP. The key idea is to make use of the vpgraph $G_v(\phi)$. For example, consider the hpgraph in Figure 4 (a) and the corresponding vpgraph in Figure 4 (b). Observe that any rl-path in the vpgraph corresponds to a rl-cut in the hpgraph. The rl-paths $\langle 1,3,4\rangle$, $\langle 2,3,4\rangle$, $\langle 5,6,8\rangle$, $\langle 5,7,8\rangle$ in the vpgraph corresponds to rl-cuts $\{1,3,4\}, \{2,3,4\}, \{5,6,8\}, \{5,7,8\}$, respectively, in the hpgraph.

Given $G = (V,E,R,L,Lit)$ we say that $C \subseteq V$ is a **minimal rl-cut** in G iff C is a rl-cut in G and no proper subset of C is a rl-cut in G. Let $nodes(\pi)$ denote the set of nodes occurring on a rl-path π. A surprising fact is that the rl-paths in the vpgraph correspond to minimal rl-cuts in the hpgraph. One can also prove that every minimal rl-cut in the hpgraph corresponds to a rl-path in the vpgraph.

THEOREM 2. *Let $G_h(\phi)$ be a hpgraph and $G_v(\phi)$ be a vpgraph for a NNF formula ϕ. (a) Let π be a rl-path in $G_v(\phi)$. Then nodes(π) form a minimal rl-cut in $G_h(\phi)$. (b) Let C be a minimal rl-cut in $G_h(\phi)$. Then there exists a rl-path π in $G_v(\phi)$ such that $C = nodes(\pi)$.*

4.1 Updating Minimal RL-cuts in Hpgraph

Our algorithm always maintains two minimal rl-cuts in the hpgraph as the watched cuts. These cuts are obtained and updated by finding rl-paths in the corresponding vpgraph by using a depth-first search like routine. The BCP algorithm relies on the ability to find acceptable rl-cuts in the hpgraph. This is done by searching for acceptable rl-paths (rl-paths with no falsified nodes) in the vpgraph. The BCP routine also requires that we find disjoint rl-cuts in the hpgraph (if possible). This is done by searching for disjoint rl-paths in the vpgraph. More precisely, suppose we are trying to replace rl-cut C_2 in the hpgraph. Let the other rl-cut in the hpgraph be C_1 and let π_1 denote the rl-path corresponding to C_1 in the vpgraph. Then we search for a rl-path in the vpgraph that is completely disjoint from π_1. If we succeed in finding a path π_3 in the vpgraph that is completely disjoint from π_1, then we obtain a replacement C_3 for C_2 in the hpgraph such that $C_1 \cap C_3 = \emptyset$. If there is no rl-path in the vpgraph that is completely disjoint from π_1, we find the set of all nodes N on π_1 that must be shared by any acceptable rl-path in the vpgraph (this can be done in linear time). Intuitively, the nodes in N give us the precise set of h-implied literals.

THEOREM 3. *Let $G_h(\phi)$ be a hpgraph and $G_v(\phi)$ be a vpgraph for a NNF formula ϕ. Let π_1 be an acceptable rl-path in $G_v(\phi)$. Suppose there is no other acceptable rl-path in $G_v(\phi)$ that is completely node disjoint from π_1. Let N denote the set of nodes that must be shared in any acceptable rl-path in $G_v(\phi)$. (a) Then $N \neq \emptyset$. For every $m \in N$ either $Lit(m)$ is true or $Lit(m)$ is implied under the current assignment. (b) If m is an h-implied node, then $m \in N$.*

Optimizations: BCP based on *only* two-watched rl-cuts can be inefficient when the hpgraph has millions of nodes. This is because even the minimal rl-cuts for the entire hpgraph can be large and will be updated frequently during BCP. In practice, there are usually many hpgraph components (due to top level conjunctions and structure sharing). Each component is small as compared to the entire hpgraph in terms of number of nodes. For efficiency we maintain two watched rl-cuts for each hpgraph component. The algorithms above apply to each individual hpgraph/vpgraph component. See [14] for other important optimizations.

The hpgraph for a CNF formula is a disjoint union of various line graphs (hpgraph components) where each line graph represents a clause. A minimal rl-cut in a line graph is simply a cut of size one. Thus, the two-watched rl-cuts for each hpgraph component reduces to two-watched-literal scheme when the input is a CNF formula.

The other important components of our SAT solver such as decision heuristics, conflict driven learning, non-chronological backtracking, and restarts are implemented in a similar manner as other state-of-the-art SAT solvers. The conflict driven learning [17, 18] generates new clauses, which are added to a CNF clause database. The BCP routine takes into account both the hpgraph and the clause database in order to detect conflicts and implied literals.

5. EXPERIMENTAL RESULTS

The experiments are performed on a 1.86 GHz Intel Xeon (R) machine with 4 GB of memory running Linux. The techniques described in the paper have been implemented in a SAT solver called NFLSAT (Non-clausal FormuLas SATisfiability checker). The input formula is given in AIG (And Inverter Graph) [1], or ISCAS format. We evaluate the solver on a collection of 2541 Boolean circuits obtained from publicly available sources. These benchmarks consist of 1) 839 bounded model checking problems and 857 k-induction problems obtained from all sequential circuits used in the 2007 hardware model checking competition [2], 2) all 341 benchmarks that were used in the AIG track in the SAT competition 2007 [6]. The remaining benchmarks are obtained from microprocessor verification and equivalence checking domains. Out of 2541 Boolean circuits, 2192 circuits are in the AIG format. We use a timeout of 10 minutes per problem per solver.

We compare NFLSAT against three state-of-the-art CNF solvers RSAT [5], PicoSAT [4], and MiniSAT [3]. In SAT 2007 competition the solvers RSAT, PicoSAT, and MiniSAT were ranked first, second, third, respectively in the industrial category. We use the SAT 2007 competition version of RSAT and PicoSAT. We use the current public version of MiniSAT (minisat2-070721). The CNF versions of the above circuits were obtained by means of the standard Tseitin transformation [21]. We use `aigtocnf` [1] to convert the benchmarks in AIG format to CNF. Note that RSAT and MiniSAT use pre-processing [9] to simplify the input CNF formulas. We include the time required to obtain hpgraph/vpgraph from a Boolean circuit in NFLSAT's runtime.

We also compare NFLSAT with MiniSAT++ 1.0 which was ranked first in the AIG track of SAT-Race 2008. MiniSAT++ simplifies the given AIG circuit using DAG-aware rewriting and then converts the simplified circuit to CNF by using an improved Tseitin translation [10]. The resulting CNF is then passed to MiniSAT 2.1, which was ranked first in the CNF track in SAT-Race 2008. We also compare NFLSAT with PicoaigerSAT which was ranked second in the AIG track of SAT-Race 2008. MiniSAT++ and PicoaigerSAT directly accept AIG inputs. We compare NFLSAT with MiniSAT++ and PicoaigerSAT on 2192 AIG benchmarks.

Figure 5 gives scatter plots comparing NFLSAT with other solvers. NFLSAT has better performance on points below the line $y = x$. NFLSAT has better runtimes than RSAT, PicoSAT, MiniSAT, MiniSAT++, and PicoaigerSAT, on respectively 89%, 91%, 86%, 83%, and 86% of the benchmarks.

The experimental results are shown in Table 1. The first four rows summarize the results for all 2541 benchmarks, while the last three rows summarize the results for 2192 AIG benchmarks. For each solver we report: 1) Number of problems solved within timeout in the "Solved" column. 2) Number of problems where a timeout occurred in the "Timeout" column. 3) The total time spent in seconds on the problems that were solved in the "STime" column. 4) Sum of "Solved Time" and timeouts in the "TTime" column.

NFLSAT solves more problems than RSAT, PicoSAT, MiniSAT, PicoaigerSAT, and it is also faster in terms of run time. The main competition to NFLSAT is given by MiniSAT++ which solves 14 more problems within timeout. These 14 benchmarks are from SAT competition 2007 AIG benchmark suite and were obtained from CNF formulas by reverse engineering. The extraction of circuit structure from CNF is not perfect and many of the extracted circuits are simply a conjunction of clauses. On such CNF-like benchmarks NFLSAT is not able to match MiniSAT++ performance.

The main conclusion is that NFLSAT is competitive to the existing state-of-the-art SAT solvers on a majority of the Boolean circuit benchmarks in terms of runtime. The NNF form of Boolean circuits has fewer variables than (pre-processed) CNF in the majority of the cases. Fewer variables in turn reduce the overhead during the BCP and can make the decision heuristics more effective. The two-watched-cut scheme carries more overhead than the two-watched-literal scheme. However, the two-watched-cut scheme can potentially update the watches for a large number of clauses with-

Solver	Solved	Timeout	STime	TTime
NFLSAT	**2364**	**177**	**29753**	**135953**
RSAT	2310	231	45794	184394
PicoSAT	2281	260	43297	199297
MiniSAT	2270	271	39489	202089
NFLSAT	**2060**	**132**	**26585**	**105785**
MiniSAT++	2074	118	32457	103257
PicoaigerSAT	2033	159	35892	131292

Table 1: Summary of each SAT solvers performance.

out having to look at each clause individually.

6. SUMMARY

We presented a DPLL-based SAT solver that operates on the graph-based representations of non-clausal formulas (Boolean circuits). The input formula is converted to a NNF formula that is represented using two graphs. The hpgraph encodes the CNF form of the given NNF formula, while the vpgraph encodes the DNF form of the given NNF formula. The key step in the DPLL algorithm is Boolean constraint propagation (BCP). We adapt the idea of the two-watched-literal scheme from CNF SAT solvers in order to efficiently carry out BCP on hpgraph. In our algorithm two cuts are watched for each hpgraph component. The watched cuts are used to detect conflicts and implied literals. We use the duality between the cuts in a hpgraph component and the paths in the corresponding vpgraph component for efficiently updating the cuts. Experimental results show that the new SAT solver is faster than the state-of-the-art solvers on a majority of the benchmarks.

Acknowledgment. We thank Per Bjesse, Sicun Gao and Will Klieber for their valuable comments. The first author is grateful to his thesis committee members for their feedback on this work.

7. REFERENCES

[1] AIGER, `http://fmv.jku.at/aiger`.

[2] Hardware model checking competition, `http://fmv.jku.at/hwmcc07/`.

[3] Minisat sat solver. `http://minisat.se/`.

[4] Picosat sat solver. `http://fmv.jku.at/picosat/`.

[5] Rsat sat solver. `http://reasoning.cs.ucla.edu/rsat/`.

[6] SAT competition 2007, `www.satcompetition.org/2007/`.

[7] P. B. Andrews. *An Introduction to Mathematical Logic and Type Theory: to Truth through Proof*. Kluwer Academic Publishers, second edition, 2002.

[8] A. Biere. Picosat essentials. *Journal on Boolean Satisfiability, Boolean Modeling and Computation (JSAT)*, 4:75–97, 2008.

[9] N. Eén and A. Biere. Effective Preprocessing in SAT Through Variable and Clause Elimination. In *SAT*, pages 61–75, 2005.

[10] N. Eén, A. Mishchenko, and N. Sörensson. Applying Logic Synthesis for Speeding Up SAT. In *SAT*, pages 272–286, 2007.

[11] N. Eén and N. Sörensson. An Extensible SAT-solver. In *SAT*, pages 502–518, 2003.

[12] M. K. Ganai, P. Ashar, A. Gupta, L. Zhang, and S. Malik. Combining Strengths of Circuit-based and CNF-based Algorithms for a High-performance SAT solver. In *DAC*, 2002.

[13] E. Goldberg and Y. Novikov. BerkMin: A Fast and Robust Sat-Solver. In *DATE*, 2002.

[14] H. Jain. Verification using satisfiability checking, predicate abstraction, and craig interpolation. Technical Report CMU-CS-08-146, SCS, CMU, 2008.

[15] H. Jain, C. Bartzis, and E. M. Clarke. Satisfiability checking of non-clausal formulas using general matings. In *SAT*, pages 75–89, 2006.

[16] Feng Lu, Li-C. Wang, Kwang-Ting Cheng, and Ric C.-Y. Huang. A Circuit SAT Solver With Signal Correlation Guided Learning. In *DATE*, 2003.

[17] J. P. Marques-Silva and K. A. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *ICCAD*, pages 220–227, November 1996.

[18] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *DAC*, pages 530–535, June 2001.

[19] K. Pipatsrisawat and A. Darwiche. A lightweight component caching scheme for satisfiability solvers. In *SAT*, pages 294–299, 2007.

[20] C. Thiffault, F. Bacchus, and T. Walsh. Solving Non-clausal Formulas with DPLL Search. In *SAT*, 2004.

[21] G.S. Tseitin. On the complexity of derivation in propositional calculus. In *Studies in Constructive Maths and Mathematical Logic*, pages 115–125, 1968.

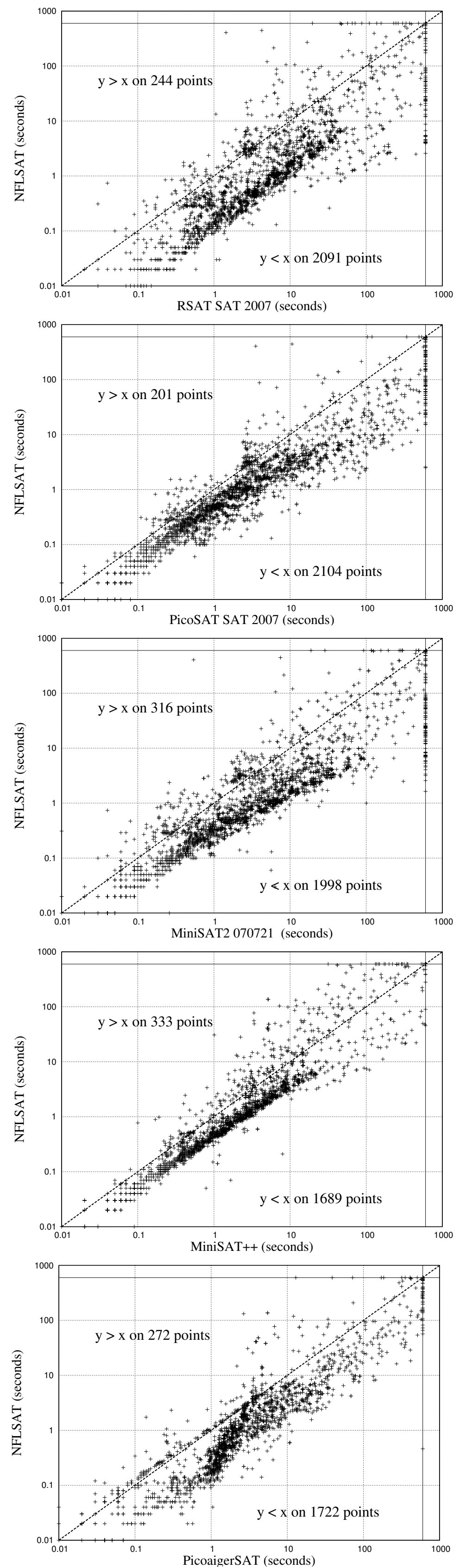

Figure 5: Scatter plots comparing run times of NFLSAT and other solvers.