Satisfiability Checking of Non-clausal formulas using General Matings ⋆

Himanshu Jain, Constantinos Bartzis, and Edmund Clarke

School of Computer Science, Carnegie Mellon University, Pittsburgh, PA

Abstract. Most state-of-the-art SAT solvers are based on DPLL search and require the input formula to be in clausal form (cnf). However, typical formulas that arise in practice are non-clausal. We present a new non-clausal SAT-solver based on General Matings instead of DPLL search. Our technique is able to handle non-clausal formulas involving $\wedge, \vee, \neg$ operators without destroying their structure or introducing new variables. We present techniques for performing search space pruning, learning, non-chronological backtracking in the context of a General Matings based SAT solver. Experimental results show that our SAT solver is competitive to current state-of-the-art SAT solvers on a class of non-clausal benchmarks.

1 Introduction

The problem of propositional satisfiability (SAT) is of central importance in various areas of computer science, including theoretical computer science, artificial intelligence, hardware design and verification. Most state-of-the-art SAT procedures are variations of the Davis-Putnam-Logemann-Loveland (DPLL) algorithm and require the input formula to be in conjunctive normal form (cnf). Typical formulas generated by the previously mentioned applications are not necessarily in cnf. As argued by Thiffault et al. [17] converting a general formula to cnf introduces overhead and may destroy the initial structure of the formula, which can be crucial in efficient satisfiability checking.

We propose a new propositional SAT-solving framework based on the General Matings technique due to Andrews [6]. It is closely related to the Connection method discovered independently by Bibel [8]. Theorem provers based on these techniques have been used successfully in higher order theorem proving [5]. To the best of our knowledge, General Matings has not been used in building SAT-solvers for satisfiability problems arising in practice. This paper presents techniques for building an efficient SAT-solver based on General Matings.

When applied to propositional formulas the General Matings approach can be summarized as follows [7]. The input formula is translated into a 2-dimensional format called *vertical-horizontal path form (vhpform)*. In this form disjuncts (operands of $\vee$) are arranged horizontally and conjuncts (operands of $\wedge$) are arranged vertically. The

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

formula is *satisfiable* if and only if there exists a *vertical path* through this arrangement that does not contain two opposite literals (l and $\neg l$). The input formula is not required to be in cnf.

We have designed a SAT procedure for non-clausal formulas based on the General Matings approach. At a high level our search algorithm enumerates all possible vertical paths in the vhpform of a given formula until a vertical path is found which does not contain two opposite literals. If every vertical path contains two opposite literals, then the given formula is unsatisfiable. The number of vertical paths can be exponential in the size of a given formula. Thus, the key challenge in obtaining an efficient SAT solver is to prevent the enumeration of vertical paths as much as possible. We present several novel techniques for preventing the enumeration of vertical paths. Our contributions can be summarized as follows:

- The vhpform of a given formula succinctly encodes: 1) disjunctive normal form (dnf) of a given formula as a set of vertical paths 2) conjunctive normal form (cnf) of a given formula as a set of *horizontal paths*. Our solver employs a combination of both vertical and horizontal path exploration for efficient SAT solving. The choice of which variable to assign next (*decision making*) is made using the vertical paths which are similar to the terms (conjunction of literals) in the dnf of a given formula. Conflict detection is aided by the use of horizontal paths which are similar to the clauses (disjunction of literals) in the cnf of a given formula.
- We show how to adapt the techniques found in the current state-of-the-art SAT solvers in our algorithm. We describe how to perform *search space pruning*, *conflict driven learning*, *non-chronological backtracking* by using the vertical paths and horizontal paths in the vhpform of a given formula.
- We present graph based representations of the set of vertical paths and the set of horizontal paths which makes it possible to implement our algorithms efficiently.

Related Work: Many SAT solvers have been developed, most employing some combination of two main strategies: the DPLL search and heuristic local search. Heuristic local search techniques [12] are not guaranteed to be complete, that is, they are not guaranteed to find a satisfying assignment if one exists or prove unsatisfiability. As a result, complete SAT solvers (such as GRASP [11], SATO [18], zChaff [14], BerkMin [10], Siege [4], MiniSat [2]) are based almost exclusively on the DPLL search. While most DPLL based SAT solvers operate on cnf, there has been some work on applying DPLL directly to circuit [9] and non-clausal [17] representations. The key differences between existing work and our approach are as follows:
- Unlike heuristic local search based techniques, we propose a complete SAT solver.
- Unlike DPLL based SAT solvers (operating on either cnf, circuit or non-clausal representation), the basis of our search procedure is General Matings. There is a crucial difference between the two techniques. In DPLL the search space is the set of all possible assignments to the propositional variables, whereas in General Matings the search space is the set of all possible vertical paths in the vertical-horizontal path form of a given formula. We give an example illustrating the difference in Section 2. In contrast to current cnf SAT solvers which produce a complete satisfying assignment (all variables are assigned), our solver produces partial satisfying assignments when possible.

- The General Matings technique is designed to work on non-clausal forms. In particular, any arbitrary propositional formula involving $\wedge, \vee, \neg$ is handled naturally, without introduction of new variables or loss of structural information.

Semantic Tableaux [16] is a popular theorem proving technique. The basic idea is to expand a given formula in the form of a tree, where nodes are labeled with formulas. If all the branches in the tree lead to contradiction, then the given formula is unsatisfiable. The tableau of a given propositional formula can blowup in size due to repetition of subformulas along the various paths. In contrast, when using General Matings a vertical-horizontal path form of a given formula is built first. This representation is a directed acyclic graph (DAG) and polynomial in the size of the given formula.

2 Preliminaries

A propositional formula is in *negation normal form (nnf)* iff it contains only the propositional connectives $\wedge$, $\vee$ and $\neg$ and the scope of each occurrence of $\neg$ is a propositional variable. It is known that every propositional formula is equivalent to a formula in nnf. Furthermore, a negation normal form of a formula can be much shorter than any dnf or cnf of that formula. The internal representation in our satisfiability solver is nnf. More specifically, we use a two-dimensional format of a nnf formula, called a *vertical-horizontal path form (vhpform)* as described in [7][1]. In this form disjunctions are written horizontally and conjunctions are written vertically. For example Fig. 1(a) shows the formula $\phi = (((p \vee q) \wedge \neg r \wedge \neg q) \vee (\neg p \wedge (r \vee \neg s) \wedge q))$ in vhpform.

Vertical path: A vertical path through a vhpform is a sequence of literals in the vhpform that results by choosing either the left or the right scope for each occurrence of $\vee$. For the vhpform in Fig. 1(a) the set of vertical paths is $\{\langle p, \neg r, \neg q\rangle, \langle q, \neg r, \neg q\rangle, \langle \neg p, r, q\rangle, \langle \neg p, \neg s, q\rangle\}$.

Horizontal path: A horizontal path through a vhpform is a sequence of literals in the vhpform that results by choosing either the left or the right scope for each occurrence of $\wedge$. For the vhpform in Fig. 1(a) the set of horizontal paths is $\{\langle p, q, \neg p\rangle, \langle p, q, r, \neg s\rangle, \langle p, q, q\rangle, \langle \neg r, \neg p\rangle, \langle \neg r, r, \neg s\rangle, \langle \neg r, q\rangle, \langle \neg q, \neg p\rangle, \langle \neg q, r, \neg s\rangle, \langle \neg q, q\rangle\}$.

The following are two important results regarding satisfiability of negation normal formulas from [7]. Let F be a formula in negation normal form and let σ be an assignment (σ can be a partial truth assignment).

Theorem 1. *σ satisfies F iff there is a vertical path P in the vhpform of F such that σ satisfies every literal in P.*

Theorem 2. *σ falsifies F iff there is a horizontal path P in the vhpform of F such that σ falsifies every literal in P.*

The vhpform in Fig. 1(a) has a vertical path $\langle p, \neg r, \neg q\rangle$ whose every literal can be satisfied by an assignment σ which sets p to true and r, q to false. It follows from Theorem 1 that σ satisfies ϕ. Thus, ϕ is satisfiable. An example of a vertical path whose every literal cannot be satisfied by any assignment is $\langle q, \neg r, \neg q\rangle$ (due to opposite literals q and

[1] In [7] the term *vertical path form (vpform)* is used in place of vertical-horizontal path form (vhpform). We use vertical-horizontal path form (vhpform) in this paper for clarity.

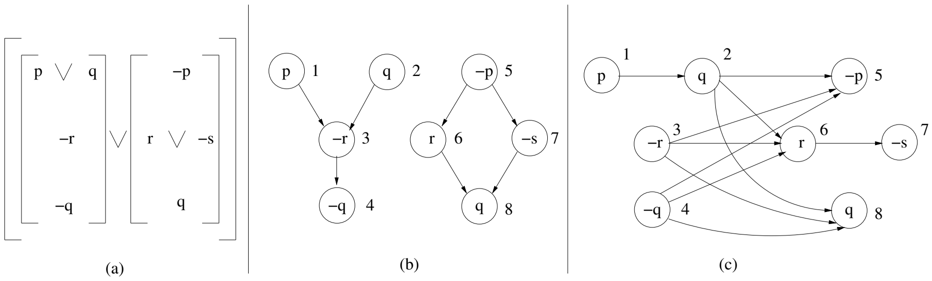

Fig. 1. We show the negation of a variable by a $-$ sign. (a) vhpform for the formula $(((p \vee q) \wedge \neg r \wedge \neg q) \vee (\neg p \wedge (r \vee \neg s) \wedge q))$ (b) the corresponding vpgraph (c) the corresponding hpgraph.

$\neg q$). An assignment σ' which sets p, r to true, falsifies every literal in the horizontal path $\langle \neg r, \neg p \rangle$ in the vhpform of ϕ. Thus, from Theorem 2 it follows that σ' falsifies ϕ.

Let $\mathcal{VP}(\phi)$ and $\mathcal{HP}(\phi)$ denote the set of vertical paths and the set of horizontal paths in the vhpform of a given formula, respectively. We use $l \in \pi$ to denote the occurrence of a literal l in a vertical/horizontal path π. The following result from [7] states that the set of vertical paths encodes the dnf and the set of horizontal paths encodes the cnf of a given formula.

Theorem 3. *(a) ϕ is equivalent to the dnf formula $\bigvee_{\pi \in \mathcal{VP}(\phi)} \bigwedge_{l \in \pi} l$. (b) ϕ is equivalent to the cnf formula $\bigwedge_{\pi \in \mathcal{HP}(\phi)} \bigvee_{l \in \pi} l$.*

Theorem 1 forms the basis of a General Matings based SAT procedure. The idea is to check the satisfiability of a given nnf formula by examining the vertical paths in its vhpform. For the vhpform in Fig. 1(a) the search space is $\{\langle p, \neg r, \neg q \rangle, \langle q, \neg r, \neg q \rangle, \langle \neg p, r, q \rangle, \langle \neg p, \neg s, q \rangle\}$. In contrast, the search space for a DPLL-based SAT solver is the set of all possible truth assignments to the variables p, q, r, s. We use Theorem 2 for efficient Boolean constraint propagation in two ways: 1) For detecting when the current candidate for a satisfying assignment falsifies the given formula (*conflict detection*). 2) For obtaining a *unit literal rule* (Section 3) similar to the one used in cnf SAT solvers.

3 Graph representations

Our SAT procedure operates on the graph based representations of the vhpform of a given formula. These graph based representations are described below.

Graphical encoding of vertical paths (vpgraph): A graph containing all vertical paths present in the vhpform of a nnf formula is called a *vpgraph*. Given a nnf formula ϕ, we define the vpgraph $G_v(\phi)$ 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 node $n \in R$ has no incoming edges and a node $n \in L$ has no outgoing edges.

The vpgraph containing all vertical paths in the vhpform of Fig. 1(a) is shown in Fig. 1(b). For the vpgraph in Fig. 1(b), we have $V = \{1, 2, 3, 4, 5, 6, 7, 8\}$, $R =$

$\{1,2,5\}$, $L = \{4,8\}$, $E = \{(1,3),(2,3),(3,4),(5,6),(5,7),(6,8),(7,8)\}$ and for each $n \in V$, $Lit(n)$ is shown inside the node labeled n in Fig. 1(b). Each path in the vpgraph $G_v(\phi)$, starting from a root node and ending at a leaf node, corresponds to a vertical path in the vhpform of ϕ. For example, path $\langle 1,3,4\rangle$ in Fig. 1(b) corresponds to the vertical path $\langle p, \neg r, \neg q\rangle$ in Fig. 1(a) (obtained by replacing node n on path by $Lit(n)$). Using this correspondence one can see that vpgraph contains all vertical paths present in the vhpform shown in Fig. 1(a).

Given ϕ, we can construct the vpgraph $G_v(\phi) = (V, R, L, E, Lit)$ directly without constructing the vhpform of ϕ. This is done inductively as follows:

- If ϕ is a literal l, then we create a graph containing just one node fv, where fv is a fresh identifier. The literal stored inside fv is set to l.

  $G_v(\phi) = (\{fv\}, \{fv\}, \{fv\}, \emptyset, Lit)$ and $Lit(fv) = l$, fv is a fresh identifier.
- If $\phi = \phi_1 \vee \phi_2$, then the vpgraph for ϕ is obtained by taking the union of the vpgraphs of ϕ_1 and ϕ_2. Let $G_v(\phi_1) = (V_1, R_1, L_1, E_1, Lit_1)$ and $G_v(\phi_2) = (V_2, R_2, L_2, E_2, Lit_2)$. Then $G_v(\phi)$ is the union of $G_v(\phi_1)$ and $G_v(\phi_2)$.

  $G_v(\phi) = (V_1 \cup V_2, R_1 \cup R_2, L_1 \cup L_2, E_1 \cup E_2, Lit_1 \cup Lit_2)$
- If $\phi = \phi_1 \wedge \phi_2$, then the vpgraph for ϕ is obtained by concatenating the vpgraph of ϕ_1 with the vpgraph of ϕ_2. Let $G_v(\phi_1) = (V_1, R_1, L_1, E_1, Lit_1)$ and $G_v(\phi_2) = (V_2, R_2, L_2, E_2, Lit_2)$. Then $G_v(\phi)$ contains all the nodes and edges in $G_v(\phi_1)$ and $G_v(\phi_2)$. But $G_v(\phi)$ has additional edges connecting leaves of $G_v(\phi_1)$ with the roots of $G_v(\phi_2)$. The set of additional edges is denoted as $L_1 \times R_2$ below. The set of roots of $G_v(\phi)$ is R_1, while the set of leaves is L_2.

  $G_v(\phi) = (V_1 \cup V_2, R_1, L_2, E_1 \cup E_2 \cup (L_1 \times R_2), Lit_1 \cup Lit_2)$

Graphical encoding of horizontal paths (hpgraph): A graph containing all horizontal paths present in the vhpform of a nnf formula is called a *hpgraph*. We use $G_h(\phi)$ to denote the hpgraph of a formula ϕ. The procedure for constructing a hpgraph is similar to the above procedure for constructing the vpgraph. The difference is that the hpgraph for $\phi = \phi_1 \wedge \phi_2$ is obtained by taking the union of hpgraphs for ϕ_1 and ϕ_2 and the hpgraph for $\phi = \phi_1 \vee \phi_2$ is obtained by concatenating the hpgraphs of ϕ_1 and ϕ_2.

The hpgraph containing all horizontal paths in the vhpform in Fig. 1(a) is shown in Fig. 1(c). For the hpgraph in Fig. 1(c), we have $V = \{1,2,3,4,5,6,7,8\}$, $R = \{1,3,4\}$, $L = \{5,7,8\}$, $E = \{(1,2),(2,5),(2,6),(2,8),(3,5),(3,6),(3,8),(4,5),(4,6),(4,8),(6,7)\}$ and for each $n \in V$, $Lit(n)$ is shown inside the node labeled n.

Using vpgraph/hpgraph: We use $G(\phi)$ to refer to either a vpgraph or hpgraph of ϕ. It can be shown by induction that the vpgraph and hpgraph of a nnf formula are directed acyclic graphs (DAGs). This fact allows obtaining more efficient versions of standard graph algorithms (such as shortest path computation) for vpgraph/hpgraph. The construction of vpgraph/hpgraph takes $O(k^2)$ time in the worst case where k is the size of the given formula. This is mainly due to the $L_1 \times R_2$ term in the handling of $\phi_1 \wedge \phi_2$ (for vpgraph construction) and $\phi_1 \vee \phi_2$ (for hpgraph construction).

Given a vpgraph or hpgraph $G(\phi) = (V, R, L, E, Lit)$, the following definitions will be used in subsequent discussion.

r-path: A path $\pi = \langle n_0, \ldots, n_k\rangle$ in $G(\phi)$ is said to be a r-path (rooted path) iff it starts with a root node ($n_0 \in R$). In Fig. 1(b), $\langle 2,3\rangle$ is a r-path while $\langle 3,4\rangle$ is not a r-path.

rl-path: A path $\pi = \langle n_0, \ldots, n_k \rangle$ in $G(\phi)$ is said to be a rl-path iff it starts at a root node and ends at a leaf node ($n_0 \in R$ and $n_k \in L$). In Fig. 1(b), both $\langle 2, 3, 4 \rangle$, $\langle 5, 6, 8 \rangle$ are rl-paths, but $\langle 3, 4 \rangle$ is not a rl-path.
Conflicting nodes: Two nodes $n_1, n_2 \in V$ are said to be conflicting iff $Lit(n_1) = \neg Lit(n_2)$. In Fig. 1(b), nodes 2,4 are conflicting.
- We say an assignment σ *satisfies (falsifies)* a node $n \in V$ iff σ satisfies (falsifies) $Lit(n)$. An assignment which sets q to true satisfies nodes 2, 8 and falsifies node 4 in Fig. 1(b).
- We say an assignment σ *satisfies (falsifies)* a path $\pi \in G(\phi)$ iff σ satisfies (falsifies) every node on π. For example, in Fig. 1(b) path $\langle 5, 6, 8 \rangle$ is satisfied by an assignment which sets p to false and r, q to true. The same path is falsified by an assignment which sets p to true and r, q to false. We say that a path $\pi \in G$ is *satisfiable* iff there exists an assignment which satisfies π. In Fig. 1(b), path $\langle 5, 6, 8 \rangle$ is satisfiable, while the path $\langle 2, 3, 4 \rangle$ is not satisfiable due to conflicting nodes 2,4.

Recall, that an rl-path in a vpgraph $G_v(\phi)$ corresponds to a vertical path in the vhpform of ϕ. Similarly, an rl-path in a hpgraph $G_h(\phi)$ corresponds to a horizontal path in the vhpform of ϕ. The following corollaries adapt Theorem 1 and Theorem 2 to the graph representations of the vhpform of a given formula ϕ.

Corollary 1. *An assignment σ satisfies ϕ iff there exists a rl-path π in $G_v(\phi)$ such that σ satisfies π.*

Corollary 2. *An assignment σ falsifies ϕ iff there exists a rl-path π in $G_h(\phi)$ such that σ falsifies π.*

The following corollary is a re-statement of corollary 1.

Corollary 3. *ϕ is satisfiable iff there exists a rl-path π in $G_v(\phi)$ which is satisfiable.*

The following corollary connects the notion of conflicting nodes with the satisfiability of a path.

Corollary 4. *A path π in $G(\phi)$ is satisfiable iff no two nodes on π are conflicting.*

Discovery of unit literals from hpgraph: Modern SAT solvers operating on a cnf representation employ a *unit literal rule* for efficient Boolean constraint propagation. The unit literal rule states that if all but one literal of a clause are set to false, then the unassigned literal in the clause must be set to true under the current assignment. In our context the input formula is not necessarily represented in cnf, however, it is still possible to obtain the unit literal rule via the use of the hpgraph of a given formula. The following claim states the unit literal rule for the non-clausal formulas.

Corollary 5. *Let an assignment σ falsify all but a subset of nodes V_s on an rl-path π in $G_h(\phi)$. If all nodes in V_s contain the same literal l and l is not already assigned by σ, then l must be set to true under σ in order to obtain a satisfying assignment.*

The above corollary follows from Theorem 3(b). Intuitively, each rl-path in the hpgraph corresponds to a clause in the cnf of a given formula. Thus, at least one literal from each rl-path in $G_h(\phi)$ must be satisfied in order to obtain a satisfying assignment.
Example: Consider the hpgraph shown in Fig. 1(c) and an assignment σ which sets p, q to false and s to true. σ falsifies all but node 6 on the rl-path $\langle 1, 2, 6, 7 \rangle$ in the hpgraph. It follows from Corollary 5 that $Lit(6)$ which is r must be set to true under σ.

Input: vpgraph $G_v(\phi) = (V, R, L, E, Lit)$ and hpgraph $G_h(\phi) = (V', R', L', E', Lit')$
Output: If $G_v(\phi)$ has a satisfiable rl-path return `SAT`, else return `UNSAT`
Algorithm:

1: $st \leftarrow R$	//push all roots in $G_v(\phi)$ on stack st
2: $\sigma \leftarrow \emptyset$	//initial truth assignment is empty
3: $\forall n \in V : mrk(n) \leftarrow false$	//all nodes are un-marked to start with
4: while ($st \neq \emptyset$)	//stack st is not empty
5: $m \leftarrow st.top()$	// top element of stack st
6: if ($mrk(m)$ == $false$)	//can we extend current r-path `CRP` with m
7: if (`conflict` == `prune`())	//check if taking m causes conflict
8: `learn`()	//compute reason for conflict and learn
9: `backtrack` ()	//non-chronological backtracking
10: `continue`	//goto while loop (line 4)
11: end if	
12: $mrk(m) \leftarrow true$	//extend current satisfiable r-path with m
13: $\sigma \leftarrow \sigma \cup \{Lit(m)\}$	//add $Lit(m)$ to current assignment
14: if ($m \in L$)	//node m is a leaf
15: return `SAT`	//we found a satisfiable rl-path in $G_v(\phi)$
16: else	
17: push all children of m on st	//try extending current r-path `CRP`$\langle m \rangle$ to reach a leaf
18: end if	
19: else	//backtracking mode
20: `backtrack` ()	//non-chronological backtracking
21: end if	
22: end while	
23: return `UNSAT`	//no satisfiable rl-path exists in $G_v(\phi)$

Fig. 2. Searching a vpgraph for a satisfiable rl-path.

4 Top level algorithm

In order to check the satisfiability of a nnf formula ϕ, we obtain a vpgraph $G_v(\phi)$. From Corollary 3 it follows that ϕ is satisfiable iff $G_v(\phi)$ has a satisfiable rl-path. At a high level our search algorithm enumerates all possible rl-paths until a satisfiable rl-path is found. If no satisfiable rl-path is found, then ϕ is unsatisfiable. For dnf (or dnf-like) formulas the number of rl-paths in vpgraph is small, linear in the size of the formula, and therefore the basic search algorithm is efficient. However, for formulas that are not in dnf form, the algorithm of just enumerating all rl-paths in $G_v(\phi)$ does not scale. We have adapted several techniques found in modern SAT solvers such as *search space pruning*, *conflict driven learning*, *non-chronological backtracking* to make the search efficient. Search space pruning and conflict driven learning will be described in detail in the following sections. We present non-chronological backtracking in the appendix of the paper.

The high level description of the algorithm is given in Fig. 2. The input to the algorithm is a vpgraph $G_v(\phi) = (V, R, L, E, Lit)$ and a hpgraph $G_h(\phi) = (V', R', L', E', Lit')$ corresponding to a formula ϕ. If $G_v(\phi)$ contains a satisfiable rl-path, then the algorithm returns `SAT` as the answer. Otherwise, ϕ is unsatisfiable and the algorithm returns `UNSAT`. The algorithm uses the hpgraph $G_h(\phi)$ in various sub-routines such as `prune` and `learn`. The following data structures are used:

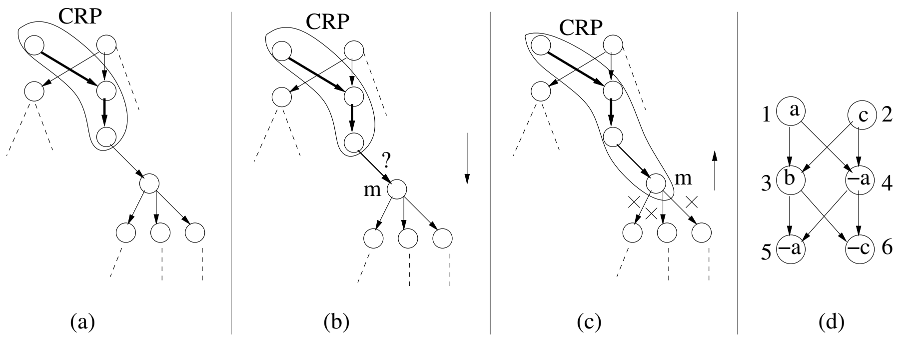

Fig. 3. (a) Current r-path or CRP in a vpgraph (b) Can CRP be extended by node m? (c) Backtracking from node m. (d) vpgraph for formula $(a \vee c) \wedge (b \vee \neg a) \wedge (\neg a \vee \neg c)$

- st is a stack. It stores a subset of nodes from V that need to be explored when searching for a satisfiable rl-path in $G_v(\phi)$. Initially, the roots in $G_v(\phi)$ are pushed on the stack st (line 1). Let $st.top()$ return the top element of st. We write st as $[n_0, \ldots, n_k]$ where the top element is n_k and the bottom element is n_0.
- σ stores the current truth assignment as a set. Each element of σ is a literal which is true under the current assignment. It is ensured that σ is *consistent*, that is, it does not contain contradictory pairs of the form l and $\neg l$. Initially, σ is the empty set (line 2). For example, an assignment with sets variables a, b to true and c to false will be denoted as $\{a, b, \neg c\}$.
- mrk maps a node in V to a Boolean value. It identifies an r-path in $G_v(\phi)$ which is currently being considered by the algorithm to obtain a satisfiable rl-path (see Fig. 3(a)). We refer to this r-path as the *current r-path* (CRP for short). Intuitively, $mrk(n)$ is true for nodes that lie on CRP ($n \in$ CRP) and false for all other nodes in $G_v(\phi)$. More precisely, the CRP is obtained by removing every node n from the stack st for which $mrk(n)$ is false. The remaining nodes constitute the CRP. Initially, $mrk(n)$ is set to false for every node n (line 3), thus, CRP is empty.

Example: The vpgraph for the formula $\phi = (a \vee c) \wedge (b \vee \neg a) \wedge (\neg a \vee \neg c)$ is shown in Fig. 3(d). Initially, we have st as $[2, 1]$ where the top element of the stack is 1, $\sigma = \emptyset$, $mrk(n) = false$ for all $n \in \{1, 2, 3, 4, 5, 6\}$. Suppose during the execution of the algorithm we have st as $[2, 1, 4, 3, 6, 5]$, and $mrk(1), mrk(3)$ are $true$ and $mrk(n) = false$ for $n \in \{2, 4, 5, 6\}$. Thus, CRP is $\langle 1, 3 \rangle$. Observe that CRP is an r-path. Intuitively, the algorithm tries to extend CRP by one node at a time, to obtain a satisfiable rl-path. In this case CRP can be extended to obtain two rl-paths $\pi_1 = \langle 1, 3, 5 \rangle$ or $\pi_2 = \langle 1, 3, 6 \rangle$. However, only π_2 is satisfiable (by $\sigma = \{a, b, \neg c\}$) and is enough to show that ϕ is satisfiable.

The main part of the algorithm is the `while` loop (lines 4-22) which executes as long as st is not empty and the algorithm has not returned SAT on line 15. The algorithm maintains the following loop invariant.

Loop invariant: At the beginning of iteration number i of the `while` loop: let the current r-path (CRP) be $\langle n_0, \ldots, n_k \rangle$. Then the assignment σ is equal to $\{Lit(n_i) | n_i \in$ CRP$\}$. That is, σ satisfies each node on CRP and thus, σ satisfies CRP. For example, suppose CRP is $\langle 1, 3 \rangle$ in the vpgraph shown in Fig. 3(d), then σ will be $\{a, b\}$.

If st is not empty, then the top element of the stack (denoted by m) is considered in line 5. There are two possibilities for node m according to the `if` statement in line 6.

• $mrk(m)$ is $false$: In this case the algorithm checks if the current r-path `CRP` can be extended by node m as shown in Fig. 3(b). This check is carried out by a call to `prune` (line 7). If `prune` returns `conflict`, then the current r-path extended by node m cannot lead to a satisfiable rl-path. Thus, the solver needs to backtrack from node m, and if possible extend `CRP` by some other node. This is done by calling `backtrack` on line 9 and going back to `while` loop (line 4) by using `continue` (line 10). Before backtracking a call to `learn` (line 8) is made which summarizes the reason for the conflict when `CRP` is extended by m. This reason is learned in form of a clause and is used later to avoid similar conflicts. We denote `CRP` concatenated with m as `CRP`$\langle m \rangle$. Depending upon the reason why there is no satisfiable rl-path with `CRP`$\langle m \rangle$ as prefix, the `backtrack` routine can pop several nodes from st (non-chronological backtracking) instead of just popping m from st.

If a call to `prune` results in `no-conflict` (line 7), then m can extend `CRP`. In this case execution reaches line 12. At line 12 $mrk(m)$ is set to true, which means that the new current r-path is `CRP` concatenated with m, that is, `CRP`$\langle m \rangle$. The algorithm maintains the loop invariant that the assignment σ satisfies the current r-path. In order to maintain this invariant σ now needs to satisfy node m which is on the current r-path `CRP`$\langle m \rangle$. This is done by adding $Lit(m)$ to σ (line 13). If m is a leaf in the vpgraph, then `CRP`$\langle m \rangle$ is a satisfiable rl-path. In this case `SAT` is returned (lines 14-15). If m is not a leaf, then the children of m are pushed on the stack (line 17). The algorithm will next attempt to extend the current r-path `CRP`$\langle m \rangle$.

• $mrk(m)$ is $true$: This happens when the current r-path is of the form $\langle n_0, \ldots, n_k, m \rangle$. Intuitively, the algorithm has explored all possible rl-paths with $\langle n_0, \ldots, n_k, m \rangle$ as prefix, but none of them leads to a satisfiable rl-path as shown in Fig. 3(c). The algorithm now backtracks from node m by calling `backtrack` on line 20 . Depending upon the reason why there is no satisfiable rl-path with $\langle n_0, \ldots, n_k, m \rangle$ as prefix, the algorithm can pop several nodes from st instead of just popping m.

For each node n removed from the stack during backtracking (lines 9, 20) $mrk(n)$ is set to false again. This enables the removed nodes to be examined again on rl-paths which have not yet been explored.

5 Search space pruning

This section describes the procedure `prune` called in the non-clausal SAT algorithm shown in Fig. 2 (line 7). A call to `prune` checks if the current r-path `CRP` can be extended by node m or not, as shown in Fig. 3(b). Intuitively, `prune` returns `conflict` if there cannot be a satisfiable rl-path in vpgraph $G_v(\phi)$ with `CRP`$\langle m \rangle$ as prefix. When `prune` is called, the current r-path `CRP` is satisfied by assignment σ, which is equal to $\{Lit(n) | n \in \texttt{CRP}\}$ (maintained as a `while` loop invariant in the top level algorithm shown in Fig. 2). The three cases when `conflict` is returned are as follows:

Case 1: When `CRP`$\langle m \rangle$ is not satisfiable. This happens when there is a node n on `CRP` such that $Lit(n) = \neg Lit(m)$. In this case no assignment can satisfy the r-path `CRP`$\langle m \rangle$ (Corollary 4). For example, in the vpgraph shown in Fig. 4(a) this conflict arises when the `CRP` is $\langle 1, 3 \rangle$ and m is node 5.

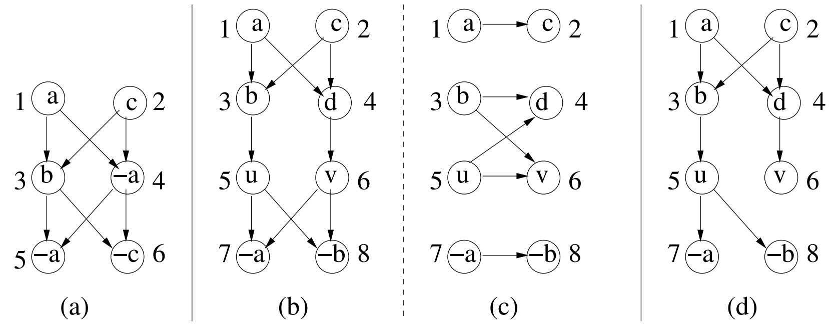

Fig. 4. (a) vpgraph for formula $(a \vee c) \wedge (b \vee \neg a) \wedge (\neg a \vee \neg c)$. (b,c) vpgraph and hpgraph for formula $(a \vee c) \wedge ((b \wedge u) \vee (d \wedge v)) \wedge (\neg a \vee \neg b)$, respectively (d) vpgraph for formula $(a \vee c) \wedge ((b \wedge u \wedge (\neg a \vee \neg b)) \vee (d \wedge v))$

Otherwise, $\texttt{CRP}\langle m\rangle$ is satisfiable and $\sigma' = \sigma \cup \{Lit(m)\}$ satisfies $\texttt{CRP}\langle m\rangle$. However, it is still possible that there is no satisfiable rl-path in $G_v(\phi)$ with $\texttt{CRP}\langle m\rangle$ as prefix. These cases are described below.

Case 2 (Global conflict): When σ' falsifies ϕ. In this case no satisfiable rl-path in $G_v(\phi)$ can be obtained with $\texttt{CRP}\langle m\rangle$ as prefix. We prove this claim by contradiction. Assume that there is an rl-path π in $G_v(\phi)$ which has $\texttt{CRP}\langle m\rangle$ as prefix and is satisfiable. By definition there exists an assignment σ'' which satisfies π. From Corollary 1 we know that σ'' satisfies ϕ. In order to satisfy π, σ'' must satisfy $\texttt{CRP}\langle m\rangle$. That is, σ'' must contain $Lit(n)$ for every $n \in \texttt{CRP}\langle m\rangle$. Since $\sigma' = \{Lit(n) | n \in \texttt{CRP}\langle m\rangle\}$, it follows that $\sigma' \subseteq \sigma''$. But σ' falsifies ϕ and hence σ'' must falsify ϕ. This leads to a contradiction.

Example: In Fig. 4(b) vpgraph for formula $\phi := (a \vee c) \wedge ((b \wedge u) \vee (d \wedge v)) \wedge (\neg a \vee \neg b)$ is given. Consider the case when $\texttt{CRP}$ is $\langle 1\rangle$ and $\sigma = \{a\}$. The algorithm checks if $\texttt{CRP}$ can be extended by node 3 ($m = 3$). Using our notation $\sigma' = \{a, b\}$. Observe that σ' falsifies ϕ by substituting $a = true, b = true$ in ϕ. There are two rl-paths $\pi_1 := \langle 1, 3, 5, 7\rangle, \pi_2 := \langle 1, 3, 5, 8\rangle$ in the vpgraph shown in Fig. 4(b) which have $\langle 1, 3\rangle$ as prefix. Neither of these rl-paths is satisfiable: π_1 is not satisfiable due to conflicting nodes 1, 7 and π_2 is not satisfiable due to conflicting nodes 3, 8.

Detection of global conflict: We use Corollary 2 to check if σ' falsifies ϕ. We check if there is an rl-path π in $G_h(\phi)$ such that σ' falsifies π. Continuing the above example, the hpgraph corresponding to ϕ is shown in Fig. 4(c). Observe that $\sigma' = \{a, b\}$ falsifies the rl-path $\langle 7, 8\rangle$ in Fig. 4(c). Thus, using Corollary 2, it follows that σ' falsifies ϕ.

If there is no global conflict, then the set of implied assignments can be found by the application of unit literal rule on $G_h(\phi)$ as described in Corollary 5.

Case 3 (Local conflict): This conflict arises when every rl-path in $G_v(\phi)$ with $\texttt{CRP}\langle m\rangle$ as prefix contains two nodes which are conflicting and one of the conflicting nodes lies on $\texttt{CRP}\langle m\rangle$. Formally, this conflict arises when for every rl-path π in $G_v(\phi)$ with $\texttt{CRP}\langle m\rangle$ as prefix there exist two nodes $k, l \in \pi$ and $k \in \texttt{CRP}\langle m\rangle$ such that $Lit(k) = \neg Lit(l)$. From Corollary 4, it follows that any rl-path π containing conflicting nodes is not satisfiable. Thus, when a local conflict occurs no rl-path in $G_v(\phi)$ with $\texttt{CRP}\langle m\rangle$ as

prefix is satisfiable. Whenever there is a global conflict (case 2 above) there is also a local conflict, however, the reverse need not hold as shown by the example below.
Example: In Fig. 4(d) the vpgraph for formula $\phi := (a \vee c) \wedge ((b \wedge u \wedge (\neg a \vee \neg b)) \vee (d \wedge v))$ is shown. Consider the case when `CRP` is $\langle 1 \rangle$ and m is node 3 ($m = 3$). Using our earlier notation $\sigma' = \{a, b\}$. Note that σ' does not falsify ϕ, which means there is no global conflict. There are two rl-paths $\langle 1, 3, 5, 7 \rangle$, $\langle 1, 3, 5, 8 \rangle$ in the vpgraph shown in Fig. 4(d) which have $\langle 1, 3 \rangle$ as prefix. Both of these rl-paths contain two conflicting nodes, nodes 1,7 are conflicting on $\langle 1, 3, 5, 7 \rangle$ and nodes 3,8 are conflicting on $\langle 1, 3, 5, 8 \rangle$. Thus, there is a local conflict and the solver needs to backtrack from node $m = 3$.

Detection of global and local conflicts can be done in linear time as described in the appendix. Depending upon the type of conflict (global or local) we perform global or local learning as described below.

6 Learning

Learning records the cause of a conflict. This enables the preemption of similar conflicts later on in the search. In the following, a *clause* will refer to a disjunction of literals. A clause C is *conflicting* under an assignment σ iff all literals in C are falsified by σ. If a clause C is not conflicting under an assignment σ, we say C is *consistent* under σ. We distinguish between two types of learning:
Global learning: A *globally learned* clause is a clause whose consistency must be maintained irrespective of the current search state, which is given by the current r-path `CRP` (and assignment $\sigma = \{Lit(n) | n \in \texttt{CRP}\}$). That is, whenever a globally learned clause becomes conflicting under σ the solver abandons the current search state and backtracks. A globally learned clause is generated from a conflicting clause. A conflicting clause C arises in two cases as described below.
1) When analyzing global conflicts as described in the previous section. When a global conflict occurs there is an rl-path π in hpgraph $G_h(\phi)$ which is falsified by the assignment σ currently under consideration. The set of literals corresponding to the nodes on π gives us a clause $C := \bigvee_{n \in \pi}(Lit(n))$. Observe that C is a conflicting clause, that is, all literals occurring in C are set to false under the current assignment.
Example: The hpgraph corresponding to $\phi := (a \vee c) \wedge ((b \wedge u) \vee (d \wedge v)) \wedge (\neg a \vee \neg b)$ is shown in Fig. 4(c). A global conflict occurs when the current assignment is $\sigma = \{a, b\}$, that is, σ falsifies ϕ. In this case the rl-path in the hpgraph which is falsified by σ is $\langle 7, 8 \rangle$. Thus the required conflicting clause is $\neg a \vee \neg b$.
2) When all literals of an existing globally learned clause C become false.

Once a conflicting clause C is obtained, we perform a 1-UIP (*first unique implication point*) analysis [19] to obtain a learned clause C'. Clause C' is added to the database of globally learned clauses. In order to perform 1-UIP analysis we maintain a notion of a decision level. We associate a decision level $dec(n)$ with each node n in the current r-path `CRP`. We also maintain a set of implied literals at each node (or decision level) along with the reason (set of variable assignments) which led to the implication. We follow the same algorithm as in [19] to perform the 1-UIP learning.
Local learning: A *locally learned* clause is associated to a node n in the vpgraph when a local conflict occurs at n. Suppose C is a locally learned clause at node n. Then the consistency of C needs to be maintained only when n is part of the current search state,

Bench -mark	#Probs	SatMate		MiniSat		BerkMin		Siege		zChaff	
		Time	Solved	Time	Solved	Time	Solved	Time	Solved	Time	Solved
QG6	256	23266	235	49386	179	46625	184	46525	184	47321	180
QG6*	256	23266	235	37562	211	15975	239	30254	225	45557	186
Mboard	19	4316	12	4331	12	4947	11	4505	12	5029	11
Pigeon	19	5110	11	6114	9	5459	10	6174	9	5483	11

Table 1. Comparison between SatMate, MiniSat, BerkMin, Siege, zChaff. "Time" gives total time in seconds and "Solved" gives #problems solved within timeout of 600 seconds/problem.

that is, $n \in \texttt{CRP}$. If n does not lie on CRP, then the consistency of C is irrelevant. This is in contrast to a globally learned clause whose consistency must always be maintained.
Example: Consider the local conflict which occurs in the vpgraph in Fig. 4(d) when CRP is $\langle 1 \rangle$ and it is checked if CRP can be extended by $m = 3$. In this case every rl-path in vpgraph with $\langle 1, 3 \rangle$ as prefix contains two conflicting nodes one of which lies on $\langle 1, 3 \rangle$. The rl-path $\langle 1, 3, 5, 7 \rangle$ has conflicting nodes 1,7 and the rl-path $\langle 1, 3, 5, 8 \rangle$ has conflicting nodes 3,8. In this case a clause $Lit(7) \vee Lit(8) = \neg a \vee \neg b$ can be learned at node 3. Intuitively, when we consider extending the CRP with node m the (locally) learned clauses at node m must be consistent with the assignment $\sigma = \{Lit(n) | n \in \texttt{CRP}\langle m \rangle\}$. Otherwise, a local conflict will occur at m causing the solver to backtrack. Having learned clauses at node m avoids repeating the work done in detecting the same local conflict. For the vpgraph in Fig. 4(d), when CRP is $\langle 2 \rangle$ and $m = 3$, $\sigma = \{c, b\}$ is consistent with the learned clause $\neg a \vee \neg b$ at node 3, thus, the solver cannot get the same local conflict at node 3 as before (when CRP was $\langle 1 \rangle$ and $m = 3$).

If a local conflict occurs when extending CRP by node m, then a clause is learned at node m as follows: For each rl-path π having $\texttt{CRP}\langle m \rangle$ as prefix let $\omega_1(\pi), \omega_2(\pi)$ denote the pair of conflicting nodes on π. Without loss of generality assume that $\omega_1(\pi)$ lies on $\texttt{CRP}\langle m \rangle$. Then the learned clause C at node m is given by $\bigvee_{\pi} Lit(\omega_2(\pi))$. Consistency of C must be maintained only when considering rl-paths passing through m.

7 Experimental results

The experiments were performed on a 1.5 GHZ AMD machine with 3 GB of memory running Linux. The techniques described in the paper have been implemented in a SAT solver called SatMate [3]. The non-clausal input formula is given in EDIMACS [1] or ISCAS format. SatMate also accepts cnf inputs in DIMACS format. We compare SatMate against four state-of-the-art cnf SAT solvers MiniSat version 1.14 [2], BerkMin version 561 [10], Siege version 4 [4], and zChaff version 2004.5.13 [14].
QG6 benchmarks: The authors of [13] provided us with a benchmark set called QG6 which consists of 256 non-clausal formulas of varying difficulty. These benchmarks were generated during the construction of classification theorems for quasigroups [13]. The cnf version of these problems was also made available to us by the authors of [13]. The cnf version was obtained by directly expressing the problem of classifying quasigroups into cnf as opposed to the translation of non-clausal formulas into cnf. The non-clausal versions of these benchmarks have 300 variables and 7500 gates (AND, OR gates) on average, while the cnf versions have 1700 variables and 7500 clauses on average. We ran SatMate on the non-clausal formulas and cnf SAT solvers on the corresponding cnf formulas from QG6 suite.

Problem	SatMate			MiniSat	BerkMin	Siege	zChaff
	Time	Local confs	Global confs	Time	Time	Time	Time
dnd02	**174**	23500	15588	1308	1085	1238	TO
brn13	**181**	20699	20062	1441	1673	1508	TO
icl39	**200**	22683	14069	TO	TO	2629	TO
icl45	TO	4850	72106	TO	2320	**1641**	TO
q2.14	237	113	15863	**23**	24	34	88
cache.inv12	58	659	7131	**1**	**1**	**1**	2

Table 2. Comparison on individual benchmarks. Timeout is 1 hour per problem per solver. "Time" sub-column gives time taken in seconds.

QG6* benchmarks: We translated the non-clausal formulas from the QG6 suite into cnf by introducing new variables [15]. The cnf formulas obtained after translation have 7500 variables and 30000 clauses on average. We ran cnf SAT solvers on the cnf formulas obtained after translation. Note that we still ran SatMate on the non-clausal formulas.
Mboard benchmarks: encode the mutilated-checkerboard problem.
Pigeon benchmarks: encode the pigeon hole principle with n holes and $n+1$ pigeons.

Both QG6 and QG6* benchmarks contain a mixture of satisfiable and unsatisfiable problems. All problems in the Mboard and Pigeon benchmarks are unsatisfiable.

The experimental results are summarized in Table 1. The column "#Probs" gives the number of problems in each benchmark set. There was a timeout of 10 minutes per problem per solver. For each solver we report two quantities: 1) "Time" is the total time spent in seconds when solving problems in a given benchmark, including the time spent (= timeout) for each instance not solved within timeout. 2) "Solved" gives the total number of problems that were solved within timeout.
Summary of results in Table 1: On QG6 benchmarks SatMate solves around 50 more problems and it is approximately 2 times faster than the cnf SAT solvers MiniSat, BerkMin, Siege, and zChaff. On QG6* benchmarks SatMate performs better than MiniSat, zChaff, Siege. However, BerkMin outperforms SatMate on QG6* benchmarks. The difference in the performance of cnf SAT solvers on QG6 and QG6* benchmarks shows how the differences in the encoding of a given problem to cnf can significantly impact the performance of cnf SAT solvers. The performance of SatMate on Mboard and Pigeon benchmarks is slightly better than the cnf SAT solvers.

Table 2 summarizes the performance of SatMate and four cnf SAT solvers on various individual problems. Problems `dnd02, brn13, icl39, icl45` are from QG6 benchmark suite. Problems `q2.14,cache.inv12` are generated by a verification tool. The sub-column "Time" gives the time required for SAT solving (in seconds). For SatMate we report the number of local conflicts and the number of global conflicts (Section 5) in the "Local confs" and "Global confs" sub-columns, respectively. A timeout of 1 hour was set per problem. We denote timeout by "TO". In case of timeout we report the number of conflicts just before the timeout for SatMate.

Performance of SatMate is correlated with the number of local conflicts and global conflicts. A local conflict is a conflict that occurs in a part of a formula and it depends on the structure of the vpgraph. There is no equivalent of local conflict in cnf SAT solvers. In cnf SAT solvers a conflict arises when the current assignment falsifies an original/learned clause which is equivalent to a global conflict. As shown in Table 2 the number of local conflicts is usually comparable to the number of global conflicts on the benchmarks where SatMate outperforms cnf SAT solvers. Indeed the performance of SatMate degrades if no local conflict detection and local learning is done.

8 Conclusion

We presented a new non-clausal SAT solver based on the General Matings approach. This approach involves the search for a vertical path which does not contain opposite literals in the vertical-horizontal path form (vhpform) of a given negation normal form formula. The main challenge in obtaining an efficient SAT solver based on the General Matings approach is to prevent the enumeration of vertical paths. We presented techniques for preventing the enumeration of vertical paths and graph based representations of the vhpform for efficient implementation of these ideas. Experimental results show that on a class of non-clausal benchmarks our SAT solver has a performance comparable to the current state-of-the-art cnf SAT solvers. Overall, our results show the promise of the General Matings approach in building SAT solvers.
Acknowledgment: We thank Peter Andrews for his useful comments and Malay Ganai, Guoqiang Pan, Sanjit Seshia, Volker Sorge for providing us with benchmarks.

References

1. Edimacs format, www.satcompetition.org/2005/edimacs.pdf.
2. Minisat sat solver, http://www.cs.chalmers.se/cs/research/formalmethods/minisat/.
3. SatMate website, http://www.cs.cmu.edu/∼modelcheck/satmate.
4. Siege (version 4) sat solver, http://www.cs.sfu.ca/∼loryan/personal/.
5. TPS and ETPS, http://gtps.math.cmu.edu/tps-papers.html.
6. Peter B. Andrews. Theorem Proving via General Matings. *J. ACM*, 28(2):193–214, 1981.
7. Peter B. Andrews. *An Introduction to Mathematical Logic and Type Theory: to Truth through Proof*. Kluwer Academic Publishers, Dordrecht, second edition, 2002.
8. Wolfgang Bibel. On Matrices with Connections. *J. ACM*, 28(4):633–645, 1981.
9. 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.
10. E. Goldberg and Y. Novikov. BerkMin: A Fast and Robust Sat-Solver. In *DATE*, 2002.
11. Joao P. Marques-Silva and Karem A. Sakallah. GRASP - A New Search Algorithm for Satisfiability. In *ICCAD*, pages 220–227, 1996.
12. David McAllester, Bart Selman, and Henry Kautz. Evidence for invariants in local search. In *AAAI*, pages 321–326, Providence, Rhode Island, 1997.
13. Andreas Meier and Volker Sorge. A new set of algebraic benchmark problems for sat solvers. In *SAT*, pages 459–466, 2005.
14. 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.
15. David A. Plaisted and Steven Greenbaum. A structure-preserving clause form translation. *J. Symb. Comput.*, 2(3), 1986.
16. R. M. Smullyan. *First Order Logic*. Springer-Verlag, 1968.
17. Christian Thiffault, Fahiem Bacchus, and Toby Walsh. Solving Non-clausal Formulas with DPLL Search. In *SAT*, 2004.
18. H. Zhang. Sato: An efficient propositional prover. In *CADE-14*, pages 272–275, 1997.
19. Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. Efficient conflict driven learning in boolean satisfiability solver. In *ICCAD*, pages 279–285, 2001.

9 Appendix: Algorithms for conflict detection

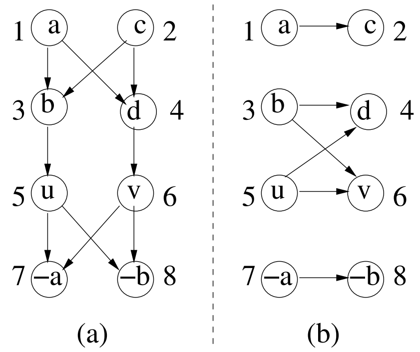

Fig. 5. (a,b) vpgraph and hpgraph for formula $(a \vee c) \wedge ((b \wedge u) \vee (d \wedge v)) \wedge (\neg a \vee \neg b)$, respectively.

9.1 Algorithm for detection of global conflict

We say that a global conflict occurs when an assignment σ falsifies a given formula ϕ. In order to detect this conflict we use Corollary 2. This requires checking if there is an rl-path π in hpgraph $G_h(\phi) = (V', R', L', E', Lit')$ such that σ falsifies π. We present an $O(V' + E')$ algorithm below. We reduce the problem of finding an rl-path π in $G_h(\phi)$ such that σ falsifies π to a shortest path computation problem as follows: It is assumed that σ is consistent, that is, it does not contain opposite literals. For each node $n \in V'$ we assign a weight $w(n) \in \{0, 1, 2\}$ to n. The assignment of weights is done as follows:

$$w(n) := \begin{cases} 0 : \neg Lit(n) \in \sigma \\ 2 : Lit(n) \in \sigma \\ 1 : \text{otherwise} \end{cases}$$

We will use the hpgraph in Fig. 5(b) as our running example. If $\sigma = \{a, b\}$, then the weight assigned to various nodes in the hpgraph is as follows: $w(1) = 2, w(2) = 1, w(3) = 2, w(4) = 1, w(5) = 1, w(6) = 1, w(7) = 0, w(8) = 0$.

Given a path π in hpgraph, we define the length of π to be the sum of weights of the nodes that lie on π. For each node n in the hpgraph we compute a shortest path estimate $\delta(n)$ which represents the length of shortest path from any root node to n. We also track the parent $par(n)$ of each node n in the shortest path to n. For the hpgraph in Fig. 5(b) and $\sigma = \{a, b\}$, we have $\delta(1) = 2, \delta(2) = 3, \delta(3) = 2, \delta(4) = 2, \delta(5) = 1, \delta(6) = 2, \delta(7) = 0, \delta(8) = 0$.

If there is an rl-path $\pi := \langle n_1, \ldots, n_k \rangle$ in $G_h(\phi)$ such that σ falsifies π, then $\delta(n_k) = 0$. This is because n_1 is a root node (by definition of rl-path) and every node on π has a weight of 0 because σ falsifies each node on π (as σ falsifies π). Thus, there is a path of length 0 to n_k which is the smallest possible length due to non-negative weights. Observe that n_k is a leaf node by definition of a rl-path. The following claim formalizes the above idea of detecting global conflicts using the shortest path estimates.

Claim. The following statements are equivalent:
1. σ falsifies ϕ.
2. There is an rl-path π in $G_h(\phi)$ such that σ falsifies π.
3. There is a node $n \in L'$ such that $\delta(n) = 0$.

Given a hpgraph and an assignment we compute the shortest path estimates for each node in the hpgraph. If there is a leaf node n in hpgraph such that $\delta(n) = 0$, then there is a global conflict. Otherwise, there is no global conflict. For the hpgraph in Fig. 5(b) and $\sigma = \{a, b\}$, we have $\delta(8) = 0$ and node 8 is a leaf node, it follows from the above claim that σ falsifies ϕ.

Extraction of falsified rl-path: If there is a a leaf node n in hpgraph such that $\delta(n) = 0$, then the actual rl-path (ending at n) which is falsified by σ can be obtained by examining the parent of each node in the shortest path tree starting from node n. We assume that parent of a root node is `nil` node. Then the required rl-path is obtained by reversing the following sequence $\langle n, par(n), par(par(n)), \ldots, \texttt{nil}\rangle$ and removing the `nil` node.

For the hpgraph in Fig. 5(b) and $\sigma = \{a, b\}$, we have $\delta(8) = 0$, $par(8) = 7, par(7) = \texttt{nil}$. Thus, the rl-path in hpgraph which is falsified by σ is $\langle 7, 8\rangle$.

Obtaining unit literals via hpgraph: If there is no leaf node n in hpgraph such that $\delta(n) = 0$, then there is no global conflict. In this case the set of implied assignments under σ can be obtained by applying Corollary 5. More specifically if there is an rl-path $\pi := \langle n_1, \ldots, n_k\rangle$ in $G_h(\phi)$ such that σ falsifies all but one node (say $n_i, 1 \leq i \leq k$) on π and $Lit(n_i)$ is not yet assigned by σ, then $\delta(n_k) = 1$. This is because n_1 is a root node (by definition of rl-path) and every node on π different from n_i has a weight of 0 and n_i has a weight of 1 as $Lit(n_i)$ has not set been assigned by σ. Thus, there is a path of length 1 to n_k which is the smallest possible length given that there is no global conflict, that is, $\delta(n_k) \neq 0$. Observe that n_k is a leaf node by definition of a rl-path. The following claim formalizes the idea of detecting unit literals using the shortest path estimates.

Claim. Assuming σ does not falsify ϕ, the following statements are equivalent:
1. σ falsifies all but one node (say n) on a rl-path π in $G_h(\phi)$ and $Lit(n)$ is not yet assigned by σ.
2. There is a node $n \in L'$ such that $\delta(n) = 1$.

Using the above claim it is possible to extract various implied literals by examining leaf nodes whose shortest path estimate is 1 (assuming no global conflict). If there is a a leaf node n in hpgraph such that $\delta(n) = 1$, then the actual rl-path π whose all but one node is falsified by current assignment σ can be obtained by examining the parent of each node in the shortest path tree starting from node n. This allows obtaining both the implied literal l and *unit-clause* (literals corresponding to nodes on π) which implied l. Example: For the hpgraph in Fig. 5(b) and $\sigma = \{a\}$. The shortest path estimates for various nodes are as follows: $\delta(1) = 2, \delta(2) = 3, \delta(3) = 1, \delta(4) = 2, \delta(5) = 1, \delta(6) = 2, \delta(7) = 0, \delta(8) = 1$. Observe that none of the leaf nodes $n \in \{2, 4, 6, 8\}$ has $\delta(n) = 0$. Thus, there is no global conflict. However, $\delta(8) = 1$ which means that there is an rl-path in hpgraph such that σ falsifies all but one node of this rl-path (using above claim).

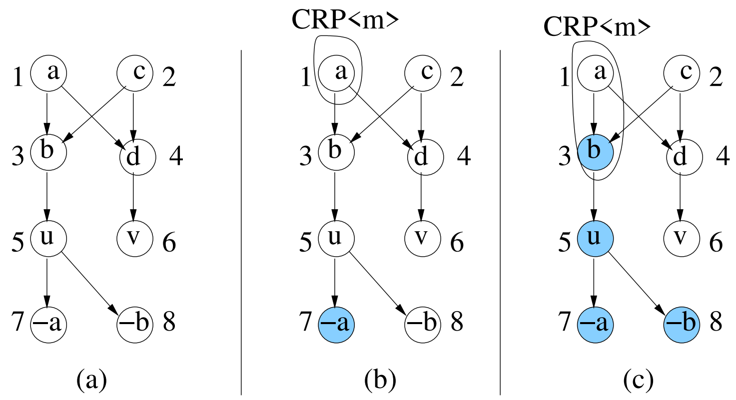

Fig. 6. (a) vpgraph for formula $(a \vee c) \wedge ((b \wedge u \wedge (\neg a \vee \neg b)) \vee (d \wedge v))$ (b) assignment of conflict labels when `CRP` is $\langle\rangle$ and $m = 1$. A colored node n denotes $conf(n)$ is true. (c) assignment of conflict labels when `CRP` is $\langle 1 \rangle$ and $m = 3$. A local conflict occurs as $conf(3)$ is true.

In this example, the required rl-path is $\langle 7, 8 \rangle$. Using Corollary 5 $Lit(8) = \neg b$ must be set to true under the current assignment to prevent a global conflict.
Efficiency issues: Since the hpgraph is a DAG the computation of $\delta(n)$ for all n can be done in linear time. However, in practice the routine for detecting global conflicts is called very often, and computing $\delta(n)$ for every (un)assignment to a variable is expensive. Thus, we use two optimizations which are crucial for efficiency: 1) incremental shortest path computation: whenever a variable is (un)assigned, instead of computing the shortest path estimate for every node in the hpgraph, we only examine the nodes whose shortest path estimate can get affected due to this assignment. 2) by limiting the range of $\delta(n)$ to only $\{0, 1, \infty\}$.

9.2 Algorithm for detection of local conflict

A local conflict occurs when every rl-path in $G_v(\phi)$ with `CRP`$\langle m \rangle$ as prefix contains two nodes which are conflicting and one of the conflicting nodes lies on `CRP`$\langle m \rangle$. This conflict can be detected by using a linear time algorithm as described below.

Let σ denote the set of literals corresponding to the nodes on `CRP`$\langle m \rangle$. That is, $\sigma = \{Lit(n) | n \in \texttt{CRP}\langle m \rangle\}$. In order to detect a local conflict, we compute for each node n in the vpgraph a flag $conf(n)$. If $conf(n)$ is true for some node n, then no satisfiable rl-path with `CRP`$\langle m \rangle$ as prefix can pass through this node. A local conflict happens when $conf(m)$ becomes true.

We compute $conf(n)$ for every n by scanning the nodes in $G_v(\phi)$ in reverse topological order (recall that $G_v(\phi)$ is a DAG). For each node n we assign $conf(n)$ as follows:
1. If $\neg Lit(n) \in \sigma$, then set $conf(n)$ to true.
2. Else if $conf(n')$ is true for every successor n' of n, then set $conf(n)$ to true.
3. Else set $conf(n)$ to false.

Example: Consider the vpgraph in Fig. 6(a). Suppose CRP is $\langle\rangle$ and $m = 1$. Using the above notation $\sigma = \{a\}$. The assignment of conflict labels sets $conf(7)$ to true and $conf(n)$ to false for all other n as shown in Fig. 6(b). Since $conf(1)$ is false, there is no local conflict when extending CRP by node 1. Now consider the case when CRP is $\langle 1\rangle$ and m is node 3. In this case $\sigma = \{a, b\}$. The assignment of conflict labels sets $conf(n) = true$ for $n \in \{3, 5, 7, 8\}$ to true and $conf(n)$ to false for all other n as shown in Fig. 6(c). Since $conf(3)$ is true, there is a local conflict when extending CRP by node 3.

Efficiency issues: In our implementation we do not carry out the above computation of scanning the entire vpgraph whenever a variable is (un)assigned. Instead, we incrementally update the $conf(n)$ flags by remembering them across multiple calls to the local conflict detection routine. Furthermore, our algorithm only looks at the nodes whose $conf(n)$ flag may get affected due to a variable (un)assignment.

10 Appendix: Non-chronological Backtracking

Analyzing conflicts to determine their causes enables modern SAT solvers to backtrack *non-chronologically* to earlier levels in the search tree, potentially pruning large portions of the search space. This technique is also applicable in our SAT procedure. The backtracking routine called in our SAT procedure depends on the type of conflict that invoked the backtracking procedure:

Non-chronological backtracking on a global conflict: When a global conflict occurs the solver calls a backtracking procedure similar to that in cnf SAT solvers. Suppose a global conflict occurs when the solver attempts to extend the CRP with node m. In this case the learning procedure produces an *asserting clause* C. That is, only one literal in C called the *asserting literal* $al(C)$ is assigned at the current decision level (corresponding to node m), and remaining literals were assigned at earlier decision levels (corresponding to nodes on CRP). The solver identifies the highest decision level $maxd(C)$ among the literals of $C \setminus \{al(C)\}$. The solver backtracks to the node m' corresponding to the decision level $maxd(C)$. At m' the clause C becomes a unit clause and $al(C)$ is set to true. The search proceeds from m' onwards.

Non-chronological backtracking on a local conflict: When a local conflict occurs the solver backtracks non-chronologically by analyzing the structure of the vpgraph and the conflict clause produced due to local conflict.

Example 3: Consider the vpgraph in shown in Fig. 7(a). Let CRP be $\langle 1, 2, 3, 5\rangle$ and the solver examines if CRP can be extended by node 7. Suppose a local conflict occurs at node 7 and a clause $\neg a \vee \neg b \vee \neg c$ is learned at node 7. Assume that all the paths starting from nodes 2, 3, 4, 5, 6 pass through node 7. This conflict can be resolved only if we backtrack to the nodes containing literal a or literal b (assuming c was assigned at node 7). Since node 2 comes later on CRP, our algorithm backtracks to node 2. Note that backtracking prevents us from examining rl-paths such as $\langle 1, 2, 3, 6, 7, \ldots\rangle$, $\langle 1, 2, 4, 5, 7, \ldots\rangle$, $\langle 1, 2, 4, 6, 7, \ldots\rangle$ all of which would lead to the same conflict at node 7. The suffix of CRP starting from node 2 onwards is removed. The search procedure attempts to extend CRP $\langle 1\rangle$ with other unexamined successors

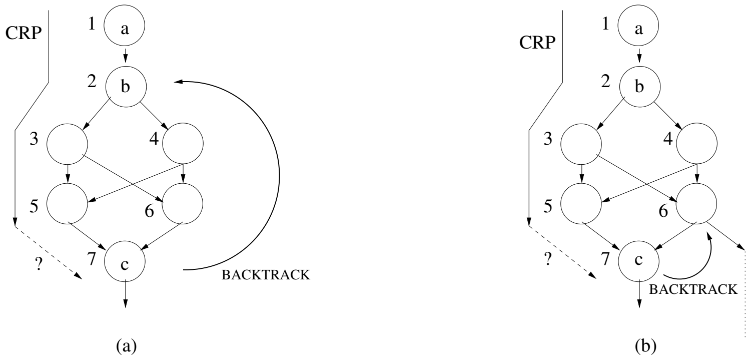

Fig. 7. Non-chronological backtracking example

of node 1. In this case there is no other unexamined successor of node 1, so the solver backtracks from node 1 to return unsatisfiable answer.

Now consider a variation of Fig. 7(a) in Fig. 7(b). Node 6 has an outgoing path that does not pass through node 7. As before, a local conflict occurs at node 7. However, now instead of backtracking to node 2 our algorithm backtracks to node 6. This is because there are alternative rl-paths such as $\langle 1, 2, 3, 6, \ldots \rangle$, $\langle 1, 2, 4, 6, \ldots \rangle$ through node 6 which could be satisfiable.

11 Decision Heuristics

In modern DPLL-based SAT solvers *decision heuristics* play an important role in pruning the search space by identifying the variables to be assigned next. In our algorithm decision heuristics are used to decide the order in which the children of the last node on the `CRP` will be examined. More precisely, we use decision heuristics when pushing the children of m on the stack (Fig. 2, line 17). The children near the end of stack get examined before the other children on the stack.

Some of our decision heuristics make use of *literal activity*. The activity of a literal indicates its usefulness (participation) in conflicts so far. It is updated in a similar manner as in zChaff [14]. The activity n of a node in vpgraph is simply the activity of literal $Lit(n)$. We describe a few decision heuristics used when pushing children of m (line 17) below.

1. Push the children in the order they occur in the adjacency list of m.
2. Push the children of m in a random order.
3. Push the children of m in the ascending order of activity. The higher activity children will be examined before other children.
4. Divide the children of m in two sets S_1 and S_2. Each node $n \in S_1$ has $Lit(n)$ already set to true. S_2 contains the remaining children of m. Push nodes in S_2 in the stack followed by nodes in S_1. Intuitively, the nodes in S_1 are *satisfied* and `prune` will not return a `conflict` for a node in S_1. Thus, a satisfying assignment

or a conflict will be reached quickly by examining the nodes in S_1 before the nodes in S_2.

5. Form the sets S_1 and S_2 as above. Push the nodes in S_2 in ascending order of their activity. Then push nodes in S_1 in ascending order to activity.

In our experiments heuristic five outperforms the other decision heuristics.

12 Conversion of Circuits to Negation Normal Form (NNF)

Our solvers handle formulas containing $\wedge, \vee, \neg$ operators and no structure (sub-formula) sharing directly, without introduction of new variables. Observe that these formulas can easily be converted to nnf by pushing the negations to the variables using DeMorgan's laws. The existing CNF and circuit based SAT solvers require introduction of new variables for each intermediate gate or sub-formula.

We are also able to handle formulas with structure sharing or formulas containing other operators such as if-then-else (ITE), iff ($\Leftrightarrow$), xor ($\oplus$) operators. This is done by converting these formulas to nnf formulas in two stages. The first stage re-writes other operators in terms of $\wedge, \vee, \neg$. For example, $\phi_1 \Leftrightarrow \phi_2$ (iff operator) is written as $(\phi_1 \wedge \phi_2) \vee (\neg\phi_1 \wedge \neg\phi_2)$. 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. It suffices to introduce a new variable for each gate that is shared (fanout ¿ 1). Alternatively, smaller sub-formulas with small fanout can be handled by inlining (duplicating) them.

It must be noted that the entire process of converting an arbitrary Boolean formula (circuit) to Negation Normal Form is linear in the size of the original formula.