2011 Fifth IEEE International Conference on Theoretical Aspects of Software Engineering

An Efficient Resolution Based Algorithm for SAT

Min Zhou
Department of Computer Science and Technologies
Tsinghua University
Beijing, China
Email: zhoumin03@mails.tsinghua.edu.cn

Fei He and Ming Gu
School of Software
Tsinghua University
Beijing, China
Email: hefei/guming@mail.tsinghua.edu.cn

***Abstract*—Propositional satisfiability problem (SAT) is a fundamental problem both in theory and practice. In the area of software engineering, people employ various techniques, such as model checking, theorem proving, automated testing and so on, to ensure the quality of software. Those techniques are usually based on SAT solvers. The efficiency is an important criterion for a good SAT solver. Besides, the ability of producing proofs is also considered to be quite useful because it provides a mechanism that the correctness of checking result is guaranteed. Moreover, proofs can be used when calculating interpolation. In this paper, we investigate a new resolution based algorithm for solving SAT problem. The algorithm combines resolution and search. It resolves certain clauses when necessary and at the same time tries to find a valuation under which the formula evaluates to true. Information found in the process of searching for such a valuation is used to guide the resolution. The algorithm stops whenever a satisfying valuation is found or empty clause is generated. So, it terminates quickly for both satisfiable and unsatisfiable clauses. Compared with other resolution based algorithms, the experiment result shows that the number of resolutions and number of generated clauses are much less than directional resolution. Another major advantage of our algorithm is, once terminates, a proof can be easily generated with very low time complexity.**

I. Introduction

Satisfiability problem of propositional logic is also called SAT. It has been densely studied for many years not only because its theoretical importance but also its practical interest. Nowadays, in software engineering and industry, SAT problem has many applications. For example, SAT-based model checking is used in hardward design, SAT solvers are also invoked when generating some test cases, and so on. It is a fundamental problem both in theory and practice.

There are already some algorithms for solving SAT problem, a few of which are already used in industry. Furthermore, an international competition (official site: http://www.satcompetition.org/) is held annually to provide a platform for comparing various algorithms and encourage researchers to optimize their solvers.

Efficiency is probably the most important thing when solving an SAT problem. However, it is not enough. When SAT solvers are used as underlying tools in model checking, the ability of generating counter examples and proofs is required. Furthermore, the counter example and proof should be as small as possible.

Historically, it is proved by Cook [1] in 1971 that 3-SAT problem is NP. What we call k-SAT is a sub-category of SAT problem where each clause is given with k literals. It is not yet proved but generally believed that completely solving NPC problem requires at least exponential time complexity. So, generally speaking, solving large SAT problems is hard. However, the applications of SAT problem have been widely spread in industry and electronic design. Therefore, algorithms that are able to give a result in an acceptable time are of great practical interest.

A. Related Work

The syntax and semantic of propositional logic are concise. Because of its simplicity, there are lots of research in reducing other problems to SAT, then one can invoke SAT solvers to solve them, such as in [2]. In industry or software engineering, some researchers try to automate test case generation using SAT, as can be found in [3].

SAT problems can be classified into some subcategories. For instance, 3-SAT problem is a class of SAT problems where each clause contains 3 literals. It is the first problem proved to be NP-Complete. 2-SAT problem where each clause consists of 2 literals is linear time solvable. SAT problem which involves only Horn Clauses (where each clause contains at most one positive literal.) is solvable in polynomial time [4].

However, solving arbitrary SAT problem requires some technique. Typically, SAT algorithms can be classified into two categories. One is search based algorithms, such as DPLL [5]. Tools based on DPLL algorithm are available, such as: Chaff [6], PicoSAT, MiniSAT [7], and so on. In DPLL, conflict clauses analysis and clause learning are adopted so as to enhance the performance. The main flow of this kind of algorithm is to look up for a satisfiable valuation. So, they are good for most satisfiable cases. But their major disadvantage is the difficulty of generating proofs for unsatisfiable cases. The other category of algorithm is based on resolution. For instance, DP algorithm [8] which deals with CNF formulas by exhaustively apply resolution until saturation (can not find any new resolvent) or empty clause is derived. An improved version of DP algorithm is Directional Resolution (DR) [9] which defines an order on those literals and restricts the resolution to be on the maximal literal. The experiment result shows DR performs significantly better than the naive DP algorithm (will be explained later in this paper). However, for satisfiable cases, both of them take quite long time to finish all resolutions.

978-0-7695-4506-6/11 $26.00 © 2011 IEEE
DOI 10.1109/TASE.2011.23

There are also researches on combining search and resolution. In [10], the author presented two hybrid approaches. The BDR-DP(i) approach performs bounded resolution prior to search, while the other scheme called DCDR(b) use resolution dynamically during search. Although combined at a high level, both of them yield better performances.

In real application, knowing the status of satisfiable or unsatisfiable is not enough. For many reasons, proofs are also needed. In model checking, proofs and counter examples are used to calculate interpolation or to refine the model. On the other hand, which is a realistic problem that, there is no guarantee that the SAT solver one invoked is correctly implemented. So, providing the proof along with the satisfiable or unsatisfiable result can help. In this case, the proof serves as a certification for the output result. For satisfiable case, a proof can be a satisfying valuation. For unsatisfiable case, a proof can be a trace $C_1, C_2, \ldots, C_N$ such that each C_k is either an initial clause in the input formula or resolved from C_i, C_j, where $i, j < k$. C_N should be the empty clause "$\square$". For specialized algorithm, the format of proof can be in a more concise form, such as a class of linear regular resolutions [11]. Some certified theorem prover, such as Coq, can read and verify if such proof supports the status of problem.

The contribution of this paper is in introducing an efficient resolution based SAT algorithm and providing experiment comparisons. The idea of the algorithm is explained and the algorithm procedures are also described in details. More importantly, proofs of soundness and completeness are provided.

The rest of this paper is organized as follows: In Sect. II, some notations are introduced. The algorithm is explained and described in Sect. III. Then the soundness and completeness is discussed in Sect. IV. In Sect. V, we talked about some implementation issues that readers may concern. Finally, the complexity analysis and experiments are shown in Sect. VI followed by the conclusion.

II. Preliminaries

We assume that readers are familiar with propositional logic. In propositional logic, Boolean values are $\mathcal{B} = \{1, 0\}$. *Atoms* are propositional variables range over $\mathcal{B}$, in this paper, they are denoted by uppercase letters. $\mathcal{A}$ is the set of all atoms. Especially, we use $\top$ ($\bot$) to denote the atom which is always *true* (*false*). A *literal* is a positive or negative form of atom such as $P, \neg P$. A *clause* is a disjunction of literals such as $P \vee \neg Q \vee R$. Clauses can be viewed as sets of literals, e.g. the previous clause can be written $\{P, \neg Q, R\}$; A *formula*(CNF) is a set of clauses. A *valuation* is a map $\mathcal{A} \mapsto \mathcal{B}$, i.e. from atom set $\mathcal{A}$ to Boolean values $\{1, 0\}$. A valuation can be interpreted on formulas intuitively. Use $v(t)$ to denote the value of t under valuation v. For convenience, a valuation can be represented as a subset of $\mathcal{A}$ i.e. the subset of atoms whose value are 1 under v. Thus, we also write v the set $\{P \in \mathcal{A} | v(P) = 1\}$. A *model* for a formula f is a valuation v such that $v(f) = 1$. A formula f is *satisfiable* iff it has at least one model. Two formulas f and g are *equiv-satisfiable* iff f is satisfiable $\iff$ g is satisfiable.

TABLE I
An example of resolution principle

Num	Clause	Note
(5)	P	Resolvent of (1) and (2)
(6)	Q	Resolvent of (1) and (3)
(7)	$\top$	Resolvent of (1) and (4)
(8)	$\top$	Resolvent of (2) and (3)
(9)	$\neg Q$	Resolvent of (2) and (4)
(10)	$\neg P$	Resolvent of (3) and (4)
(11)	$\square$	Resolvent of (5) and (10)

In this paper we consider satisfiability problem of CNF formulas. By introducing new variables, it is possible to convert an arbitrary formula f to an equiv-satisfiable CNF formula $\hat{f}$ in linear time [4]. Besides, the number of introduced new variables is linear to the size of f. Thus, SAT problem on an arbitrary propositional formula can be reduced to SAT problem on CNF formulas.

Resolution: It is a theorem in propositional logic that if $L \vee X$ and $\neg L \vee Y$ are both true under some valuation v, then $X \vee Y$ is also true under v. Formally, we write:

$$\frac{L \vee X, \neg L \vee Y}{X \vee Y} \quad \textbf{Res}$$

where L is a literal and X, Y are both disjunction of literals. $X \vee Y$ is called the resolvent of $L \vee X$ and $\neg L \vee Y$, denoted by $X \vee Y = \mathsf{Res}(L \vee X, \neg L \vee Y)$. It is obvious that $\{L \vee X, \neg L \vee Y\}$ is satisfiable if and only if $\{L \vee X, \neg L \vee Y, X \vee Y\}$ is satisfiable. Therefore, given a set of clauses, one can resolve any pair of resolvable clauses and add the resolvent to the initial clause set. If the empty clause is found before saturation, then the initial clause set is unsatisfiable, otherwise satisfiable. This is called resolution principle [4]. Actually, some algorithms are based on the idea.

III. Searching Guided Resolution Algorithm

A. The idea of our algorithm

Resolution principle can be used to check satisfiability of a set of clauses. It is simple and easy to implement. DP [8] and DR [4] algorithms are based on it. DP is a simple improvement of resolution principle which pick the initial clause set S, do all possible resolution and collect all the resolvents S_1, then apply the same procedure on S_1 to obtain S_2. Repeat until some F_i contains empty clause "$\square$" or $F_i = \emptyset$. DP algorithm does avoid adding all resolvents to the initial set and reduced the size of clause set. However, it is not enough.

For example, for $S = \{(1) : P \vee Q, (2) : P \vee \neg Q, (3) : \neg P \vee Q, (4) : \neg P \vee \neg Q\}$. The resolution procedures are listed in Table I. In all, 7 resolutions are needed in order to derive the empty clause "$\square$".

Directional Resolution [4] is big improvement for resolution based algorithm. We already know that it is not always necessary to apply all possible resolutions. What's needed is essentially a trace to the empty clause (if there is any). So one can define an order "$>$" on $\mathcal{A}$ and resolve only upon the maximal literal. For example, in the example above, if we define $P > Q$, then empty clause is derived after 5 resolutions

(resolvent 5 and 10 will not appear). Using directional resolution will save quite a lot for unsatisfiable cases. However, it is still not satisfactory. Because for satisfiable clause set, one still has to finish all resolutions (although restricted on maximal literals, that is still a big number).

DPLL algorithm is based on search. It explores every possible valuation in order to find a model. For performance consideration, DPLL employs conflict clause analysis and clause learning. In practice, DPLL is quite efficient when the initial clause set is satisfiable. However, for unsatisfiable clauses, it also requires that all possible valuations are explored. Another disadvantage is, because each clause is not a simple resolution, it requires to record quite a lot of information as well as all generated clauses in order to find a proof for unsatisfiable result, which is not easy.

We see that the search based algorithms are good for satisfiable cases, while the resolution based algorithms are good for unsatisfiable cases. A natural idea is: if the initial clause set is satisfiable, we hope to find a model as early as possible; if unsatisfiable, resolve to empty clause as early as possible. This paper is based on the idea. Our algorithm has the following characteristics:

- based on resolution;
- all literals and clauses are ordered, to reduce the number of resolutions;
- it tries to find a model;
- if a model is not yet derived, resolution is guided by current valuation.

For both satisfiable and unsatisfiable initial clause set, our algorithm terminates quickly. The number of resolution steps is much less than other resolution based algorithms. Compared with DPLL algorithm, our algorithm is easy to implement and better at generating proofs.

B. Algorithm SGR

We call our algorithm SGR since it is a **S**earch **G**uided **R**esolution based SAT algorithm. In our algorithm, an order on atoms is needed. Use "$>$" to denote the order relation. It can be an arbitrary well-defined order relation on $\mathcal{A}$. In the rest of this paper, we assume $\mathcal{A} = \{A_1, A_2, \ldots, A_n\}$ and the order is $A_n > A_{n-1} > \cdots > A_1$. "$>$" can be extended to literals and clauses:

- For literals of different atoms, the order is decided by comparing underlying atoms.
- For literals of the same atom, the negative form is larger than the positive form. i.e.

$$\neg A_n > A_n > \neg A_{n-1} > A_{n-1} > \cdots > \neg A_1 > A_1$$

- The order on clauses is the defined as the multi-set order. In other words, empty clause is the minimal clause. For any two non-empty clauses X and Y (viewed as literal sets). Let P, Q be the maximal literals in X, Y respectively. If $P \neq Q$ then the $X > Y$ iff $P > Q$ otherwise $P = Q$, then $X > Y$ iff $X - \{P\} > Y - \{Q\}$. e.g.

$$\neg A_3 \vee A_1 > A_3 \vee A_1, \qquad A_3 \vee A_2 > A_3 \vee A_1$$

During the execution of the SGR, a valuation v is maintained. v is modified from time to time in order to form a model. σ is a map: $\mathcal{A} \mapsto ClauseSet$ which records the reason that $v(A_i)$ is modified from 0 to 1.

The procedures of SGR are described in Algorithm 1. We say a clause C is an F-Clause if $v(C) = 0$:

Algorithm 1 SGR algorithm

S0 (Initialization)
Let $S = \{C_1, C_2, \ldots, C_m\}$ be the initial clause set. S is sorted such that $C_m > C_{m-1} > \cdots > C_1$.
Let v be the initial valuation s.t. $\forall a \in \mathcal{A}. v(a) = 0$.
σ is undefined for all atoms.

S1 (Locate the minimal F-Clause under v)
Let $C_k = min\{C_i | v(C_i) = 0\}$. If such C_k does not exists, return SAT; otherwise goto S2.

S2 (Check C_k)
If the maximal literal in C_k is in the positive form then goto S3, otherwise goto S4.

S3 (Modify the valuation v)
In this case, $C_k = A_p \vee D$, where A_p is the maximal literal in C_k. We modify v and σ, let $v(A_p) = 1$, and $\sigma(A_p) = C_k$, goto S1.

S4 (Resolution)
In this case, $C_k = \neg A_p \vee D$, where $\neg A_p$ is the maximal literal in C_k. Then $\sigma(A_p)$ must be already defined and in the form of $A_p \vee E$ (i.e. A_p is the maximal literal and in the positive form).
Resolve C_k and $\sigma(A_p)$, we get $C' = D \vee E$.
If C' is empty clause then return UNSAT, otherwise, insert C' to S. For convenience, we still write $S = \{C_1, C_2, \ldots, C_m\}$.
Suppose A_q is the maximal literal in C_{new}, then for all $A_p \geq A_r > A_q$, reset $v(A_r) = 0$ and set $\sigma(A_r)$ to undefined. Goto S1.

In S4, we asserted that "$\sigma(A_p)$ must be already defined and in the form of $A_p \vee E$". We informally explain a little on this. Because $v(C_k) = 0$, $v(A_p) = 1$ must hold at that time. But by default, all atoms have value 0 under v, thus in S3 there should be some C_t such that $C_t < C_k$ and A_p is the maximal literal in C_t. Furthermore, C_t is 0 under the valuation v' when in S3. Only in this case, $v(A_p) = 1$ and $\sigma(A_p) = C_t$ are updated. Actually, $v = v' \cup \{A_p\}$ and $\sigma(A_p) = C_t = A_p \vee E$.

Take $S = \{(1) : P \vee Q, (2) : P \vee \neg Q, (3) : \neg P \vee Q, (4) : \neg P \vee \neg Q\}$ as an example. Assume the order is defined as $P > Q$. The detailed procedures are listed in Table II.

Note that in the SGR we resolve only upon the maximal literal, therefore, the number of resolution is less or equal to that in Directional Resolution. Actually, in this example, we did not resolve clauses like $P \vee Q$ and $\neg P \vee \neg Q$ which is a valid step in Directional Resolution. As a result, we only need 3 resolutions steps. In practice, SGR usually takes much less steps than Directional Resolution. In case the initial clauses are satisfiable, our algorithm is possible to find a model, then terminates without exhaustively resolved.

TABLE II
AN EXAMPLE DEMONSTRATING THE ALGORITHM

#	Related Clause	v before	v after	Explanation
1	(1): $P \vee Q$	$\{\}$	$\{P\}$	(1) is the minimal F-Clause discovered in S1. Because P is the maximal literal, we add P to v, so that $v(P \vee Q) = 1$
2	(3): $\neg P \vee Q$	$\{P\}$	$\{\}$	(3) is the minimal F-Clause discovered in S1. Because $\neg P$ is the maximal literal, which is in negative form. We resolve (1) and (3) to (a): Q, and insert it to S. At the same time, Q is the maximal literal in (a). So, in S4, we remove P ($> Q$) and v is then $\emptyset$.
3	(a): Q	$\{\}$	$\{Q\}$	(a) is the minimal F-Clause discovered in S1. Because Q is the maximal literal, we add Q to v, so that $v(Q) = 1$
4	(2): $P \vee \neg Q$	$\{Q\}$	$\{P, Q\}$	Similar to step 1.
5	(4): $\neg P \vee \neg Q$	$\{P, Q\}$	$\{Q\}$	Similar to step 2. Resolvent is $\neg Q$.
6	(b): $\neg Q$	$\{Q\}$	$\{\}$	(b) is the minimal F-Clause found in S1. Because $\neg Q$ is the maximal literal in (b), so we resolve (a) and (b). Empty clause is generated, return UNSAT.

C. Proof generation

For an satisfiable case, the proof is a model. If such model is found in the process of algorithm, the proof is already there. For an unsatisfiable case, a proof is a sequence of clauses which ends with empty clause and each clause is either an initial clause or a resolvent of two antecedents.

In S4, we can record how each clause is derived. Use τ_1 and τ_2 to record the two antecedents if there are. i.e. if C is an initial clause, $\tau_1(C)$ and $\tau_2(C)$ are both undefined, otherwise, if $C = \mathsf{Res}(D, E)$, then $\tau_1(C) = D$ and $\tau_2(C) = E$. We also say that C depends on D and E. The dependence relation actually defines a partial order on clauses. We show that, in the process of SGR algorithm, the partial order is well defined. Therefore, it is possible to do topological sort to derive a total order on these clauses. Then the proof can be simply the sequence of relative clauses in the total order. Here, relative clauses are those clauses that the empty clause depends directly or indirectly. Irrelative clauses are not to appear in the proof.

Lemma 3.1: If $C = \mathsf{Res}(D, E)$, then $D > C$ and $E > C$.

Proof: Because in step S4, we only resolve on the maximal literal, say A_p. Thus, A_p does not appear in C, and the maximal literal of C, say A_q, is less than A_p. By definition of the order on clauses, $D > C$ and $E > C$. ∎

By Lemma 3.1, actually, we can observe that the total order "$>$" is already a candidate total order for topological sort. This fact means we don't even have to do topological sort. Just use the existing "$>$" order. More importantly, if we maintain all the clauses in order (that's what we do in the implementation), what we need to do is to find all the relative clauses and output from the maximal one to the minimal one. The algorithm for finding all relative clauses is described in Algorithm 2, a stack is used. This procedure will take $O(N)$ where N is the size of proof.

The proof is stored in $proof$, in order. Take the same example we used in Sect. III-B. When the empty clause is found, all clauses are shown in Table III. Those clauses are listed in order, from minimal to maximal. The proof is [(4), (3), (2), (1), (a), (b), (c)], as in Fig. III-C:

The complexity for generating a proof is linear to the size of generated proof. i.e. at most $O(m)$, where m is the size of all clauses. Compared with other DPLL based algorithms, or algorithm is much easier to generate unsatisfiable proof.

Algorithm 2 Generate proof

```
push(□)
while stack is not empty do
  c ← pop()
  if c is not marked then
    mark c
    insert c to the beginning of proof
    if c is a resolvent then
      push(τ1(c))
      push(τ2(c))
    end if
  end if
end while
```

TABLE III
AN EXAMPLE OF GENERATING PROOF

Id	Clause	τ_1 and τ_2
(c)	$\square$	(a) and (b)
(b)	Q	(1) and (3)
(a)	$\neg Q$	(2) and (4)
(1)	$P \vee Q$	initial clause
(2)	$P \vee \neg Q$	initial clause
(3)	$\neg P \vee Q$	initial clause
(4)	$\neg P \vee \neg Q$	initial clause

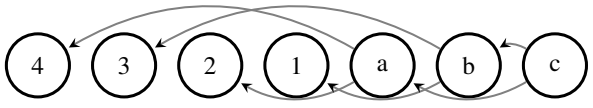

Fig. 1. Dependence relation

IV. SOUNDNESS AND COMPLETENESS

This section proves that the SGR algorithm is both sound and complete.

Lemma 4.1: If $\sigma(A)$ is defined on some atom A, then $\sigma(A)$ must have a positive literal A. i.e. $\sigma(A) = A \vee E$. Furthermore, A is the maximal literal in $\sigma(A)$.

Proof: For any given atom A, we consider the last time $\sigma(A)$ was defined. Because σ can only be defined in step S3. If $\sigma(A)$ is defined (step S3 entered), that implies A is the maximal literal of some minimal F-Clause C_k. Otherwise, it will not branch from S2 to S3.
Then, in S4, $\sigma(A)$ is set to C_k which is in the form of $A \vee E$. ■

Lemma 4.2: For each atom A, at every moment, $\sigma(A)$ is defined iff A evaluates to 1 under the valuation.

Proof: Trivial, because σ and v are always modified at the same time. And each $\sigma(A)$ is defined, $v(A)$ is set to 1; $\sigma(A)$ is undefined, $v(A)$ is set to 0. ■

Lemma 4.3: In step S4, the generated clause C' does not exist in S.

Proof: Assume the valuation is v in S4 and the minimal F-Clause founded is $\neg L \vee C$ ($\neg L$ is the maximal literal). Then $v(L) = 1$ and $v(C) = 0$ must hold. By Lemma 4.2, we know $\sigma(L)$ is defined and in the form of $L \vee D$. Assume the valuation was v' at the moment $\sigma(L)$ was defined.
$v'(L) = v'(D) = 0$ because $L \vee D$ is the minimal F-Clause when $\sigma(L)$ was defined. Furthermore, we can conclude that $v = v' \cup \{L\}$ because when v' is modified to v in S3, the minimal F-Clause is $\neg L \vee C$. In the next iteration, the minimal F-Clause must be $L \vee D$.
v and v' are only different at the atom L. The fact $v'(D) = 0$, along with $v'(C) = 0$ leads to $v'(C \vee D) = 0$. If the resolvent $C \vee D$ is identical to some clauses in S, in the algorithm, we would not pick $\neg L \vee C$ as the minimal F-Clause, because $v'(C \vee D) = 0$ and $C \vee D < \neg L \vee C$. Contradicts. ■

Theorem 4.1: For any input clause set, our algorithm must terminate eventually.

Proof: First of all, the number of different clauses defined on n atoms can not exceed 3^n. If we haven't do resolution for a certain period, then during this period, the minimal F-Clause found in step S1 should be monotonically increasing. All together, there are n atoms. So, each n times we turn to step S2, either a model is found or resolution happens. In the first case, a model is found, the algorithms already terminates. In the other case, there are at most 3^n distinct clauses, and by Lemma 4.3, each resolvent is different to all existing clauses. Thus, a upper bound of the number we turn to S2 is $n \cdot 3^n$. ■

Actually, $n \cdot 3^n$ is a very loose upper bound. However, it has already guaranteed that our algorithm terminates. Based on this result, we can give the main theorem of this section.

Theorem 4.2: Our algorithm is sound and complete.

Proof: By Theorem 4.1, our algorithm always terminates. We show that once terminates, our algorithm gives the correct answer. This will prove the soundness and completeness.
If the algorithm terminates with a SAT result, then a model is returned. In this case, no F-Clause can be found, the model is indeed a model for the initial clause set. Otherwise, empty clause is derived, by resolution principle, the initial clause is unsatisfiable. ■

V. Implementation Issue

It is intuitive to organize the clauses set in order. This will help in locating the minimal F-Clause. The efficiency of our algorithm relies on: 1) How to find the minimal F-Clause under current valuation v; 2) How to maintain the clause set where new clause can be easily inserted while preserving the order.

For the first problem, it is straight forward and intuitive way to scan from the first (minimal) clause and stop until a false clause is found. This is easy to implement but not so efficient. There are a few obvious improvements: 1) We know that the minimal F-Clause monotonically increase before step S4 is executed. Thus, we don't need to do it from the very beginning, instead, we can setup a lower bound and only search clauses that are larger than this bound until step S4 is executed. For example, if we start from C_1 and found the minimal F-Clause C_k and S4 is not executed (that means S3 executes) then next time we can start from C_{k+1}. 2) Furthermore, checking the clauses one by one is not necessary. If $v(L) = 1$ then all clauses containing a positive literal L are true under v. At least, all clauses with the maximal literal L are true under v. Thus, in that case, we can skip an interval (where the maximal literal is L) of clauses at one time.

For the second problem, we employ a data structure on which binary search can be performed in $O(log(N))$ and insertion can be in $O(1)$.

A. Find the minimal F-Clause

Observe the fact that if $\mathcal{A} = \{A_1, A_2, \ldots, A_n\}$ and $A_n > A_{n-1} > \cdots > A_1$, we can use a n-dimensional vector over $\{0, 1, 2\}$ to represent a clause: the i-th dimension of the vector is decided by the occurrence of A_i in the clause. 0 means A_i does not occur, 1 means A_i occurs and 2 means $\neg A_i$ occurs. Vector is presented by a pair of bracket. A direct observation is: For two clauses $C = (x_n, x_{n-1}, \ldots, x_1), D = (y_n, y_{n-1}, \ldots, y_1)$, the relation $C > D$ is equivalent to:

$$\exists 1 \leq k \leq n. ((\forall k < i \leq n. x_i = y_i)) \wedge (x_k > y_k))$$

The algorithm for finding the minimal F-Clause is described in Algorithm 3. The function $FindMinFClause$ takes 3 parameters. $start$ (including) and end (excluding) defines the interval to search, i.e. finding the minimal F-Clause in the range of $[start, end)$. This procedure is done recursively, so we need a parameter $depth$ to record which level we are at. It is guaranteed that, each step, clauses in $[start, end)$ are identical from the n-th to $(depth + 1)$-th dimension.

A function $Partition$ is referred. It requires a precondition that clauses in $[start, end)$ are identical from the n-th to the $(depth + 1)$-th dimension. We are lucky to have this condition satisfied in $FindMinFClause$. $Partition$ will return 2 bounds, p_1, p_2, the $depth$-th dimension is 0 for $[start, p_1)$, 1 for $[p_1, p_2)$ and 2 for $[p_2, end)$. In the implementation, p_1, p_2 are decided by binary search.

B. A data structure for storing the clauses

The data structure we use to store all clauses should satisfy our requirement:

Algorithm 3 Find the minimal F-Clause

Function $FindMinFClause$(Clause $start$, Clause end, int $depth$)
if $start = end$ **then**
  return $null$;
else if $start < 0$ **then**
  return $start$;
else
  $(p_1, p_2) = Partition(start, end, depth)$
  $first \leftarrow FindMinFClause(start, p_1, depth - 1)$
  if $first \neq null$ **then**
    return $first$
  else
    if $v(A_{depth}) = 1$ **then**
      return $FindMinFClause(p_2, end, depth - 1)$
    else
      return $FindMinFClause(p_1, p_2, depth - 1)$
    end if
  end if
end if

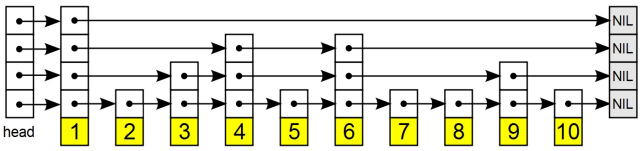

Fig. 2. An example of SkipList

- Low complexity for locating a specified item;
- Low complexity for inserting a new item at any position;

It's theoretically applicable to use balanced tree, which offers $O(log(N))$ locating and $O(log(N))$ insertion (maintain the tree balanced) while using $O(N)$ space. However, to implement such data structure is complicated, the inherent complexity may be quite big. Here, we use a simple and efficient stat structure, SkipList [12]. The complexity for locating and insertion are both $O(log(m))$ but easy to implement. Links in the SkipList are of different levels. When locating an element, we search from the higher level to the lower level. Fig. 2 is a demonstration of logical structure of SkipList from Wikipedia. Detailed description can be found in the cited literature.

With the above optimizations, the performance is acceptable. For further improvement, other tricks are needed. Actually, almost every mature solver employs several optimizations. While writing this paper, we can not dive into the very detail. The proposed algorithm is further improvable by applying other techniques, such as defining a hash function which preserving the clause order to speed up clause comparison, calculating a maximal acceptable level while using SkipList, and so on.

VI. Complexity analysis and experiment results

First of all, the execution time of our algorithm is highly related to the size of the clause set. Each resolution increases the clause set by 1, thus the complexity highly depends on how many resolutions we applied. We mentioned in 4.1 that there is an upper bound for the number of resolutions which is exponential to the size of atoms. However, an exponential complexity is usually (not theoretically proved) a lower bound for NP-Complete problems.

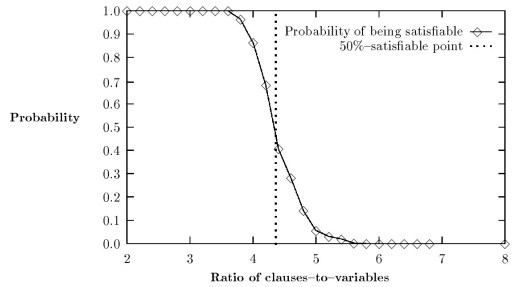

Fig. 3. The probability of being satisfiable

Compared with other resolution based algorithm, the following property hold: *The number of resolutions in our algorithm does not exceed the number of resolutions in directional resolution*. Because each resolution is applied on the maximal literal. Actually, experiment results show that our algorithm takes much less resolutions than directional resolution.

In order to test the performance of our algorithm, we need a serial of test cases. Two categories of test cases are usually used as benchmarks: one is random generated cases, while the other is structured cases.

A. Random generated test cases

For random generated cases, it is almost canonical to test on 3-SAT problem where the difficulty of a problem can be reflected by the number of clauses m and the number of atoms n. Some research shows that the ratio of m/n statistically determines the probability that a random generated formula being satisfiable. When $m/n > 4.3$, the probability is above 50%, while for $m/n < 4.3$, the probability is below 50%. As in Fig. 3 [13], the horizontal axis is the ratio of m/n, the vertical axis is the statistical data that a random generated formula is satisfiable. For a certain n, statistics shows that almost every existing sat solver spends most time solving a random generated formula when $m/n = 4.3$, so this kind of problem is usually considered as the most difficult class of 3-SAT problem.

We are going to discuss these 3 cases respectively. Because directional resolution is better than many other resolution algorithms, we compare with directional resolution.

1) For $m/n < 4.3$: Statistically, formulas are likely to be satisfiable. Our algorithm can find a model very quickly. In directional resolution, resolvent may already exist. But in our algorithm, resolvent must be a fresh clause. So, we distinguish "Added"(the number of added clauses) and "Resolution"(the number of resolutions). In the table, "Modify v" is how often we modified the valuation v is S3, "Added" is how often we resolve in S4.

TABLE IV
EXPERIMENT RESULTS OF $m/n < 4.3$ CASES

Test Case				Directional Resolution			Algorithm in this paper		
n	m	m/n	Status	Time	Added	Resolutions	Time	Added	Modify v
10	30	3.0	SAT	0.01s	300	3860	0.00s	1	4
20	60	3.0	SAT	159.4s	80545	237106115	0.00s	18	44
40	120	3.0	SAT	–	–	–	0.00s	17	73
80	240	3.0	SAT	–	–	–	0.00s	68	326
100	300	3.0	SAT	–	–	–	0.00s	300	1445
200	600	3.0	SAT	–	–	–	0.00s	325	2207
300	900	3.0	SAT	–	–	–	0.03s	1709	10381
100	320	3.2	SAT	–	–	–	0.00s	319	1358
100	340	3.4	SAT	–	–	–	0.01s	912	2920
100	360	3.6	SAT	–	–	–	0.14s	5176	5536
100	380	3.8	SAT	–	–	–	0.19s	6325	11901
100	400	4.0	SAT	–	–	–	1.30s	14031	27005

In Table IV, "–" means the program has run for more than 10 minutes without returning a result. We can see from the table that our algorithm actually found a model quickly. We don't need to complete all resolutions.

2) For $m/n > 4.3$: Compared with previous result, unsatisfiable formulas need more time to verify because we have to do resolution until saturation, shown in Table V.

3) For $m/n = 4.3$: In this case, there is half chance for a random formula to be satisfiable (or unsatisfiable). It is considered that this kind of problem is the most difficult class to verify. They are even difficult to verify than many non 4.3 problems which are in very large scale. There is an international benchmark package, "SAT-LIB", containing many random 4.3 ratio 3-SAT problems. Our test cases are picked from this package.

Compared with resolution based algorithm, our algorithm is much better than directional resolution both in the added clauses and execution time, as shown in Table VI.

B. Structured cases

For structured cases, different algorithm has different performance due to the characteristics of problem. We researched 2 kinds of structured problems which are also used as benchmarks in [10], the *chains* problem and *(k,m)-tree* problem. They each has a specialized structure which consists of several *cliques*. Each clique is an set of clauses that involving several certain variables. There are very few clauses that involving variables interleaving different cliques.

The chains problem test cases are generated as follows: Firstly, we create a sequence of independent uniform k-cnf cliques and connects each pair of successive cliques by a 2-cnf clause containing variables from two consecutive cliques. The parameters of generator are the number of cliques, N_{clique}, the number of variables perclique, N, and the number of clauses per clique, C.

The (k,m)-tree problem test cases are generated as follows: A tree of cliques each having $(k+m)$ nodes where k is the size of the intersection between two neighboring cliques. Given k, m, the number of cliques N_{clique}, and the number of clauses perclique, N_{cls}, the generator produces a clique of $k+m$ size with N_{cls} clauses and then generate ech of the other $N_{clique}-1$ cliques by selecting randomly an existing clique and its k variables, adding m new variables, and generating N_{cls} clauses on that new clique.

TABLE VII
EXPERIMENT RESULTS ON *chains* CASES

Test Case			Algorithm in this paper			
n	m	Status	Time	Added	Modify v	Proof Size
125	299	SAT	0.02s	42	81	–
125	399	UNSAT	0.01s	13	15	25
125	499	UNSAT	0.00s	16	22	22
300	1299	UNSAT	0.1s	149	223	54
300	1524	SAT	0.1s	6	33	–
300	1599	SAT	0.4s	209	350	–
400	1419	SAT	1.2s	723	1240	–
400	1619	SAT	0.15s	27	88	–
500	1774	UNSAT	1.9s	645	1156	162
750	5024	SAT	2.1s	173	419	–
1000	6274	UNSAT	5.1s	1117	1971	1089

TABLE VIII
EXPERIMENT RESULTS ON *(k,m)-tree* CASES

Test Case			Algorithm in this paper			
n	m	Status	Time	Added	Modify v	Proof Size
300	1563	SAT	0.1s	19	39	–
300	1607	SAT	0.1s	181	310	–
300	1260	UNSAT	0.1s	123	180	46
400	1422	SAT	0.2s	690	1122	–
400	1646	SAT	0.2s	33	79	–
500	1605	SAT	0.8s	620	1232	–
500	1808	UNSAT	1.7s	797	1923	302
1000	6221	UNSAT	3.1s	1090	1902	980
1000	7002	UNSAT	7.2s	3987	8293	2841

We implemented both generators and run tests on the generated data. Results are listed in Table VII and Table VIII respectively. We can see our algorithm works well and the number of resolution is small. On these structured problems, our algorithm out performs those algorithms [1], which also combines search and resolution, described in [10].

[1]The conclusion comes from comparing the experiment data of ours and that from the referred paper. We have neither exactly the same test cases nor their executable solvers. However, we believe the test case generator employs the same algorithm as they did.

TABLE V
EXPERIMENT RESULTS OF $m/n > 4.3$ CASES

Test Case				Directional Resolution			Algorithm in this paper		
n	m	m/n	Status	Time	Added	Resolutions	Time	Added	Modify v
10	48	4.8	UNSAT	0.01s	540	14978	0.00s	33	38
20	96	4.8	UNSAT	174.12s	84562	296650036	0.00s	137	206
40	192	4.8	UNSAT	–	–	–	0.00s	1733	3032
60	288	4.8	UNSAT	–	–	–	0.34s	7777	13081
80	384	4.8	UNSAT	–	–	–	12.77s	34051	52444
100	480	4.8	UNSAT	–	–	–	580.32s	199025	256943

TABLE VI
EXPERIMENT RESULTS OF $m/n = 4.3$ CASES

Test Case				Directional Resolution			Algorithm in this paper		
n	m	m/n	Status	Time	Added	Resolutions	Time	Added	Modify v
10	43	4.3	SAT	0.00s	101	602	0.00s	3	6
20	86	4.3	UNSAT	730.26s	147881	1077149856	0.00s	161	238
30	129	4.3	UNSAT	–	–	–	0.00s	346	584
40	172	4.3	SAT	–	–	–	0.00s	455	978
50	215	4.3	SAT	–	–	–	0.07s	3446	5633
60	258	4.3	UNSAT	–	–	–	1.01s	13130	19564
70	301	4.3	UNSAT	–	–	–	10.68s	33062	51211
80	344	4.3	UNSAT	–	–	–	30.06s	48365	76503
90	387	4.3	SAT	–	–	–	57.92s	67977	96984
100	430	4.3	UNSAT	–	–	–	378.3s	154026	213773

C. Proof generation

The advantage of our algorithm is that it is easy to implement and can generate proof. Although SAT-solvers such as zChaff, PicoSAT and MiniSAT also provide proofs for unsatisfiable cases, the formats are quite different from each other. So it seems not reasonable to compare the proof size. However, our proof format is quite simple, which is given in a sequence of clauses and its two resolution antecedents. Furthermore, from Table VII and VIII, one can see intuitively that the proof size is much smaller than the scale of problem input.

The idea of our algorithm is adoptable, based on this idea, we combine both search and resolution. Every algorithm is not born efficient, our SGR algorithm provides a new way of thinking about resolutions in solving the SAT problem. We will work on this because we believe that all algorithms can be further improved.

VII. CONCLUSION AND FUTURE WORK

This paper investigated a search guided resolution based algorithm for solving SAT problem. We explained in details the idea and procedure of algorithm, proved the soundness and correctness. Then we compared our algorithm with directional resolution which is another resolution based algorithm. Our algorithm is more scalable. We believe that all algorithms are not born efficient. It is hopeful that in the future we can construct a new and more efficient algorithm based on ideas in this paper.

ACKNOWLEDGMENT

Min Zhou, Fei He and Ming Gu are with the Tsinghua National Laboratory for Information Science and Technology (TNList), and School of Software, Tsinghua University, and the Key Laboratory for Information System Security, Ministry of Education, Beijing, China. This research is sponsored by the NSFC Program (No.91018015, 60811130468, 60903030) and 973 Program (No.2010CB328003) of China.

REFERENCES

[1] S. Cook, "The complexity of theorem-proving procedures," in *Proceedings of the third annual ACM symposium on Theory of computing*. ACM, 1971, pp. 151–158.

[2] C. Barrett, D. Dill, and A. Stump, "Checking satisfiability of first-order formulas by incremental translation to SAT," in *Computer Aided Verification*. Springer, 2002, pp. 681–710.

[3] J. Zhang and X. Wang, "A constraint solver and its application to path feasibility analysis," *International Journal of Software Engineering and Knowledge Engineering*, vol. 11, no. 2, pp. 139–156, 2001.

[4] J. Robinson and A. Voronkov, *Handbook of automated reasoning*. North Holland, 2001.

[5] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Communications of the ACM*, vol. 5, no. 7, pp. 394–397, 1962.

[6] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Design Automation Conference, 2001. Proceedings*. IEEE, 2001, pp. 530–535.

[7] N. Sörensson and N. Een, "Minisat v1. 13-a sat solver with conflict-clause minimization," *SAT*, vol. 2005, p. 53, 2005.

[8] M. Davis and H. Putnam, "A computing procedure for quantification theory," *Journal of the ACM (JACM)*, vol. 7, no. 3, pp. 201–215, 1960.

[9] R. Dechter and I. Rish, *Directional resolution: The davis-putnam procedure, revisited*. Citeseer, 1994.

[10] I. Rish and R. Dechter, "Resolution versus search: Two strategies for SAT," *Journal of Automated Reasoning*, vol. 24, no. 1, pp. 225–275, 2000.

[11] P. Beame, H. Kautz, and A. Sabharwal, "Understanding the power of clause learning," in *International Joint Conference on Artificial Intelligence*, vol. 18. Citeseer, 2003, pp. 1194–1201.

[12] W. Pugh, "Skip lists: a probabilistic alternative to balanced trees," *Communications of the ACM*, vol. 33, no. 6, pp. 668–676, 1990.

[13] D. Mitchell, B. Selman, and H. Levesque, "Hard and easy distributions of SAT problems," in *Proceedings of the National Conference on Artificial Intelligence*. Citeseer, 1992, pp. 459–459.