# The 2-MAXSAT Problem Can Be Solved in Polynomial Time

Yangjun Chen*

*Abstract*—By the MAXSAT problem, we are given a set $V$ of $m$ variables and a collection $C$ of $n$ clauses over $V$. We will seek a truth assignment to maximize the number of satisfied clauses. This problem is *NP*-hard even for its restricted version, the 2-MAXSAT problem by which every clause contains at most 2 literals. In this paper, we discuss an efficient algorithm to solve this problem. Its worst-case time complexity is bounded by $O(n^3 m^3)$. This shows that the 2-MAXSAT problem can be solved in polynomial time.

*Index Terms*—satisfiability problem, maximum satisfiability problem, NP-hard, NP-complete, conjunctive normal form, disjunctive normal form.

## I. INTRODUCTION

THE satisfiability problem is perhaps one of the most well-studied problems that arise in many areas of discrete optimization, such as artificial intelligence, mathematical logic, and combinatorial optimization, to just name a few. Given a set $V$ of Boolean (*true/false*) variables and a collection $C$ of clauses over $V$, or say, a logic formula in *CNF* (Conjunctive Normal Form), the satisfiability problem is to determine if there is a truth assignment that satisfies all clauses in $C$ [1]. The problem is *NP*-complete even when every clause in $C$ has at most three literals [3]. The maximum satisfiability (MAXSAT) problem is an optimization version of satisfiabiltiy that seeks a truth assignment to maximize the number of satisfied clauses [5]. This problem is *NP*-hard even for its restricted version, the so-called 2-MAXSAT problem, by which every clause in $C$ has at most two literals [4]. Its application can be seen in an extensive biliography [2], [4], [7], [10]–[13], [16].

Over the past several decades, a lot of research on the MAXSAT has been conducted. Almost all of them are the approximation methods [6], [14], [15], such as (1-1/$e$)-approximation, 3/4-approximation [15], as well as the method based on the integer linear programming.

In this paper, we discuss a polynomial time algorithm to solve the 2-MAXSAT problem. Its worst-case time complexity is bounded by $O(n^3 m^3)$, where $n$ and $m$ are the numbers of clauses and the number of variables in $C$, respectively. Thus, our algorithm is in fact a proof of $P = NP$.

The main idea behind our algorithm can be summarized as follows.

*Y. Chen is with the Department of Applied Computer Science, University of Winnpeg, Manitoba, Canada, R3B 2E9.
E-mail: see http://www.uwinnipeg.ca/~ychen2

1) Given a collection $C$ of $n$ clauses over a set of variables $V$ with each containing at most 2 literals. Construct a formula $D$ over another set of variables $U$, but in *DNF* (Disjunctive Normal Form) containg $2n$ conjunctions with each of them having at most 2 literals such that there is a truth assignment for $V$ that satisfies at least $n* \leq n$ clauses in $C$ if and only if there is a truth assignment for $U$ that satisfies at least $n*$ conjunctions in $D$.

2) For each $D_i$ in $D$ ($i \in \{1, ..., 2n\}$), construct a graph, called a $p*$-graph to represent all those truth assignments $\sigma$ of variables such that under $\sigma$ $D_i$ evaluates to *true*.

3) Organize the $p*$-graphs for all $D_i$'s into a trie-like graph $G$. Searching $G$ bottom up, we can find a maximum subset of satisfied conjunctions in polynomial time.

The organization of the rest of this paper is as follow. First, in Section II, we restate the definition of the 2-MAXSAT problem and show how to reduce it to a problem that seeks a truth assignment to maximize the number of satisfied conjunctions in a formula in *DNF*. Then, we discuss our algorithm in Section III. Next, in Section IV, we discuss how the basic algorithm can be improved. Section V is devoted to the analysis of the time complexity of the algorithm. Finally, a short conclusion is set forth in Section VI.

## II. 2-MAXSAT PROBLEM

We will deal solely with Boolean variables (that is, those which are either *true* or *false*), which we will denote by $c_1$, $c_2$, etc. A literal is defined as either a variable or the negation of a variable (e.g., $c_7$, $\neg c_{11}$ are literals). A literal $\neg c_i$ is *true* if the variable $c_i$ is *false*. A clause is defined as the OR of some literals, written as $(l_1 \vee l_2 \vee .... \vee l_k)$ for some $k$, where each $l_i$ ($1 \leq i \leq k$) is a literal, as illustrated in $\neg c_1 \vee c_{11}$. We say that a Boolean formula is in conjunctive normal form (*CNF*) if it is presented as an AND of clauses: $C_1 \wedge ... \wedge C_n$ ($n \geq 1$). For example, $(\neg c_1 \vee c_7 \vee \neg c_{11}) \wedge (c_5 \vee \neg c_2 \vee \neg c_3)$ is in *CNF*. In addition, a disjunctive normal form (*DNF*) is an OR of conjunctions: $D_1 \vee D_2 \vee ... \vee D_m$ ($m \geq 1$). For instance, $(c_1 \wedge c_2) \vee (\neg c_1 \wedge c_{11})$ is in *DNF*.

Finally, the MAXSAT problem is to find an assignment to the variables of a Boolean formula in *CNF* such that the maximum number of clauses are set to *true*, or are satisfied. Formally:

2-MAXSAT

- Instance: A finite set $V$ of variables, a Boolean formula $C = C_1 \wedge ... \wedge C_n$ in *CNF* over $V$ such that each $C_i$ has

$0 < |C_i| \leq 2$ literals ($i = 1, ..., n$), and a positive integer $n^* \leq n$.

- Question: Is there a truth assignment for $V$ that satisfies at least $n^*$ clauses?

In terms of [4], 2-MAXSAT is *NP*-complete.

To find a truth assignment $\sigma$ such that the number of clauses set to *true* is maximized under $\sigma$, we can try all the possible assignments, and count the satisfied clauses as discussed in [12]. We may also use a heuristic method to find an approximate solution to the problem as described in [5].

In this paper, we propose a quite different method, by which for $C = C_1 \wedge ... \wedge C_n$, we will consider another formula $D$ in *DNF* constructed as follows.

Let $C_i = c_{i1} \vee c_{i2}$ be a clause in $C$, where $c_{i1}$ and $c_{i2}$ denote either variables in $V$ or their negations. For $C_i$, define a variable $x_i$. and a pair of conjunctions: $D_{i1}$, $D_{i2}$, where

$$D_{i1} = c_{i1} \wedge x_i,$$
$$D_{i2} = c_{i2} \wedge \neg x_i.$$

Let $D = D_{11} \vee D_{12} \vee D_{21} \vee D_{22} \vee ... \vee D_{n1} \vee D_{n2}$. Then, given an instance of the 2-MAXSAT problem defined over a variable set $V$ and a collection $C$ of $n$ clauses, we can construct a logic formula $D$ in *DNF* over the set $V \cup X$ in polynomial time, where $X = \{x_i | i = 1, ..., n\}$. $D$ has $m = 2n$ conjunctions.

Concerning the relationship of $C$ and $D$, we have the following proposition.

**Proposition 1.** *Let $C$ and $D$ be a formula in CNF and a formula in DNF defined above, respectively. No less than $n^*$ clauses in $C$ can be satisfied by a truth assignment for $V$ if and only if no less than $n^*$ conjunctions in $D$ can be satisfied by some truth assignment for $V \cup X$.*

*Proof.* Consider every pair of conjunctions in $D$: $D_{i1} = c_{i1} \wedge x_i$ and $D_{i2} = c_{i2} \wedge \neg x_i$ ($i \in \{1, ..., n\}$). Clearly, under any truth assignment for the variables in $V \cup X$, at most one of $D_{i1}$ and $D_{i2}$ can be satisfied. If $x_i = true$, we have $D_{i1} = c_{i1}$ and $D_{i2} = false$. If $x_i = false$, we have $D_{i2} = c_{i2}$ and $D_{i1} = false$.

"$\Rightarrow$" Suppose there exists a truth assignment $\sigma$ for $C$ that satisfies $p \geq n^*$ clauses in $C$. Without loss of generality, assume that the $p$ clauses are $C_1, C_2, ..., C_p$.

Then, similar to Theorem 1 of [7], we can find a truth assignment $\tilde{\sigma}$ for $D$, satisfying the following condition:

For each $C_j = c_{i1} \vee c_{i2}$ ($j = 1, ..., p$), if $c_{j1}$ is *true* and $c_{j2}$ is *false* under $\sigma$, (1) set both $c_{j1}$ and $x_j$ to *true* for $\tilde{\sigma}$. If $c_{j1}$ is *false* and $c_{j2}$ is *true* under $\sigma$, (2) set $c_{j2}$ to *true*, but $x_j$ to *false* for $\tilde{\sigma}$. If both $c_{i1}$ and $c_{i2}$ are *true*, do (1) or (2) arbitrarily.

Obviously, we have at least $n^*$ conjunctions in $D$ satisfied under $\tilde{\sigma}$.

"$\Leftarrow$" We now suppose that a truth assignment $\tilde{\sigma}$ for $D$ with $q \geq n^*$ conjunctions in $D$ satisfied. Again, assume that those $q$ conjunctions are $D_{1b_1}, D_{2b_2}, ..., D_{qb_q}$, where each $b_j$ ($j = 1, ..., q$) is 1 or 2.

Then, we can find a truth assignment $\sigma$ for $C$, satisfying the following condition:

For each $D_{jb_j}$ ($j = 1, ..., q$), if $b_j = 1$, set $c_{j1}$ to *true* for $\sigma$; if $b_j = 2$, set $c_{j2}$ to *true* for $\sigma$.

Clearly, under $\sigma$, we have at lease $n^*$ clauses in $C$ satisfied. The above discussion shows that the proposition holds. $\square$

As an example, consider the following logic formula in *CNF*:

$$\begin{aligned} C &= C_1 \wedge C_2 \wedge C_3 \\ &= (c_1 \vee c_2) \wedge (c_2 \vee \neg c_3) \wedge (c_3 \vee \neg c_1) \end{aligned} \quad (1)$$

Under the truth assignment $\sigma = \{c_1 = 1, c_2 = 1, c_3 = 1\}$, $C$ evaluates to *true*, i.e., $C_i = 1$ for $i = 1, 2, 3$. Thus, $n^* = 3$.

For $C$, we will generate another formula $D$, but in *DNF*, according to the above discussion:

$$\begin{aligned} D &= D_{11} \vee D_{12} \vee D_{21} \vee D_{22} \vee D_{31} \vee D_{32} \\ &= (c_1 \wedge c_4) \vee (c_2 \wedge \neg c_4) \vee \\ &\quad (c_2 \wedge c_5) \vee (\neg c_3 \wedge \neg c_5) \vee \\ &\quad (c_3 \wedge c_6) \vee (\neg c_1 \wedge \neg c_6). \end{aligned} \quad (2)$$

According to Proposition 1, $D$ should also have at least $n^* = 3$ conjunctions which evaluates to *true* under some truth assignment. In the opposite, if $D$ has at least 3 satisfied conjunctions under a truth assignment, then $C$ should have at least three clauses satisfied by some truth assignment, too. In fact, it can be seen that under the truth assignment $\tilde{\sigma} = \{c_1 = 1, c_2 = 1, c_3 = 1, c_4 = 1, c_5 = 1, c_6 = 1\}$, $D$ has three satisfied conjunctions: $D_{11}$, $D_{21}$, and $D_{31}$, from which the three satisfied clauses in $C$ can be immediately determined.

In the following, we will discuss a polynomial time algorithm to find a maximum set of satisfied conjunctions in any logic formular in *DNF*, not only restricted to the case that each conjunction contains up to 2 conjuncts.

## III. ALGORITHM DESCRIPTION

In this section, we discuss our algorithm. First, we present the main idea in Section III-A. Then, in Section III-B, the basic algorithm for solving the problem will be described in great detail. The further improvement of the basic algorithm will be discussed in the next section.

### A. Main idea

To develop an efficient algorithm to find a truth assignment that maximizes the number of satisfied conjunctions in formula $D = D_1 \vee ... \vee D_n$, where each $D_i$ ($i = 1, ..., n$) is a conjunction, we need to represent each $D_i$ as a variable sequence. For this purpose, we introduce a new notation:

$$(c_j, *) = c_j \vee \neg c_j = true,$$

which will be inserted into $D_i$ to represent any missing variable $c_j$ in $D_i$. Obviously, the truth value of each $D_i$ remains unchanged.

In this way, the above $D$ can be rewritten as a new formula in *DNF* as follows:

$$D = D_1 \vee D_2 \vee D_3 \vee D_4 \vee D_5 \vee D_6$$
$$= (c_1 \wedge (c_2, *) \wedge (c_3, *) \wedge c_4 \wedge (c_5, *) \wedge (c_6, *) \vee$$
$$((c_1, *) \wedge c_2 \wedge (c_3, *) \wedge \neg c_4 \wedge (c_5, *) \wedge (c_6, *)) \vee$$
$$((c_1, *) \wedge c_2 \wedge (c_3, *) \wedge (c_4, *) \wedge c_5 \wedge (c_6, *)) \vee$$
$$((c_1, *) \wedge (c_2, *) \wedge \neg c_3 \wedge (c_4, *) \wedge \neg c_5 \wedge (c_6, *)) \vee$$
$$((c_1, *) \wedge (c_2, *) \wedge c_3 \wedge (c_4, *) \wedge (c_5, *) \wedge c_6) \vee$$
$$(\neg c_1 \wedge (c_2, *) \wedge (c_3, *) \wedge (c_4, *) \wedge (c_5, *) \wedge \neg c_6)$$

$$\text{(3)}$$

Doing this enables us to represent each $D_i$ as a variable sequence, but with all the negative literals being removed. It is because if the variable in a negative literal is set to *true*, the corresponding conjunction must be *false*. See Table I for illustration.

First, we pay attention to the variable sequence for $D_2$ (the second sequence in the second column of Table I), in which the negative literal $\neg c_4$ (in $D_2$) is eliminated. In the same way, you can check all the other variable sequences.

Now it is easy for us to compute the appearance frequencies of different variables in the variable sequences, by which each $(c, *)$ is counted as a single appearance of $c$ while any negative literals are not considered, as illustrated in Table II, in which we show the appearance frequencies of all the variables in the above $D$.

According to the variable appearance frequencies, we will impose a global ordering over all variables in $D$ such that the most frequent variables appear first, but with ties broken arbitrarily. For instance, for the $D$ shown above, we can specify a global ordering like this: $c_2 \to c_3 \to c_1 \to c_4 \to c_5 \to c_6$.

Following this general ordering, each conjunction $D_i$ in $D$ can be represented as a sorted variable sequense as illustrated in the third column of Table I, where a start symbol # and an end symbol $ are used for technical convenience. In fact, any ordering of variables works well, based on which a graph representation of assignments can be established. However, ordering variables according to their appearance frequencies can greatly improve the efficiency when searching the trie (to be defined in the next subsection) constructed over all the variable sequences for conjunctions in $D$.

Later on, by a variable sequence, we always mean a sorted variable sequence. Also, we will use $D_i$ and the variable sequence for $D_i$ interchangeably without causing any confusion.

In addition, for our algorithm, we need to introduce a graph structure to represent all those truth assignments for each $D_i$ ($i = 1, ..., n$) (called a $p^*$-graph), under which $D_i$ evaluates to *true*. In the following, however, we first define a simple concept of $p$-graphs for ease of explanation.

**Definition 1.** ($p$-graph) Let $\alpha = c_0 c_1 ... c_k c_{k+1}$ be an variable sequence representing a $D_i$ in $D$ as described above (with $c_0$ = # and $c_{k+1}$ = $). A $p$-graph over $\alpha$ is a directed graph, in which there is a node for each $c_j$ ($j = 0, ..., k + 1$); and an edge for $(c_j, c_{j+1})$ for each $j \in \{0, 1, ..., k\}$. In addition, there may be an edge from $c_j$ to $c_{j+2}$ for each $j \in \{0, ..., k - 1\}$ if $c_{j+1}$ is a pair of the form $(c, *)$, where c is a variable name.

In Fig. 1(a), we show such a $p$-graph for $D_1 = \#.(c_2, *).(c_3, *).c_1.c_4.(c_5, *).(c_6, *).\$$. Beside a main path going through all the variables in $D_1$, there are four off-path edges (edges not on the main path), referred to as *spans*, corresponding to $(c_2, *)$, $(c_3, *)$, $(c_5, *)$, and $(c_6, *)$, respectively. Each span is represented by the sub-path covered by it. For example, we will use the sub-path $<v_0, v_1, v_2>$ (sub-path going three nodes: $v_0, v_1, v_2$) to stand for the span connecting $v_0$ and $v_2$; $<v_1, v_2, v_3>$ for the span connecting $v_2$ and $v_3$; $<v_4, v_5, v_6>$ for the span connecting $v_4$ and $v_6$, and $<v_5, v_6, v_7>$ for the span connecting $v_6$ and $v_7$. By using spans, the meaning of '*'s (it is either 0 or 1) is appropriately represented since along a span we can bypass the corresponding variable (then its value is set to 0) while along an edge on the main path we go through the corresponging variable (then its value is set to 1).
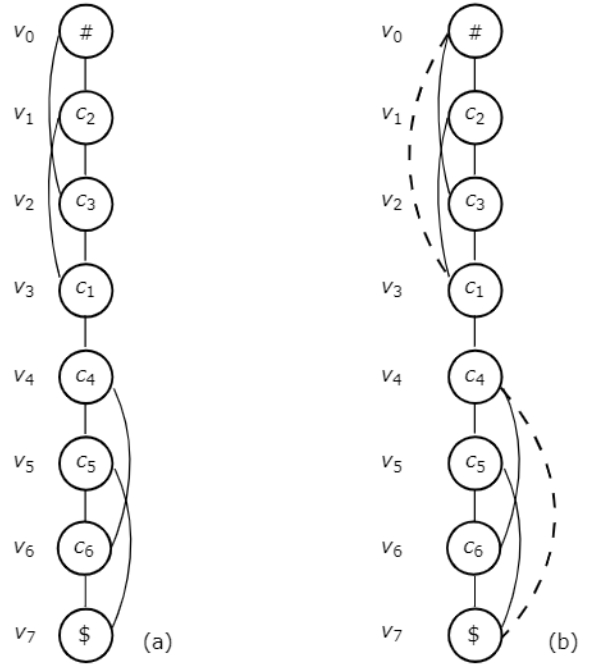


FIG. 1: A $p$-path and a $p*$-path.

In fact, what we want is to represent all those truth assignments for each $D_i$ ($i = 1, ..., n$) in an efficient way, under which $D_i$ evaluates to *true*. However, $p$-graphs fail to do so since when we go through from a node $v$ to another node $u$ through a span, $u$ must be selected. If $u$ represents a $(c, *)$ for some variable name $c$, the meaning of this '*' is not properly rendered.

For this reason, we introduced the concept of $p^*$-graphs, described as below.

Let $s_1 = <v_1, ..., v_k>$ and $s_2 = <u_1, ..., u_l>$ be two spans attached on a same path. We say, $s_1$ and $s_2$ are overlapped, if $u_1 = v_j$ for some $j \in \{1, ..., k - 1\}$, or if $v_1 = u_{j'}$ for some $j' \in \{1, ..., l - 1\}$. For example, in Fig. 1(a), $<v_0, v_1, v_2>$ and $<v_1, v_2, v_3>$ are overlapped. $<v_4, v_5, v_6>$ and $<v_5, v_6, v_7>$ are also overlapped.

TABLE I: Conjunctions represented as sorted variable sequences.

| conjunction | variable sequences | sorted variable sequences |
|---|---|---|
| $D_1$ | $c_1.(c_2, *).(c_3, *).c_4.(c_5, *).(c_6, *)$ | $\#.(c_2, *).(c_3, *).c_1.c_4.(c_5, *).(c_6, *).\$$ |
| $D_2$ | $(c_1, *).c_2.(c_3, *).(c_5, *).(c_6, *)$ | $\#.c_2.(c_3, *).(c_1, *).(c_5, *).(c_6, *).\$$ |
| $D_3$ | $(c_1, *).c_2.(c_3, *).(c_4, *).c_5.(c_6, *)$ | $\#.c_2.(c_3, *).(c_1, *).(c_4, *).c_5.(c_6, *).\$$ |
| $D_4$ | $(c_1, *).(c_2, *).(c_4, *).(c_6, *)$ | $\#.(c_2, *).(c_1, *).(c_4, *).(c_6, *).\$$ |
| $D_5$ | $(c_1, *).(c_2, *).c_3.(c_4, *).(c_5, *).c_6$ | $\#.(c_2, *).c_3.(c_1, *).(c_4, *).(c_5, *).c_6.\$$ |
| $D_6$ | $(c_2, *).(c_3, *).(c_4, *).(c_5, *)$ | $\#.(c_2, *).(c_3, *).(c_4, *).(c_5, *).\$$ |

TABLE II: Appearance frequences of variables.

| variables | $c_1$ | $c_2$ | $c_3$ | $c_4$ | $c_5$ | $c_6$ |
|---|---|---|---|---|---|---|
| appearance frequencies | 5/6 | 6/6 | 5/6 | 5/6 | 5/6 | 5/6 |

Here, we notice that if we had one more span, $<v_3, v_4, v_5>$, for example, it would be connected to $<v_1, v_2, v_3>$, but not overlapped with $<v_1, v_2, v_3>$. Being aware of this difference is important since the overlapped spans imply the consecutive '*'s, just like $<v_1, v_1, v_2>$ and $<v_1, v_2, v_3>$, which correspond to two consecutive '*'s: $(c_2, *)$ and $(c_3, *)$. Therefore, the overlapped spans exhibit some kind of *transitivity*. That is, if $s_1$ and $s_2$ are two overlapped spans, the $s_1 \cup s_2$ must be a new, but bigger span. Applying this operation to all the spans over a $p$-path, we will get a 'transitive closure' of overlapped spans. Based on this observation, we give the following definition.

**Definition 2.** ($p$*-graph) Let $P$ be a $p$-graph. Let $p$ be its main path and $S$ be the set of all spans over $p$. Denote by $S$* the 'transitive closure' of $S$. Then, the $p$*-graph with respect to $P$ is the union of $p$ and $S$*, denoted as $P$* $= p \cup S$*.

In Fig. 1(b), we show the $p$*-graph with respect to the $p$-graph shown in Fig. 1(a). Concerning $p$*-graphs, we have the following lemma.

**Lemma 1.** Let $P$* be a $p$*-graph for a conjunction $D_i$ (represented as a variable sequence) in $D$. Then, each path from # to $ in $P$* represents a truth assignment, under which $D_i$ evaluate to true.

*Proof.* (1) Corresponding to any truth assignment $\sigma$, under which $D_i$ evaluates to $true$, there is definitely a path from # to $ in $p$*-path. First, we note that under such a truth assignment each variable in a positive literal must be set to 1, but with some '*'s set to 1 or 0. Especially, we may have more than one consecutive '*'s that are set 0, which are represented by a span that is the union of the corresponding overlapped spans. Therefore, for $\sigma$ we must have a path representing it.

(2) Each path from # to $ represents a truth assignment, under which $D_i$ evaluate to *true*. To see this, we observe that each path consists of several edges on the main path and several spans. Especially, any such path must go through every verable in a positive literal since for each of them there is no span covering it. But each span stands for a '*' or more than one successive '*'s. □

*B. Algorithm*

To find a truth assignment to maximize the number of satisfied $D'_j$s in $D$, we will first construct a *trie-like* structure

$G$ over $D$, and then search $G$ bottom-up to find answers.

Let $P_1$*, $P_2$*, ..., $P_n$* be all the $p$*-graphs constructed for all $D_j$'s in $D$, respectively. Let $p_j$ and $S_j$* ($j = 1, ..., n$) be the main path of $P_j$* and the transitive closure over its spans, respectively. We will construct $G$ in two steps. In the first step, we will establish a *trie* [9], denoted as $T = trie(R)$ over $R = \{p_1, ..., p_n\}$ as follows.

If $|R| = 0$, $trie(R)$ is, of course, empty. For $|R| = 1$, $trie(R)$ is a single node. If $|R| > 1$, $R$ is split into $m$ (possibly empty) subsets $R_1$, $R_2$, ..., $R_m$ so that each $R_i$ ($i = 1, ..., m$) contains all those sequences with the same first variable name. The tries: $trie(R_1)$, $trie(R_2)$, ..., $trie(R_m)$ are constructed in the same way except that at the $k$th step, the splitting of sets is based on the $k$th variable name (along the global ordering of variables). They are then connected from their respective roots to a single node to create $trie(R)$.

In Fig. 2, we show the trie constructed for the variable sequences shown in the third column of Table I. In such a trie, special attention should be paid to all the leaf nodes each labeled with $, representing a conjunction (or a subset of conjunctions, which can be satisfied under the truth assignment represented by the corresponding main path.)

Each edge in the trie is referred to as a tree edge. In addition, the variable $c$ associated with a node $v$ is referred to as the label of $v$, denoted as $l(v) = c$. Also, we will associate each node $v$ in the trie $T$ a pair of numbers (*pre*, *post*) to speed up recognizing ancestor/descendant relationships of nodes in $T$, where *pre* is the order number of $v$ when searching $T$ in preorder and *post* is the order number of $v$ when searching $T$ in postorder.

These two numbers can be used to characterize the ancestor-descendant relationships in $T$ as follows.

- Let $v$ and $v'$ be two nodes in $T$. Then, $v'$ is a descendant of $v$ iff $pre(v') > pre(v)$ and $post(v') < post(v)$.

For the proof of this property of any tree, see Exercise 2.3.2-20 in [8].

For instance, by checking the label associated with $v_2$ against the label for $v_9$ in Fig. 2, we see that $v_2$ is an ancestor of $v_9$ in terms of this property. We note that $v_2$'s label is (3, 12) and $v_9$'s label is (10, 6), and we have $3 < 10$ and $12 > 6$. We also see that since the pairs associated with $v_{14}$ and $v_6$ do not satisfy the property, $v_{14}$ must not be an ancestor of $v_6$ and *vice versa*.

In the second step, we will add all $S_i$* ($i = 1, ..., n$) to the trie $T$ to construct a trie-like graph $G$, as illustrated in Fig. 3. This trie-like graph is constructed for all the variable sequences
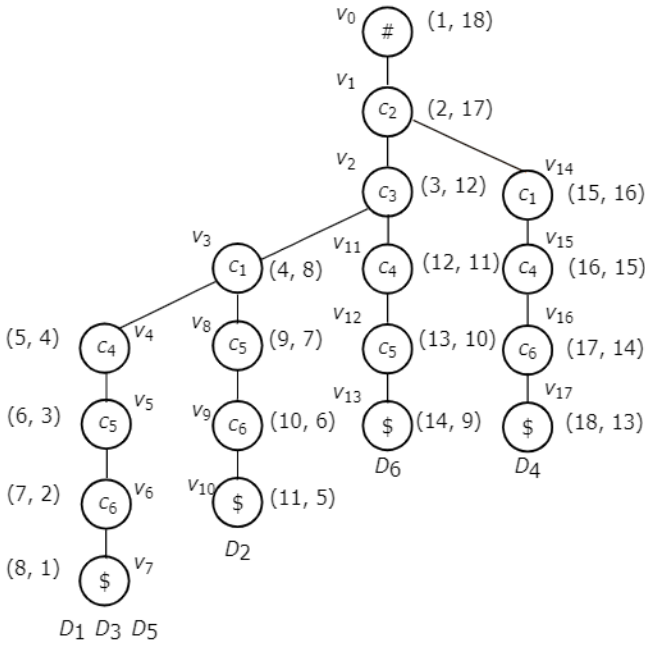
FIG. 2: A trie and tree encoding.

given in Table I, in which each span is associated with a set of numbers used to indicate what variable sequences the span belongs to. For example, the span $<v_0, v_1, v_2>$ (in Fig. 3) is associated with three numbers: 1, 5, 6, indicating that the span belongs to 3 conjunctions: $D_1$, $D_5$, and $D_6$. But no numbers are associated with any tree edges. In addition, each $p^*$-graph itself is considerred to be a simple trie-like graph.

From Fig. 3, we can see that although the number of truth assignments for $D$ is exponential, they can be represented by a graph with polynomial numbers of nodes and edges.
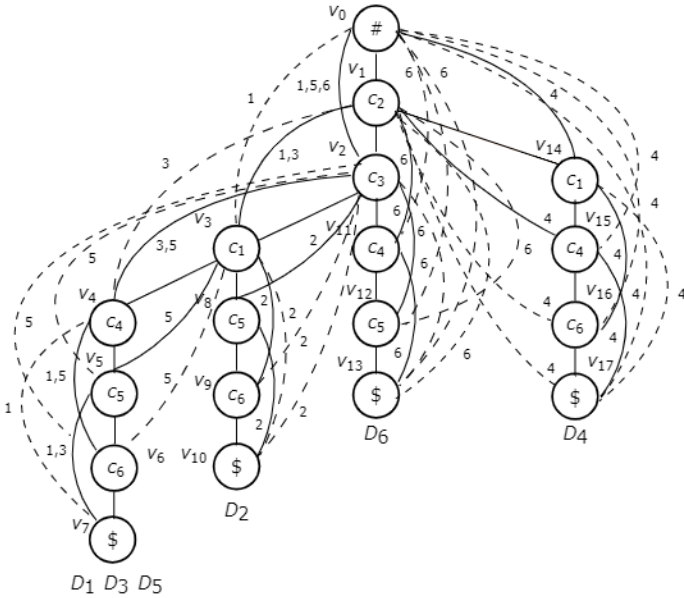


FIG. 3: A trie-like graph $G$.

In a next step, to find the answer, we will search $G$ bottom-up level by level. First of all, for each leaf node, we will figure out all its parents. Then, all such parent nodes will be categorized into different groups such that the nodes in the same group will have the same label (variable name), which enables us to recognizes all those conjunctions which can be satisfied by a same assignment efficiently. All the groups containing only a single node will not be further explored. (That is, if a group contains only one node $v$, the parent of $v$ will not be checked.) Next, all the nodes with more than one node will be explored. We repeat this process until we reach a level at which each group contains only one node. In this way, we will find a set of subgraphs, each rooted at a certain node $v$, in which the nodes at the same level must be labeled with the same variable name. Then, the path in the trie from the *root* to $v$ and any path from $v$ to a leaf node in the subgraph correspond to an assignment satisfying all the conjunctions labeling a leaf node in it.

See Fig. 4 for illustration.

In Fig. 4, we show part of the bottom-up process of searching the trie-like graph $G$ shown in Fig. 3.

- step 1: The leaf nodes of $G$ are $v_7$, $v_{10}$, $v_{13}$, $v_{17}$ (see level 1), representing the 6 variable sequences in $D$ shown in Table I, respectively. (Especially, node $v_7$ alone represents three of them: $D_1$, $D_3$, $D_5$.) Their parents are all the remaining nodes in $G$ (see level 2 in Fig. 4). Among them, $v_6$, $v_9$, $v_{16}$ are all labeled with the same variable name '$c_6$' and will be put in a group $g_1$. The nodes $v_5$, $v_8$, and $v_{15}$ are labeled with '$c_5$' and will be put in a second group $g_2$. The nodes $v_4$, $v_{11}$, and $v_{15}$ are labeled with '$c_4$' and will be put in the third group $g_3$. Finally, the nodes $v_3$ and $v_{14}$ are labeled with '$c_1$' and are put in group $g_4$. All the other nodes: $v_0$, $v_1$, $v_2$ each are differently labeled and therefore will not be further explored.

- step 2: The parents of the nodes in all groups $g_1$, $g_2$, $g_3$, and $g_4$ will be explored. We first check $g_1$. The parents of the nodes in $g_1$ are shown at level 3 in Fig. 4. Among them, the nodes $v_5$ and $v_8$ are labeled with '$c_5$' and will be put in a same group $g_{11}$; the nodes $v_4$ and $v_{15}$ are labeled with '$c_4$' and put in another group $g_{12}$; the nodes $v_3$ and $v_{14}$ are labeled with '$c_1$' and put in group $g_{13}$. Again, all the remaining nodes are differently labeled and will not be further considered. The parents of $g_2$, $g_3$, and $g_4$ will be handled in a similar way.

- step 3: The parents of the nodes in $g_{11}$, $g_{12}$, $g_{13}$, as well as the parents of the the nodes in any other group whose size is larger than 1 will be checked. The parents of the nodes in $g_{11}$ are $v_2$, $v_3$, and $v_4$. They are differently labeled and will not be further explored. However, among the parents of the nodes in group $g_{12}$, $v_4$ and $v_{15}$ are labeled with '$c_1$' and will be put in a group $g_{121}$. The parents of the nodes in $g_{13}$ are also differently labeled and will not be searched. Again, the parents of all the other groups at level 3 in Fig. 4 will be checked similarly.

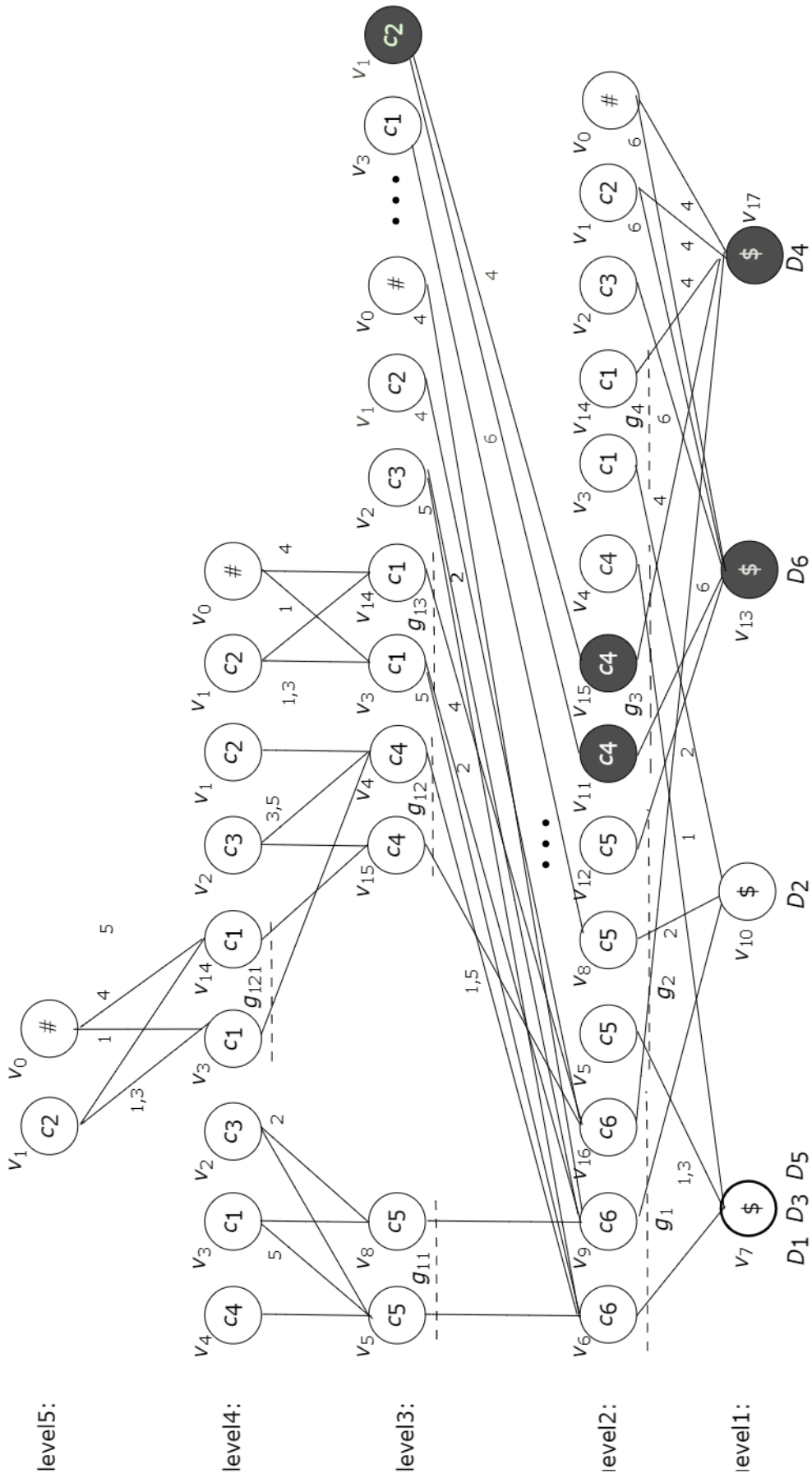- step 4: The parents of the nodes in $g_{121}$ and in any other

FIG. 4: Illustration for the layered representation $G'$ of $G$.

groups at level 4 in Fig. 4 will be explored. Since the parents of the nodes in $g_{121}$ are differently labeled the whole working process terminates if the parents of the nodes in any other other groups at this level are also differently labeled.

We call the graph illustrated in Fig. 4 a layered representation $G'$ of $G$. From this, a maximum subset of conjunctions satisfied by a certain truth assignment represented by a subset of variables that are set to 1 (while all the remaining variables are set to 0) can be efficiently calculated. As mentioned above, each node which is the unique node in a group will have no parents. We refer to such a node as a *s-root*, and the subgraph made up of all nodes reachable from the *s-root* as a rooted subgraph. For example, the subgraph made up of the blackened nodes in Fig. 4 is one of such subgraphs.

Denote a rooted subgraph rooted at node $v$ by $G_v$. In $G_v$, the path labels from $v$ to a leaf node are all the same. Then, any conjunction $D_i$ associated with a leaf node $u$ is satisfied by a same truth assignment $\sigma$:

$\sigma$ = {the lables on the path $P$ from $v$ to $u$} $\cup$ {the labels on the path from the the root of the whole trie to $v$},

if any edge on $P$ is a tree edge or the set of numbers assiciated with it contains $i$. We call this condition the *assignment condition*.

For instance, in the rooted subgraph mentioned above (represented by the blackened nodes in Fig. 4), we have two root-to-leaf paths: $v_1 \xrightarrow{6} v_{11} \xrightarrow{6} v_{13}$, $v_1 \xrightarrow{4} v_{15} \xrightarrow{4} v_{17}$, with the same path label; and both satisfy the *assignment condition*. Then, this rooted subgraph represents a subset: $\{D_4, c_6\}$, which are satisfied by a truth assignment: $\{c_2, c_4\} \cup \{c_2\} = \{c_2, c_4\}$ (i.e., $\{c_2 = 1, c_4 = 1, c_1 = 0, c_3 = 0, c_5 = 0, c_6 = 0\}$)

Now we consider the node $v_4$ at level 4 in Fig. 4. The rooted subgraph rooted at it contains only one path $v_4 \to v_5 \to v_6$, where each edge is a tree edge and $v_6$ represents $\{D_1, D_3, c_5\}$. This path corresponds to a truth assignment $\sigma = \{c_4, c_5, c_6\} \cup \{c_1, c_2, c_3\} = \{c_1, c_2, c_3, c_4, c_5, c_6\}$ (i.e., $\sigma = \{ c_1 = 1, c_2 = 1, c_3 = 1, c_4 = 1, c_5 = 1, c_6 = 1\}$), showing that under $\sigma$: $D_1, D_3, D_5$ evaluate to *true*, which are in fact a maximum subset of satisfied conjunctions in $D$. From this, we can deduce that in the formula $C$ we must also have a maximum set of three satisfied clauses. Also, according to $\sigma$, we can quickly find those three satisfied clauses in $C$.

In terms of the above discussion, we give the following algorithm. In the algorithm, a *stack* $S$ is used to explore $G$ to form the layered graph $G'$. In $S$, each entry is a subset of nodes labeled with a same variable name.

The algorithm can be divided into two parts. In the first part (lines 2 - 12), we will find the layered representation $G'$ of $G$. In the second part (line 13), we call subprocedure *findSubset( )*, by which we check all the rooted subgraphs to find a truth assignment such that the satisfied conjunctions are maximized. This is represented by a triplet $(u, s, f)$, corresponding to a rooted subgraph $G_u$ rooted at $u$ in $G'$. Then, the variable names represented by the path from the root of the whole trie to $u$ and the variable names represented by any path in $G'$

---

**Algorithm 1:** *SEARCH(G)*

**Input** : a trie-like graph $G$.
**Output**: a largest subset of conjunctions satisfying a certain truth assignment.

1   $G'$ :={all leaf nodes of $G$}; $g$ := {all leaf nodes of $G$};
2   push($S$, $g$);           (\* find the layered graph $G'$ of $G$ \*)
3   **while** $S$ *is not empty* **do**
4      $g$ := pop($S$);
5      find the parents of each node in $g$; add them to $G'$;
6      divide all such parent nodes into several groups: $g_1, g_2, ..., g_k$ such that all the nodes in a group with the same label;
7      **for** *each* $j \in \{1, ..., k\}$ **do**
8         **if** $|g_j| > 1$ **then**
9            push($S$, $g_j$);
10   return $findSubset(G')$;

---

**Algorithm 2:** *findSubset(G')*

**Input** : a layered graph $G'$.
**Output**: a largest subset of conjunctions satisfying a certain truth assignment.

1   $(u, s, f)$ := $(null, 0, \Phi)$; (\* find a truth assignment satisfying a maximum subset of conjunctions. $\Phi$ represents an *empty* set.\*)
2   **for** *each rooted subgraph* $G_v$ **do**
3      determine the subset $D'$ of satisfied conjunctions in $G_v$;
4      **if** $|D'| > s$ **then**
5         $u$ := $v$; $s$ := $|D'|$; $f$ := $D'$;
6   return $(u, s, f)$;

---

make up a truth assignment that satisfies a largest subset of conjunctions stored in $f$, whose size is $s$.

Concerning the correctness of the algorithm, we have the following proposition.

**Proposition 2.** *Let $D$ be a formula in DNF. Let $G$ be a trie-like graph created for $D$. Then, the result produced by $SEARCH(G)$ must be a truth assignment satisfying a maximum subset of conjunctions in $D$.*

*Proof.* By the execution of $SEARCH(G)$, we will first generate the layered reprsentation $G'$ of $G$. Then, all the rooted subgraphs in $G'$ will be checked. By each of them, we will find a truth assignment satisfying a subset of conjunctions, which will be compared with the largest subset of conjunctions found up to now. Only the larger between them is kept. Therefore, the result produced by $SEARCH(G)$ must be correct. □

## IV. IMPROVEMENTS

### A. Redundancy analysis

The working process of constructing the layered representation $G'$ of $G$ does a lot of redundant work, but can be effectively removed by interleaving the process of *SEARCH* and *findSubset* in some way. We will recognize any rooted subgraph as early as possible, and remove the relevant nodes to avoid any possible redundancy. To see this, let us have a look at Fig. 5, in which we illustrate part of a possible layered graph, and assume that from group $g_1$ we generate another two groups $g_2$ and $g_3$. From them a same node $v_3$ will be accessed. This shows that the number of the nodes at a layer in $G'$ can be larger than O($nm$) (since a node may appear more than once.)

Fortunately, such kind of repeated appearance of a node can be avoided by applying the *findSubset* procedure multiple times during the execution of *SEARCH*( ) with each time applied to a subgraph of $G'$, which represents a certain truth assignment satisfying a subset of conjunctions that cannot be involved in any larger subset of satisfiable conjunctions.

For this purpose, we need first to recognize what kinds of subgraphs in a trie-like graph $G$ will lead to the repeated appearances of a node at a layer in $G'$.

In general, we distingush among three cases, by which we assume two nodes $u$ and $v$ respectively appearing in $g_2$ and $g_3$ (in Fig. 5), with $v_3 \rightarrow u$, $v_3 \rightarrow v \in G$.
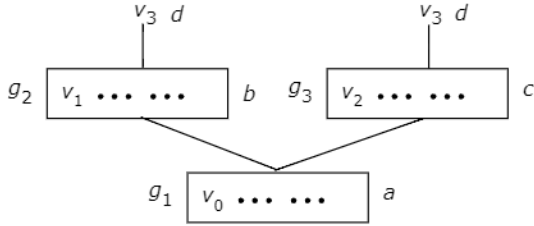


FIG. 5: A possible part in a layered graph $G'$.

- Case 1: $u$ and $v$ appear on different paths in $G$, as illustrated in Fig. 6(a)), in which nodes $v_1$ and $v_2$ are differently labelled. Thus, when we create the corresponding layered representation, they will belong to different groups, as shown in Fig. 6(b), matching the pattern shown in Fig. 5.
- Case 2: $u$ and $v$ appear on a same path in $G$, as illustrated in Fig. 6(c)), in which two nodes $v_1$ and $v_2$ appear on a same path (and then must be differently labelled.) Hence, when we create the corresponding layered representation, they definitely belong to different groups, as illustrated in Fig. 6(d), also matching the pattern shown in Fig. 5.
- Case 3: The combination of Case 1 and Case 2. To know what it means, assume that in $g_1$ (in Fig. 5) we have two nodes $u$ and $u'$ with $u \rightarrow v_3$ and $u' \rightarrow v_3$. Thus, if $u$ and $v$ appear on different paths, but $u'$ and $v$ on a same path in $T$, then we have Case 3, by which Case 1
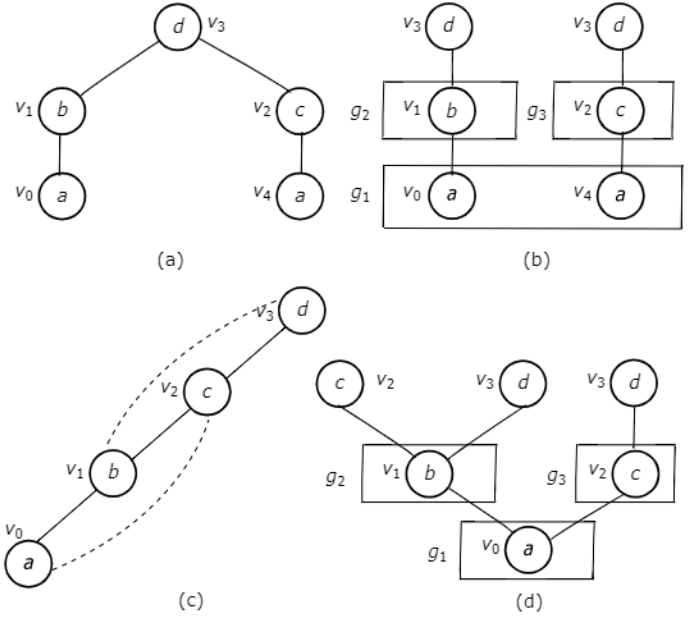


FIG. 6: Two reasons for repeated appearances of nodes at a level in $G'$.

and Case 2 occur simuteneously by a repeated node at a certain layer in $G'$.

Case 1 and Case 2 can be efficiently differetiated by using the tree encoding illuatrated in Fig. 2.

In Case 1 (as illustrate in Fig. 6(a) and (b)), a node $v$ which appears more than once at a level in $G'$ must be a branching node (i.e., a node with more than one child) in $T$. Thus, each subset of conjunctions represented by all those substrees repectively rooted at the same-labelled children of $v$ must be a largest subset of conjunctions that can be satisfied by a truth assignment with $v$ (or say, the variable represented by $v$) being set to *true*. Therefore, we can merge all the repeated nodes to a single one and call *findSubset*( ) immediatly to find all such subsets for all the children of $v$.

In Case 2 (as illustrated in Fig. 6(c) and (d)), some more effort should be made. In this case, the multiple appearances of a node $v$ at a level in $G'$ correspond to more than one descendants of $v$ on a same path in $T$: $v_1, v_2, ..., v_k$ for some $k > 1$. (As demonstrated in Fig. 6(c), both $v_1$ and $v_2$ are $v_3$'s descendants.) Without loss of generality, assume that $v_1 \Leftarrow v_2 \Leftarrow ... \Leftarrow v_k$, where $v_i \Leftarrow v_{i+1}$ represents that $v_i$ is a descendant of $v_{i+1}$ ($1 \leq i \leq k - 1$).

In this case, we will merge the multiple appearances of $v$ to a single appearance of $v$ and connect $v_k$ to $v$. Any other $v_i$ ($i \in \{1, 2, ..., k- 1\}$) will be simply connected to $v$ if the following condition is satisfied.

- $v_i$ appears in a group which contains at least another node $u$ such that $u'$s parent is diffrent from $v$, but with the same label as $v$.

Otherwise, $v_i$ ($i \in \{1, 2, ..., k- 1\}$) will not be connected to $v$. It is because if the condition is not met the truth assignment

represented by the path in $G$, which contains the span $(v, v_i)$, cannot satisfy any two or more conjunctions. But the single satisfied conjunction is already figured out when we create the trie at the very beginning. However, we should know that this checking is only for efficiency. Whether doing this or not will not impact the correctness of the algorithm or the worst-case running time analysis.

Based on Case 1 and Case 2, Case 3 is easy to handle. We only need to check all the children of the repeated nodes and carefully distinguish between Case 1 and Case 2 and handle them differently.

See Fig. 4 and 7 for illustration.

First, we pay attention to $g_1$ and $g_2$ at level 2 in Fig. 4, especially nodes $v_6$ and $v_9$ in $g_1$, and $v_8$ in $g_2$, which match the pattern shown in Fig. 5. As we can see, $v_6$ and $v_8$ are on different paths in $T$ and then we have Case 1. But $v_9$ and $v_8$ are on a same path, which is Case 2. To handle Case 1, we will search along two paths in $G'$: $v_3 \xrightarrow{1,5} v_6 \rightarrow v_7$ (labeled with $\{D_1, D_3, D_5\}$), $v_3 \rightarrow v_8 \rightarrow v_{10}$ (labeled with $\{D_2\}$), and find a subset of three conjunctions $\{D_1, D_5, D_2\}$, satisfied by a truth assignment: $\{c_1 = 1, c_2 = 1, c_3 = 1, c_4 = 0, c_5 = 0, c_6 = 1\}$. To handle Case 2, we simply connect $v_8$ to the first appearance of $v_3$ as illustrated in Fig. 7 and then eliminate second appearance of $v_3$ from $G'$.

*B. Improved algorithm*

In terms of the above discussion, the method to generate $G'$ should be changed. We will now generate $G'$ level by level. After a level is created, the repeated appearances of nodes will be checked and then eliminated. In this way, the number of nodes at each layer can be kept $\leq O(nm)$.

However, to facilitate the recognition of truth assignments for the corresponding satisfied conjunctions, we need a new concept, the so-called *reachable subsets* of a node $v$ through spans, denoted as $RS_v$.

**Definition 3.** (reachable subsets through spans) Let $v$ be a repeated node of Case 1. Let $u$ be a node on the tree path from *root* to $v$ in $G$ (not including $v$ itself). A reachable subset of $u$ through spans are all those nodes with a same label $c$ in different subgraphs in $G[v]$ and reachable from $u$ through a span, denoted as $RS_u[c]$.

For instance, for node $v_2$ in Fig. 3 (which is on the tree path from *root* to $v_3$ (a repeated node of Case 1), we have two $RS$s:

- $RS_{v_2}[c_5] = \{v_5, v_8\}$,
- $RS_{v_2}[c_6] = \{v_6, v_9\}$.

We have $RS_{v_2}[c_5]$ due to two spans $v_2 \xrightarrow{5} v_5$ and $v_2 \xrightarrow{2} v_8$ going out of $v_2$, respectively reaching $v_5$ and $v_8$ on two different $p^*$-graphs in $G[v_3]$ with $l(v_5) = l(v_8) = {}'c_5'$. We have $RS_{v_2}[c_6]$ due to another two spans going out of $v_2$: $v_2 \xrightarrow{5} v_6$ and $v_2 \xrightarrow{2} v_9$ with $l(v_6) = l(v_9) = {}'c_6'$.

In general, we are interested only in $RS_v$'s with $|RS_v| \geq 2$. So, in the subsequent discussion, by an $RS_v$, we mean an $RS_v$ with $|RS_v| \geq 2$.

The definition of this concept for a repeated node $v$ of Case 1 is a little bit different from any other node on the tree path (from *root* to $v$). Specifically, each of its $RS$s is defined to be a subset of nodes reachable from a span or from a tree edge. So for $v_3$ we have:

- $RS_{v_3}[c_5] = \{v_5, v_8\}$,
- $RS_{v_3}[c_6] = \{v_6, v_9\}$,

respectively due to $v_3 \xrightarrow{5} v_5$ and $v_3 \rightarrow v_8$ going out of $v_3$ with $l(v_6) = l(v_8) = {}'c_5'$; and $v_3 \xrightarrow{5} v_6$ and $v_3 \xrightarrow{2} v_9$ going out of $v_3$ with $l(v_6) = l(v_8) = {}'c_6'$.

Based on the concept of reachable subsets through spans, we are able to define another more important concept, upper boundaries (denoted as *upBounds*), given below.

**Definition 4.** (upper boundaries) Let $v$ be a repeated node of Case 1. Let $G_1, G_2, ..., G_k$ be all the subgraphs rooted at a child of $v$ in $G$. Let $RS_{v_i}[c_j]$ ($i = 1, ..., l; j = 1, ..., q$ for some $l, q$) be all the reachable subsets through spans. An upBound with respect to $v$ is a subset of nodes $\{u_1, u_2, ..., u_f\}$ with the following properties:

1) Each $u_i$ ($1 \leq i \leq l$) appears in some $G_j$ ($1 \leq j \leq k$).
2) For each pair $u_i, u_i$ ($i \neq j$) they are not related by the ancestor/descendant relationship.
3) Each $u_i$ ($i = 1, ..., l$) is in some $RS_q[c_r]$. But there is no any other $RS_{q'}[c_{r'}]$, containing a node at a higher position than $u_i$ (or say, a node closer to the *root* than $u_i$) in the same subgraph.

Fig. 8 gives an intuitive illustration of this concept.

As a concrete example, consider $v_5$ and $v_8$ in Fig. 3. They make up an upBound with respect to $v_3$ (repeated node of Case 1). Then, we will construct a trie-like graph over two subgraphs, rooted at $v_5$ and $v_8$, respectively. This can be done by a recursive call of the algorithm itself. Here, however, $v_4$ is not included since the truth assignment with $v_4$ being set to *true* satisfies only the conjunctions associated with leaf node $v_{10}$. This has already been determined when the initial trie is built up. In fact, the purpose of upper boundaries is to take away the nodes like $v_4$ from the subsequent computation.

Specifically, the following operations will be carried out when meet a repeated node $v$ of Case 1.

- Calculate all $RS$s with respect $v$.
- Calculate the upBound in terms of $RS$s.
- Make a recursive call of the algorithm over all the $p^*$-subgraphs each starting from a node on the corresponding upBound.
- Merge the repeated nodes of Case 1 to a single one at the corresponding layer in $G'$.

See the following example for illustration.

**Example 1.** *When checking the repeated node $v_3$ in the bottom-up search process, we will calculate all the reachable subsets through spans with respect to $v_3$ as described above: $RS_{v_2}[c_5]$, $RS_{v_2}[c_6]$, $RS_{v_3}[c_5]$, and $RS_{v_3}[c_6]$. In terms of these reachable subsets through spans, we will get the corresponding upBound $\{v_5, v_8\}$. Node $v_4$ (above the upBound) will not be involved by the recursive execution of the algorithm.*
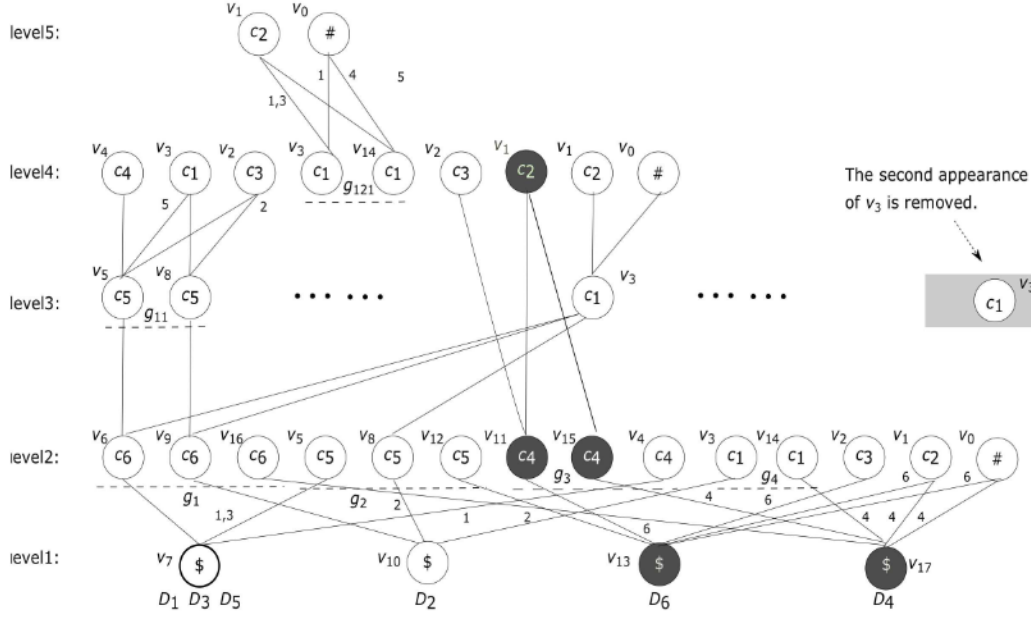
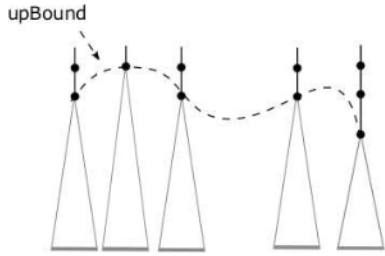FIG. 7: Illustration for removing repeated nodes.
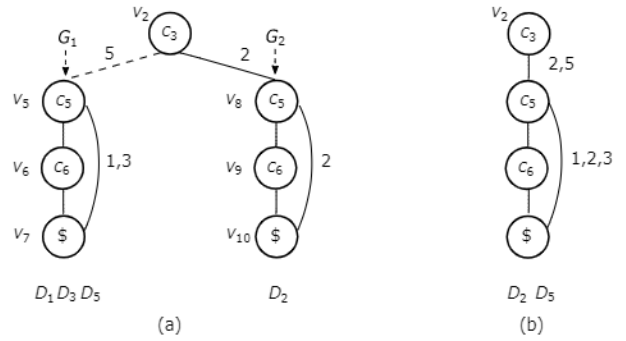


FIG. 8: Illustration for upBounds.



FIG. 9: Illustration for recursive call of the algorithm.

Concretely, when we make a recursive call of the algorithm, applied to two subgraphs: $G_1$ - rooted at $v_5$, and $G_2$ - rooted at $v_8$ (see Fig. 9(a)), we will first constrcut a trie-like graph as shown in Fig. 9(b). Here, we notice that the subset associated with its unique leaf node is $\{D_2, D_5\}$, instead of $\{D_1, D_2, D_3, D_5\}$. It is because the number associated with span $v_2 \xrightarrow{5} v_5$ is 5 while the number associated with span $v_2 \xrightarrow{2} v_8$ is 2.

By searching the trie-like graph shown in Fig. 9(b), we will find the truth assignment satisfying $\{D_2, D_5\}$. This truth assignment is represented by a path consisting three parts: the tree path from *root* to $v_2$, the span $v_2 \xrightarrow{5} v_5$, the subpath $v_5$ to $v_7$. So the truth assignment is $\{c_1 = 0, c_2 = 1, c_3 = 1, c_4 = 0, c_5 = 1, c_6 = 1\}$.

We remember that when generating the trie $T$ over the main paths of the $p*$-graphs over the variable sequences shown in Table I, we have already found a subset of conjunctions $\{D_1, D_3, D_5\}$, which can be satisfied by a truth assignment

represented by the corresponding main path. This is larger than $\{D_2, D_5\}$. Therefore, $\{D_2, D_5\}$ should not be kept around and this part of computation is futile. However, this kind of useless work can be avoided by performing a pre-checking: if the number of $p*$-subgraphs, over which the recursive call of the algorithm will be invoked, is smaller than the size of the partial answer already obtained, the recursive call of the algorithm should not be conducted.

In terms of the above discussion, we change *SEARCH*( ) to a recursive algorithm shown below.

The improved algorithm (Algorithm 3) works in a quite different way from Algorithm 1. Concretely, $G'$ will be created level by level (see line 6), and for each created level all the multiple appearances of nodes will be recognized and handled according to the three cases described in the previous

**Algorithm 3:** *SEARCH(D)*

**Input** : a set of p*-graphs or a set of subgraphs $D$.
**Output:** a largest subset of conjunctions satisfying a
certain truth assignment.

1 let $D = \{G_1{}^*, G_2{}^*, ..., G_n{}^*\}$;
2 construct a trie-like graph $G$ over $G_1{}^*, G_2{}^*, ..., G_n{}^*$;
3 assume that the height of $G$ is $m$;
4 let $L_1 = \{$all the leaf nodes of $G\}$;
5 **for** $i = 1$ *to* $m$ - 1 **do**
6     generate $L_{i+1}$ from $L_i$; (*each node in $L_{i+1}$ is a parent of some nodes in $L_i$*)
7     **for** *each repeated node $v$ in $L_{i+1}$* **do**
8         Case 1: calculate $RS$s with respect to $v$ and the corresponding upBound; let $D'$ be the set of all the subgraphs each rooted at a node on upBound; call *SEARCH(D')*; Merge all the appearances of $v$ to a single one;
9         Case 2: merge all the multiple appearances of $v$ to a single node;
10         Case 3: distinguish between the nodes of Case 1 and the nodes of Case 2; and handle them differently;
11 denote by $G'$ the generated layered graph;
12 return $findSubset(G')$;

subsection (see lines 7 - 10). Especially, in Case 1, a recursive call to the algorithm itself will be invoked.

The sample trace given in the following example helps for illustration.

**Example 2.** *When applying SEARCH( ) to the p*-graphs shown in Fig. 3, we will meet three repeated nodes of Case 1: $v_3$, $v_2$, and $v_1$.*

- Intially, when creating $T$, a subset of conjunctions $\{D_1, D_2, D_5\}$, is found (see Fig. 2), which can be satisfied by a same truth assignment: $c_1 = 1$, $c_2 = 1$, $c_3 = 1$, $c_4 = 1$, $c_5 = 1$, $c_6 = 1$. (See Fig. 2.)
- Checking $v_3$. As shown in Example 1, by this checking, we will find a subset of conjunction $\{D_2, D_5\}$ satisfied by a truth assignment $\{c_1 = 0, c_2 = 1, c_3 = 1, c_4 = 0, c_5 = 1, c_6 = 1\}$, smaller than $\{D_1, D_2, D_5\}$. Thus, this result will not be kept around.
- Checking $v_2$. When we meet this repeated node of Case 1 during the generation of $G'$, we have two subgraphs in $G'[v_2]$, as shown in Fig. 10.

  With spect to $v_2$, we will calculate all the relevant reachable subsets through spans for all the nodes on the tree path from *root* to $v_2$ in $G$. Altogether we have five reachable subsets through spans. Among them, associated with $v_1$ (on the tree path from *root* to $v_2$ in Fig. 3), we have
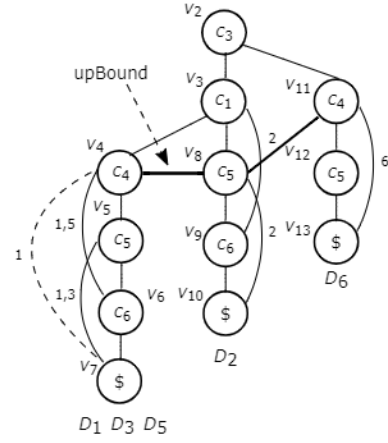


FIG. 10: Two subgraphs in $G'[v_2]$ and a upBound.

- $RS_{v_1}[c_4] = \{v_4, v_{11}\}$,

due to the following two spans (see Fig. 3):

- $\{v_1 \xrightarrow{3} v_4, v_1 \xrightarrow{6} v_{11}\}$.

Associated with $v_2$ (the repeated node itself) have we the following four reachable subsets through spans:

- $RS_{v_2}[c_4] = \{v_4, v_{11}\}$,
- $RS_{v_2}[c_5] = \{v_5, v_8, v_{12}\}$,
- $RS_{v_2}[c_6] = \{v_6, v_9\}$,
- $RS_{v_2}[\$] = \{v_{10}, v_{13}\}$,

due to four groups of spans shown below:

- $\{v_2 \xrightarrow{3,5} v_4, v_2 \xrightarrow{6} v_{11}\}$,
- $\{v_2 \xrightarrow{5} v_5, v_2 \xrightarrow{2} v_8, v_2 \xrightarrow{6} v_{12}\}$,
- $\{v_2 \xrightarrow{5} v_6, v_2 \xrightarrow{2} v_9\}$,
- $\{v_2 \xrightarrow{2} v_{10}, v_2 \xrightarrow{6} v_{13}\}$.

In terms of the reachable subsets through spans, we can establish the corresponding upper boundary $\{v_4, v_8, v_{11}\}$ (which is illustrated as a thick line in Fig. 10). Then, we can determine what subgraphs will be utilized to establish a trie-like graph, over which the algorithm is recursively executed.

In Fig. 11(a), we show the trie-like graph built over the three p*-subgraphs (starting respectively from $v_4$, $v_8$, $v_{11}$ on the upBound shown in Fig. 10), in which $v_{4-11}$ stands for the merging of $v_4$ and $v_{11}$, and $v_{5-12}$ for the merging of $v_5$ and $v_{12}$. Especially, $v_2$ should be involved, working as a bridge between the newly contructed trie-like graph and the rest part of $G$. However, this part of the operation is not specified in Algorithm *SEARCH( )* for ease of explanation. But it can be easily extended with this operation included.

By a recursive call of *SEARCH( )*, we will construct this graph and then search this graph bottom up, by which we will create a layed graph as shown in Fig. 12. At level 2 in Fig.
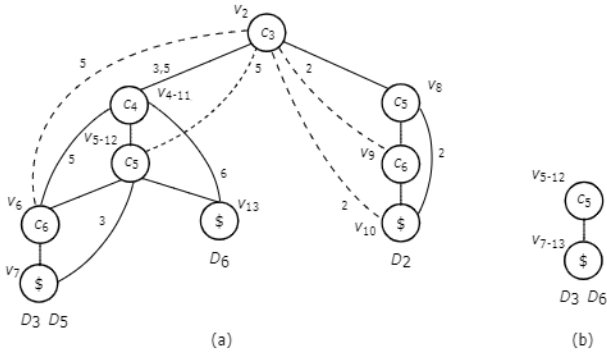
FIG. 11: A trie-like graph and a path.

12, we can see a repeated nodes of Case 1: node $v_{5-12}$. Then, we will make a recursive call of the algorithm, generating an upBound is $\{v_7, v_{13}\}$. Accordingly, we will find a single path as shown in Fig. 11(b), by which we will find a largest subset of conjunctions $\{D_3, D_6\}$, which can be satisfied by a certain truth assignment. We notice that the subset associated with this path is $\{D_3, D_6\}$, instead of $\{D_3, D_5, D_6\}$. It is because the span from $v_{5-12}$ to $v_7$ (in Fig. 11(a)) is labeled with 3 and $D_5$ should be removed.
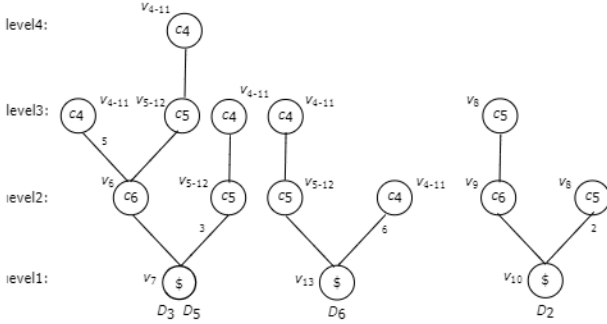


FIG. 12: A naive bottom-up search of $G$.

When we meet the second repeated node $v_2$, we will create the following $RS$'s:

- $RS_{v_1} = \Phi$, (Note that any $RS$ with $|RS| < 2$ will not be considered.)
- $RS_{v_2}[c_5] = \{v_{5-12}, v_8\}$, (due to the span $v_2 \xrightarrow{5} v_{5-12}$ and the tree edge $v_2 \to v_8$.)
- $RS_{v_2}[c_6] = \{v_6, v_9\}$, (due to the spans $v_2 \xrightarrow{5} v_6$ and $v_2 \xrightarrow{2} v_9$.)

Accordingly, the corresponding upBound is $\{v_{5-12}, v_8\}$. Then, by the recursive execution of the algorithm, we will create a trie-like graph as shown in Fig. 13(a). The only branching node is $v_{5-12-8}$. Checking this node, we will finally get a single path as shown in Fig. 13(b), showing a largest subset of conjunctions which can be satisfied by a certain truth assignment.

In the whole working process, a simple but very powerful heuristics can be used to improve the efficiency. Let $\alpha$ be
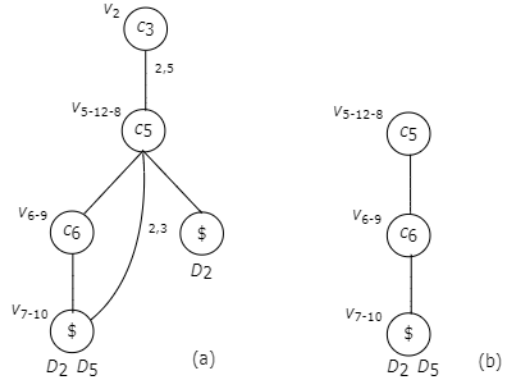


FIG. 13: Illustration for the recursive execution of the algorithm.

the size of the largest subset of conjunctions found up to now, which can be satisfied by a certain truth assignment. Then, any recursive call of the algorithm over a smaller than $\alpha$ subset of $p^*$-subgraphs will be surpressed.

After $v_2$ is removed from the corresponding levels in $G'$, the next repeated node $v_1$ of Case 1 will be checked in a way similar to $v_3$ and $v_2$.

Concerning the correctness of Algorithm 3, we have the following proposition.

**Proposition 3.** *Let $D$ be a logic formula in DNF. Applying Algorithm 3 to $D$, we will get a maximum subset of conjunctions satisfying a certain truth assignment.*

*Proof.* To prove the proposition, we first show that any subset of conjunctions found by the algorithm must be satisfied by a same truth assignment. This can be observed by the definition of $RS$s and the corresponding upBounds.

Then, we need to show that any subset of conjunctions, which can be satisfied by a certain truth assignment, can be found by the algorithm. For this purpose, consider a subset of conjunctions $D'$ which can be satisfied by a truth assignment represented by a path

$$v_1 \to v_2 \ldots v_{i-1} \to v_i \to v_j \ldots \to v_m,$$

in which $v_i \to v_j$ corresponds to a span. According to the definition of $RS$s, the algorithm will find this path. This analysis can be extended to the case that a path contains more than one span. $\square$

## V. TIME COMPLEXITY ANALYSIS

The total running time of the algorithm consists of three parts.

The first part $\tau_1$ is the time for computing the frenquencies of variable appearances in $D$. Since in this process each variable in a $D_i$ is accessed only once, $\tau_1 = O(nm)$.

The second part $\tau_2$ is the time for constructing a trie-like graph $G$ for $D$. This part of time can be further partitioned into three portions.

- $\tau_{21}$: The time for sorting variable sequences for $D_i$'s. It is obviously bounded by $O(nm\log_2 m)$.

- $\tau_{22}$: The time for constructing $p$*-graphs for each $D_i$ ($i = 1, ..., n$). Since for each variable sequence a transitive closure over its spans should be first created and needs $O(m^2)$ time, this part of cost is bounded by $O(nm^2)$.
- $\tau_{23}$: The time for merging all $p$*-graphs to form a trie-like graph $G$, which is also bounded by $O(nm^2)$.

The third part $\tau_3$ is the time for searching $G$ to find a maximum subset of conjunctions satisfied by a certain truth assignment. It is a recursive procedure. To analyze its running time, therefore, a recursive equation should established. Let $l = nm$ (the upper bound on the number of nodes in $T$). Assume that the average outdegree of a node in $T$ is $d$. Then the average time complexity of $\tau_3$ can be characterized by the following recurrence based on an observation that for each branching node a recursive call of the algorithm will be performed:

$$
\Gamma(l) = \begin{cases} O(1), & \text{if } l \leq \text{ a constant,} \\ \sum_{i=1}^{\lceil log_d\, l \rceil} d^i \Gamma(\frac{l}{d^i}) + O(l^2 m), & \text{otherwise.} \end{cases}
$$
(4)

Here, in the above recursuve equation, $O(l^2 m)$ is the cost for generating all the reachable subsets of a node through spans and upper boundries, together with the cost for generating all the trie-like subgraphs for each recursive call of the algorithm. We notice that the size of all the $RS$s together is bounded by the number of spans in $G$, which is $O(lm)$.

From (4), we can get the following inequality:

$$
\Gamma(l) \leq d \cdot log_d\, l \cdot \Gamma(\frac{l}{d}) + O(l^2 m).
$$
(5)

Solving this inequality, we will get

$$
\Gamma(l) \leq d \cdot log_d\, l \cdot \Gamma(\frac{l}{d}) + O(l^2 m)
$$

$$
\leq d^2 (log_d\, l)(\, log_d \tfrac{l}{d}) \Gamma(\tfrac{l}{d^2}) + (log_d\, l)\, l^2 m + l^2 m
$$

$$
\leq \,...\,...
$$

$$
\leq d^{\lceil log_d^l \rceil}(log_d\, l)\, (log_d(\tfrac{l}{d}))\, ...\, (\, log_d \tfrac{l}{d^{\lceil log_d^l \rceil}})
$$

$$
+ l^2 m((log_d\, l)(log_d(\tfrac{l}{d}))\, ...\, (log_d \tfrac{l}{d^{\lceil log_d^l \rceil}}) + ... + log_d\, l + 1)
$$

$$
\leq O(l(log_d\, l)^{log_d\, l} + O(l^2 m(log_d\, l)^{log_d\, l})
$$

$$
\sim O(l^2 m\, (log_d\, l)^{log_d\, l}).
$$
(6)

Thus, the value for $\tau_3$ is $\Gamma(l) \sim O(l^2 m\, (log_d\, l)^{log_d\, l})$.

From the above analysis, we have the following proposition.

**Proposition 4.** *The average running time of our algorithm is bounded by*

$$
\Sigma_{i=1}^3 \tau_i = O(nm) + (O(nm\, log_2\, m) + O(nm^2)
$$

$$
+ O(l^2 m\, (log_d\, l)^{log_d\, l})
$$
(7)

$$
= O(n^2 m^3 (log_d\, nm)^{log_d\, nm}).
$$

But we remark that if $d = 1$, we can immediately determine the maximum subset of satisfied conjunctions. It is just the set of conjunctions associated with the leaf node of the unique $p$*-graph.

Thus, it is reasonable to assume that in (7) we have $d > 1$.

The upper bound given above is much larger than the actual running time and cannot properly exhibit the quality of the algorithm. In the following, we give a worst-case time analysis which shows a much better running time complexity of the algorithm.

First, we notice that in all the generated trie-like subgraphs, the number of all the branching nodes is bounded by $O(nm)$. But each branching node may be involved in at most $O(n)$ recursive calls and for each recursive call at most $O(nm^2)$ time can be required to create the corresponding trie-like subgraph. Thus, the worst-case time complexity of the algorithm is bounded by $O(n^3 m^3)$.

In the above discussion, we claim that each branching node can be involved at most in $O(n)$ recursive calls based on the following analysis. Let $v$ be a top branching node, which does not have any ancestor branching node. Let $P$ be a longest path from *root* through $v$ to a leaf node in $G$. Then, corresponging to each descendant $u$ of $v$ on $P$, we may have a recursive call on a $G'[v]$ if there is a span from an ancestor of $v$ to $u$, where $G'$ is a dynamically created trie-like subgraph (see lines 14 - 16 of the algorithm). Thus, in the worst case, $v$ can be involved in $n$ - 1 recursive calls. We call all such recursive calls a $v$-recursion. Now, we consider another branching node $v'$, which is a descendant of $v$. But between $v$ and $v'$ there is not any other branching node. For the same reason, $v'$ can be involved in $n$ - 2 recursive calls. But it could also be involved in a $v$-recursion. So, $v'$ can be involved at most in $n$ - 2 + 1 = $n$ - 1 recursive calls. Further, let $v''$ be a descedant branching node of $v'$. Then, $v''$ can then also be involved in $n$ - 1 recursive calls according to the above analysis. It is because besides $n$ - 3 recursive calls involving $v''$, $v''$ can also be involved in a $v$-recursion and a $v'$-recursion.

In a similar way, we can analyze all the other branching nodes.

## VI. CONCLUSIONS

In this paper, we have presented a new method to solve the 2-MAXSAT problem. The worst-case time complexity of the algorithm is bounded by $O(n^3 m^3)$, where $n$ and $m$ are respectively the numbers of clauses and variables of a logic formula $C$ (over a set $V$ of variables) in *CNF*. The main idea behind this is to construct a different formula $D$ (over a set $U$ of variables) in *DNF*, according to $C$, with the property that

for a given integer $n* \leq n$ $C$ has at least $n*$ clauses satisfied by a truth assignment for $V$ if and only if $D$ has least $n*$ conjunctions satisfied by a truth assignment for $U$. To find a truth assignment that maximizes the number of satisfied conjunctions in $D$, a graph structure, called $p*$-graph, is introduced to represent each conjunction in $D$. In this way, all the conjunctions in $D$ can be represented as a trie-like graph. Searching $G$ bottom up, we can find the answer efficiently.

## REFERENCES

[1] S. A. Cook, *The complexity of theorem-proving procedures,* in: Proc. of the 3rd Annual ACM Symposium on the Theory of Computing, 1971, pp. 151-158.

[2] Y. Djenouri, Z. Habbas, D. Djenouri, *Data Mining-Based Decomposition for Solving the MAXSAT Problem: Toward a New Approach,* IEEE Intelligent Systems, Vol. No. 4, 2017, pp. 48-58.

[3] Y. Even, A. Itai, and A. Shamir, *On the complexity of timetable and multicommodity flow problems,,* SIAM J. Comput., 5 (1976), pp. 691-703.

[4] M. R. Garey, D. S. Johnson, and L. Stockmeyer, *Some simplified NP-complete graph problems,* Theoret. Comput. Sci., (1976), pp. 237-267.

[5] M.S. Johnson, *Approximation Algorithm for Combinatorial Problems,* J. Computer System Sci., 9(1974), pp. 256-278.

[6] M. Krentel, *The Complexity of Optimization Problems,* J. Computer and System Sci., 36(1988), pp. 490-509.

[7] R. Kohli, R. Krishnamurti, and P. Mirchandani, *The Minimum Satisfiability Problem,* SIAM J. Discrete Math., Vol. 7, No. 2, June 1994, pp. 275-283.

[8] D.E. Knuth, *The Art of Computer Programming,* Vol.1, Addison-Wesley, Reading, 1969.

[9] D.E. Knuth, *The Art of Computer Programming,* Vol.3, Addison-Wesley, Reading, 1975.

[10] A. Kügel, *Natural Max-SAT Encoding of Min-SAT,* in: Proc. of the Learning and Intelligence Optimization Conf., LION 6, Paris, France, 2012.

[11] C.M. Li, Z. Zhu, F. Manya and L. Simon, *Exact MINSAT Solving,* in: Proc. of 13th Intl. Conf. Theory and Application of Satisfiability Testing, Edinburgh, UK, 2010, PP. 363-368.

[12] C.M. Li, Z. Zhu, F. Manya and L. Simon, *Optimizing with minimum satisfiability,* Artificial Intelligence, 190 (2012) 32-44.

[13] A. Richard, *A graph-theoretic definition of a sociometric clique,* J. Mathematical Sociology, 3(1), 1974, pp. 113-126.

[14] C. Papadimitriou, *Computational Complexity,* Addison-Wesley, 1994.

[15] V. Vazirani, *Approximaton Algorithms,* Springer Verlag, 2001.

[16] H. Zhang, H. Shen, and F. Manyà, *Exact Algorithms for MAX-SAT,* Electronic Notes in Theoretical Computer Science 86(1):190-203, May 2003.