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^2m^4). 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, just to 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 [3]. The problem is *NP*-complete even when every clause in C has at most three literals [6]. The maximum satisfiability (MAXSAT) problem is an optimization version of satisfiabiltiy that seeks a truth assignment to maximize the number of satisfied clauses [9]. 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 [7]. Its application can be seen in an extensive biliography [4], [7], [12], [15]–[18], [20].

Over the past several decades, a lot of research on the MAXSAT has been conducted. Almost all of them are the approximation methods [1], [5], [9], [11], [19], [21], such as (1-1/e)-approximation, 3/4-approximation [21], as well as the method based on the integer linear programming [10]. The only algorithms for exact solution are discussed in [22], [23]. The worst-case time complexity of [23] is bounded by O($b2^m$), where b is the maximum number of the ccurrences of any variable in the clauses of C, while the worst-case time complexity of [22] is bounded by *max*$\{$O(2^m), O*(1.2989^n)$\}$. In both algorithms, the traditional branch-and-bound method is used for solving the satisfiability problem, which will search for a solution by letting a variable (or a literal) be 1 or 0. In terms of [8], any algorithm based on branch-and-bound runs in O*(c^m) time with $c \geq 2$.

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^2m^4), 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.

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) containing $2n$ 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 σ of variables such that under σ 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 a basic algorithm in Section III. Next, in Section IV, how to improve the basic algorithm is discussed. Section V is devoted to the analysis of the time complexity of the improved 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*.

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

The article is a modification and extension of a conference paper [2]: Y. Chen, The 2-MAXSAT Problem Can Be Solved in Polynomial Time, in Proc. CSCI2022, IEEE, Dec. 14-16, 2022, Las Vegas, USA, pp. 473-480.

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 [7], 2-MAXSAT is *NP*-complete.

To find a truth assignment σ such that the number of clauses set to $true$ is maximized under σ, we can try all the possible assignments, and count the satisfied clauses as discussed in [17], by which bounds are set up to cut short branches. We may also use a heuristic method to find an approximate solution to the problem as described in [9].

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 = $\{x_1, ..., x_n\}$. 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 σ 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 [12], 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 σ, (1) set both c_{j1} and x_j to *true* for $\tilde{\sigma}$. If c_{j1} is *false* and c_{j2} is *true* under σ, (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 σ for C, satisfying the following condition:

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

Clearly, under σ, we have at lease n^* clauses in C satisfied.

The above discussion shows that the proposition holds. □

Proposition 1 demonstrates that the 2-MAXSAT problem can be transformed, in polynomial time, to a problem to find a maximum number of conjunctions in a logic formula in *DNF*.

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

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

Under the truth assignment σ = $\{c_1$ = 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:

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

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, a 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 of variables c ($\in V$), we need to represent each D_i as a sequence of variables. 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 $c_j \in D_i$ (i.e., $c_j \in V$, but not appearing 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:

$$
\begin{aligned}
D = {} & D_1 \vee D_2 \vee D_3 \vee D_4 \vee D_5 \vee D_6 \\
= {} & (c_1 \wedge (c_2,*) \wedge (c_3,*) \wedge c_4 \wedge (c_5,*) \wedge (c_6,*)) \vee \\
& ((c_1,*) \wedge c_2 \wedge (c_3,*) \wedge \neg c_4 \wedge (c_5,*) \wedge (c_6,*)) \vee \\
& ((c_1,*) \wedge c_2 \wedge (c_3,*) \wedge (c_4,*) \wedge c_5 \wedge (c_6,*)) \vee \\
& ((c_1,*) \wedge (c_2,*) \wedge \neg c_3 \wedge (c_4,*) \wedge \neg c_5 \wedge (c_6,*)) \vee \\
& ((c_1,*) \wedge (c_2,*) \wedge c_3 \wedge (c_4,*) \wedge (c_5,*) \wedge c_6) \vee \\
& (\neg c_1 \wedge (c_2,*) \wedge (c_3,*) \wedge (c_4,*) \wedge (c_5,*) \wedge \neg c_6)
\end{aligned} \tag{3}
$$

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 elimilated. 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: $c_2 \rightarrow c_3 \rightarrow c_1 \rightarrow c_4 \rightarrow c_5 \rightarrow c_6$.

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 the variables in a sequence are ordered in terms of their appearance frequencies such that more frequent variables appear before less frequent ones. In addition, a start symbol # and an end symbol \$ are used as *sentinels* for technical convenience. In fact, any global ordering of variables works well (i.e., you can specify any global ordering of variables), 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 α 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 p going from # to \$, and through all the variables in D_1, there are four off-path edges (edges not on the main path), referred to as *spans* attached to p, corresponding to (c_2, *), (c_3, *), (c_5, *), and (c_6, *), respectively. Each span is represented by the subpath covered by it. For example, we will use the subpath $<v_0, v_1, v_2>$ (subpath 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 corresponding variable (then its value is set to 1).

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. It is because (c, *) indicates that c is optional, but going through a span from v to (c, *) makes c always selected. So, the intention of (c, *) is not well implemented.

For this reason, we introduce 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.

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

TABLE I: Conjunctions represented as sorted variable sequences with # and \$ used as sentinels.

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 frequencies 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

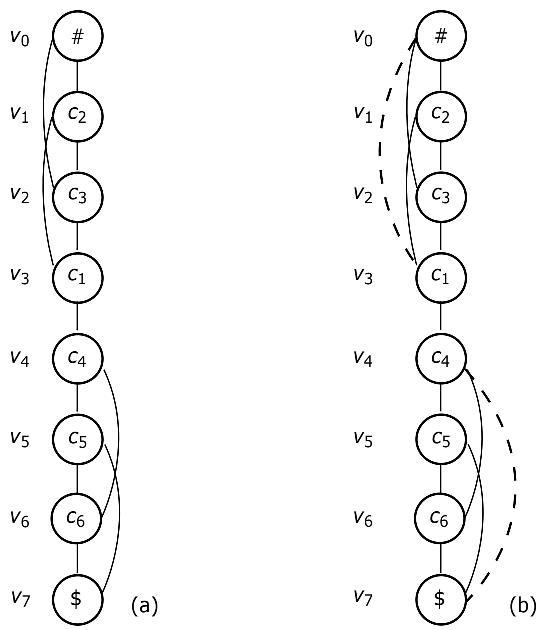

FIG. 1: A p-path and a $p*$-path.

'*'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 evaluates to true.*

Proof. (1) Corresponding to any truth assignment σ, 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 σ 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 variable 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. $\square$

B. Basic 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* [14], 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, $\ldots$, R_m so that each R_i (i = 1, $\ldots$, m) contains all those sequences with the same first variable name. The tries: $trie(R_1)$, $trie(R_2)$, $\ldots$, $trie(R_m)$ are constructed in the same way except that at the kth step, the splitting of sets is based on the kth 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.)

The advantage of tries is to cluster common parts of variable sequences together to avoid possible repeated checking. (Then, this is the main reason why we sort variable sequences according to their appearance frequencies.) Especially, this idea can also be applied to the variable subsequences (as will be seen in Section IV-B), over which some dynamical tries can be recursively constructed, leading to a polynomial time algorithm for solving the problem.

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.

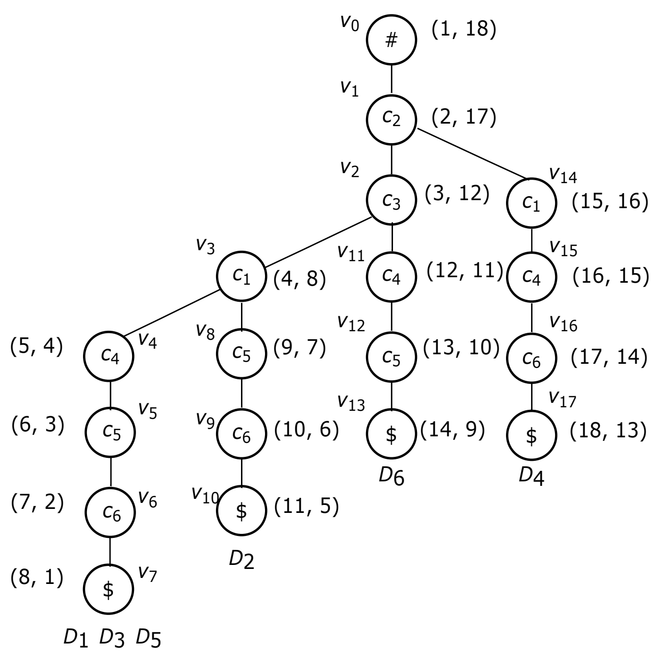

FIG. 2: A trie and tree encoding.

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 [13].

For instance, by checking the pair associated with v_2 against the pair 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 pair is (3, 12) and v_9's pair 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 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. In the same way, the labels for tree edges can also be determined. However, for simplicity, the tree edge labels are not shown in Fig. 3.

In addition, each p*-graph itself is considered 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. In fact, in a single p*-graph, the number of edges is bounded by O(n^2). Thus, a trie-like graph over m p*-graphs has at most O(n^2m) edges.

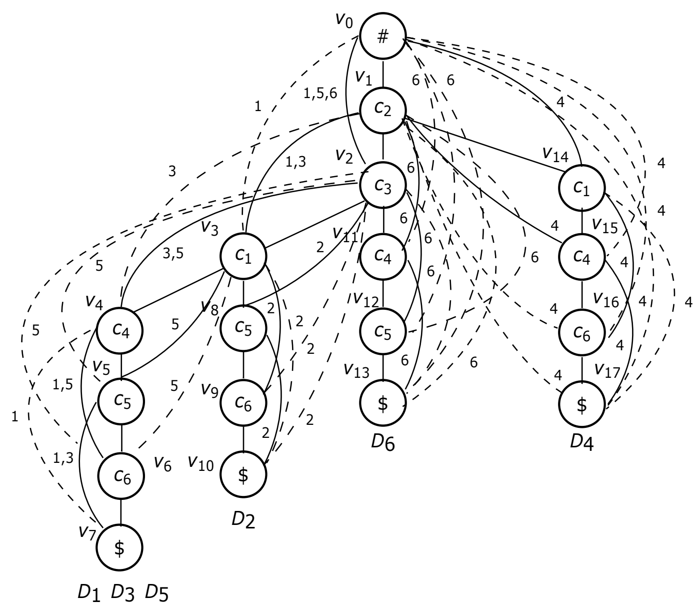

FIG. 3: A trie-like graph G.

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 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 σ:

σ = $\{$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 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 σ: 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 σ, 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. Initially, $S = \emptyset$.

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 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' 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.*

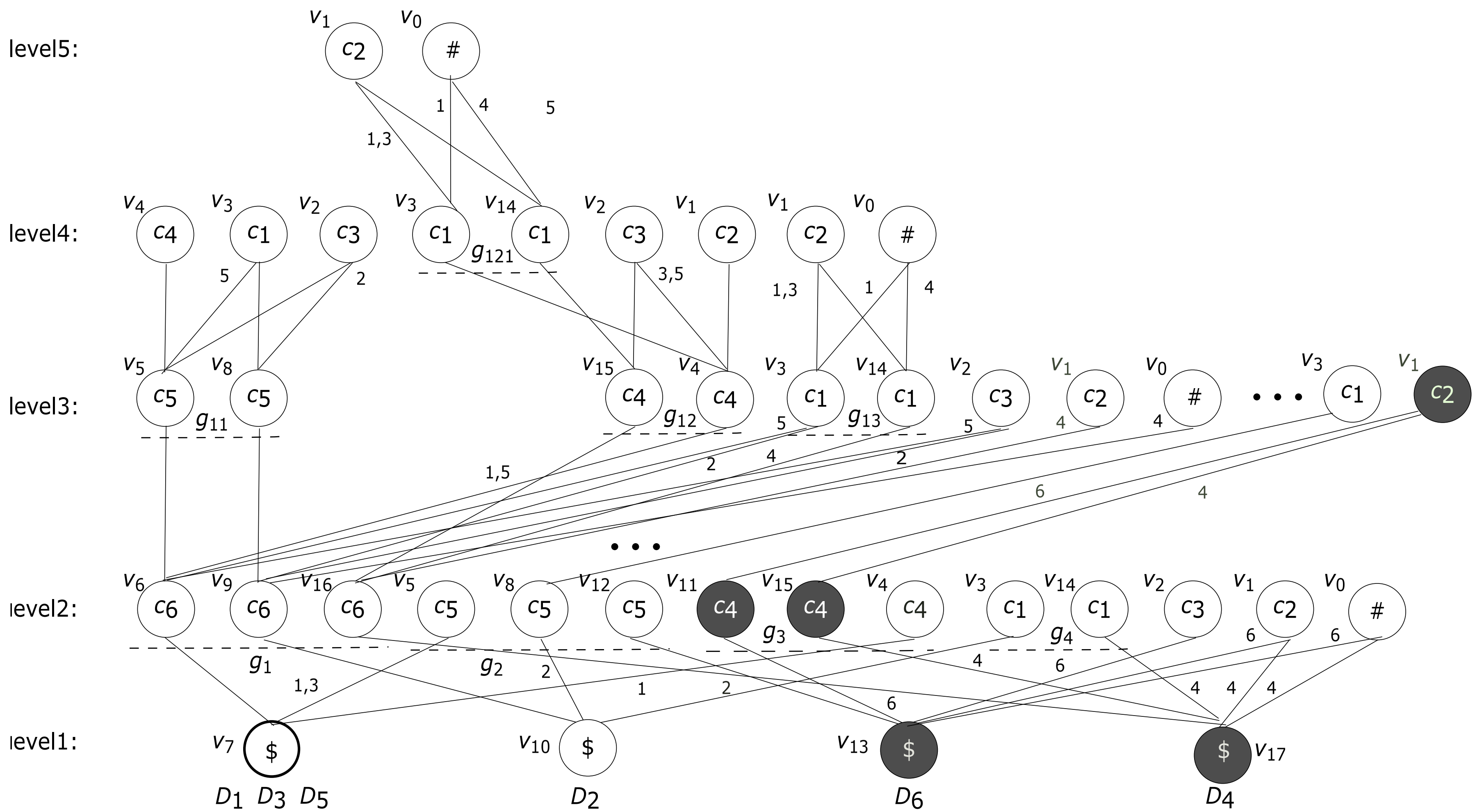

FIG. 4: Illustration for the layered representation G' of G.

Algorithm 1: *SEARCH*(G)

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

G' :={all leaf nodes of G}; g := {all leaf nodes of G};
push(S, g); (* find the layered graph G' of G *)
while S *is not empty* **do**
  g := pop(S);
  find the parents of each node in g; add them to G';
  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;
  for *each* $j \in \{1, ..., k\}$ **do**
    if $|g_j| > 1$ **then**
      push(S, g_j);
return $findSubset(G')$;

Algorithm 2: *findSubset*(G')

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

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

Proof. By the execution of $SEARCH(G)$, we will first generate the layered reprentation 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. In the worst case, the number of nodes in G' can be exponential. However, such a redundancy can be effectively removed by interleaving the process of *SEARCH* and *findSubset* in some way. To this end, 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 $v_3 \to u$, $v_3 \to v \in G$.

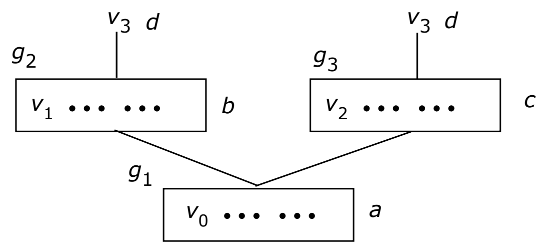

FIG. 5: A possible part in a layered graph G'.

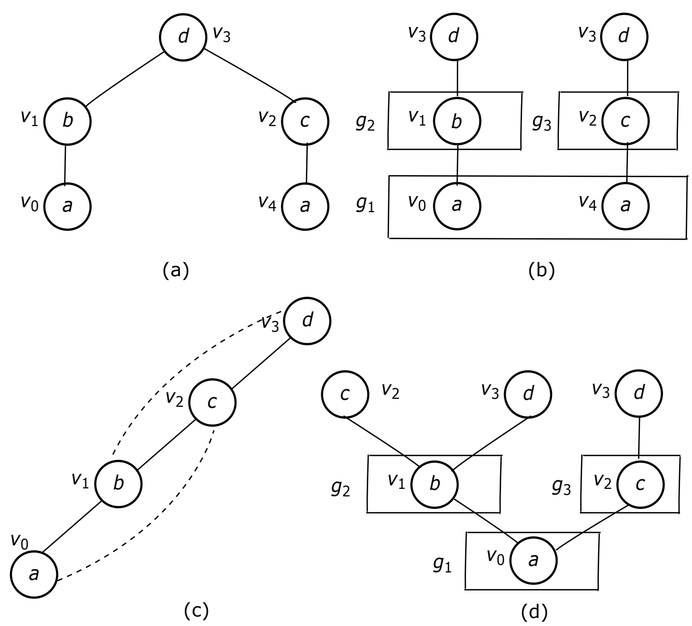

FIG. 6: Two reasons for repeated appearances of nodes at a level in G'.

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

Case 1 and Case 2 can be efficiently differentiated from each other by using the tree encoding, as illustrated 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*( ) immediately 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 impact neither the correctness of the algorithm nor 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.

Definition 3. (reachable subsets through spans) Let v be a repeated node of Case 1. Let u be a node on the tree path (in T) from *root* to v (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 $RS_s^{v,u}[c]$, where s is a set containing all the labels associated with the corresponding spans.

For $RS_s^{v,u}[c]$, node u is called its *anchor* node.

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 with respect to v_3:

- $RS_{\{2,5\}}^{v_3,v_2}[c_5]$ = $\{v_5, v_8\}$,
- $RS_{\{2,5\}}^{v_3,v_2}[c_6]$ = $\{v_6, v_9\}$.

We have $RS_{\{2,5\}}^{v_3,v_2}[c_5]$ 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 $RS_{\{2,5\}}^{v_3,v_2}[c_6]$ 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'.

Hence, v_2 is not only the anchor node of $\{v_5, v_8\}$, but also the anchor node of $\{v_6, v_9\}$.

In general, we are interested only in those *RS*s with |*RS*| $\geq$ 2 since any *RS* with $|RS|$ = 1 only leads us to a leaf node in T, and no larger subsets of conjunctions can be found. So, in the subsequent discussion, by an *RS*, we mean an *RS* with |*RS*| $\geq$ 2.

The definition of this concept for a repeated node v of Case 1 itself 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:

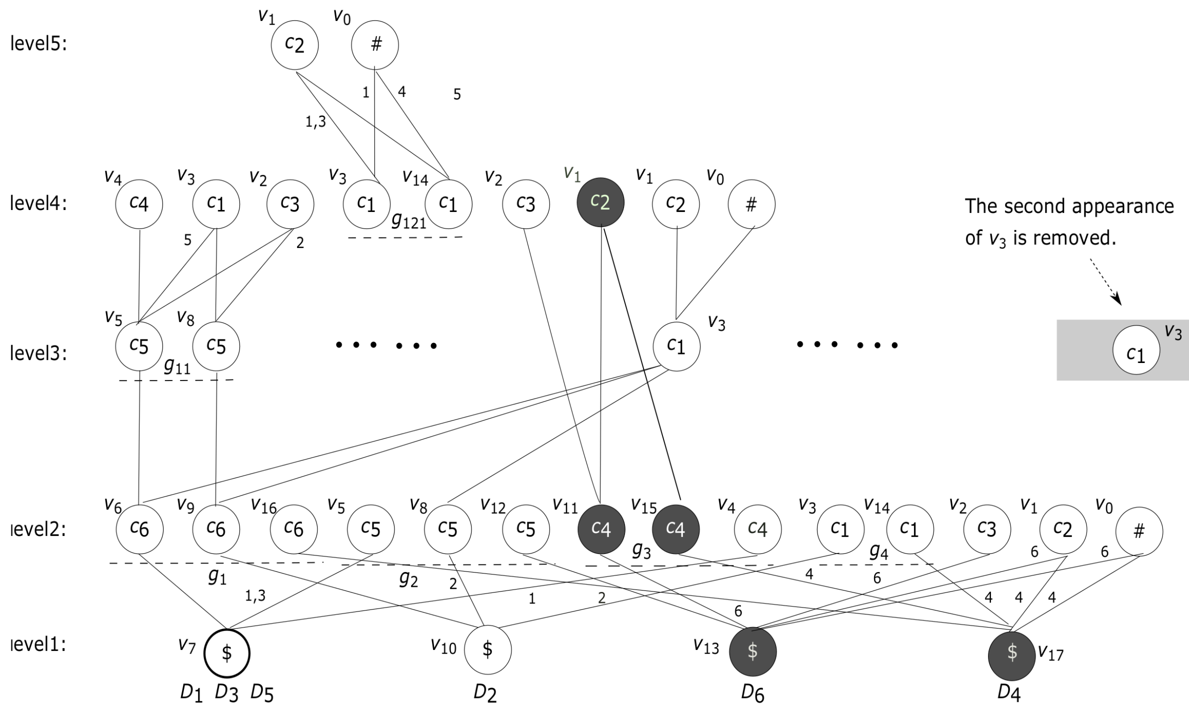

FIG. 7: Illustration for removing repeated nodes.

- $RS^{v_3,v_3}_{\{2,5\}}[c_5]$ = $\{v_5, v_8\}$,
- $RS^{v_3,v_3}_{\{2,5\}}[c_6]$ = $\{v_6, v_9\}$,

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'. Here, we notice that the label for the tree edge $v_3 \rightarrow v_8$ is 2 since this tree edge belongs to D_2.

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

Definition 4. (upper boundaries) Let v be a branching node. Let v_1, v_2, ..., v_k be all the nodes on the path from *root* to v. An upper boundary (denoted as *upBounds*) with respect to v is a largest subset of nodes $\{u_1, u_2, ..., u_f\}$ with the following properties satisfied:

1) Each u_g ($1 \leq g \leq f$) appears in some $RS^{v,v_i}_s[c]$ ($1 \leq i \leq k$), where c is a label.
2) For any two nodes u_g, $u_{g'}$ ($g \neq g'$), they are not related by the ancestor/descendant relationship.

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 in a way similar to the construction of G over all the initial

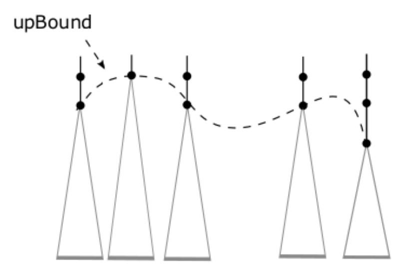

FIG. 8: Illustration for upBounds.

p*-graphs. Here, we remark that v_4 is not included since it is not invlved in any *RS* with respect to v_3 with $|RS| \geq 2$. In fact, 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 general, going through a span with the corresponding $|RS|$ = 1, we cannot get any new answers.

Mainly, the following operations will be carried out when meeting 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.*

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 construct a trie-like graph as shown in Fig. 9(b). It is in fact a single path, where v_{5-8} stands for the merging of v_5 and v_8, v_{6-9} for the merging of v_6 and v_9, and v_{7-10} for the merging of v_7 and v_{10}.

In addition, for technical convenience, we will add the corresponding repeated node (v_3) to the trie as a virtual root, and $v_3 \xrightarrow{2,5} v_{5-8}$ as a virtual edge. Here, the virtual root, as well as the virtual edge, is added to keep the connection of the trie-like subgraph to the tree path from the root to the repeated node in T, which will greatly facilitate the trace of truth assignments for the corresponding satisfied conjunctions. In general, the label s for a virtual edge $u \to v$ is created as follows.

Assume that v represents a merging of nodes v_1, ..., v_k ($k \geq 2$). Denote by s_i the label associated with the span to v_i ($1 \leq i \leq k$). Then, s is set to be $s_1 \cup ... \cup s_k$.

For example, searching the path from v_{7-10} to v_{5-8} in Fig. 9(c) bottom-up, and then going through the virtual node v_3 to find the path from the corresponding anchor node v_2 to v_0 in T (see Fig. 3), we will find a truth assignment $\{c_1 = 0, c_2 = 1, c_3 = 1, c_4 = 0, c_5 = 1, c_6 = 1\}$, satisfying $\{D_2, D_5\}$.

Finally, we notice that the subset associated with the unique leaf node of the path is $\{D_2, D_5\}$, instead of $\{D_1, D_2, D_3, D_5\}$. It is because the number associated with the virtual edge $v_2 \to v_{5-8}$ is $\{2, 5\}$, by which D_1 and D_3 are filtered out.

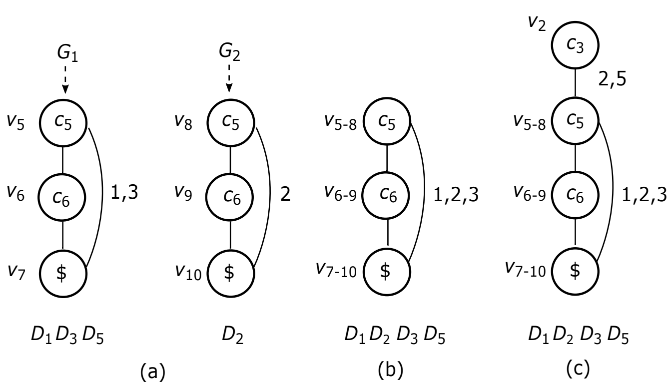

FIG. 9: Illustration for construction of trie-like subgraphs.

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 (largest) 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. To avoid this kind of useless work, we can simply perform 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 a satisfiable subset of conjunctions already obtained, the recursive call of the algorithm should not be conducted.

In terms of the above discussion, we come up with a recursive algorithm (Algorithm 3, also called *SEARCH*( )) shown below.

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; $D' := \{v\} \cup D'$; 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 $findSubset(G')$;

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

In this algorithm, we also notice that $D' := \{v\} \cup D'$ (in line 8) is used to add the repeated node of Case 1 as a virtual node. But it a simplified representation of the operation, by which we add not only v, but also the corresponding virtual edges to D'.

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, each subset of conjunctions associated with a leaf node v is satisfiable by a certain truth assignment represented by the corresponding main path (from *root* to v). Especially, $\{D_1, D_2, D_5\}$ associated with v_{10} (see Fig. 2) is a largest subset of conjunctions, 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.
- 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 will make a recursive call of *SEARCH*( ) applied to a trie-like subgraph constructed over two subgraphs in $G[v_2]$ (respectively rooted at v_3 and v_{11}), as shown in Fig. 10.

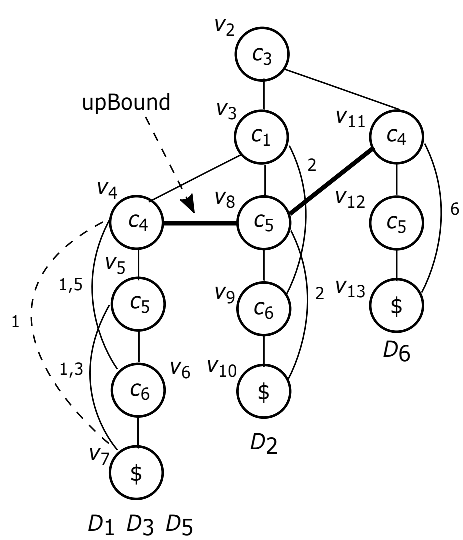

FIG. 10: Two subgraphs in $G[v_2]$ and an upBound.

First, with respect 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

- $RS^{v_2,v_1}_{\{3,6\}}[c_4] = \{v_4, v_{11}\}$,

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:

- $RS^{v_2,v_2}_{\{3,5,6\}}[c_4] = \{v_4, v_{11}\}$,

- $RS^{v_2,v_2}_{\{2,5,6\}}[c_5] = \{v_5, v_8, v_{12}\}$,

- $RS^{v_2,v_2}_{\{3,5\}}[c_6] = \{v_6, v_9\}$,

- $RS^{v_2,v_2}_{\{2,6\}}[\$] = \{v_{10}, v_{13}\}$,

respectively due to four groups of spans shown below (see Fig. 3):

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

Then, in terms of these 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). Finally, we will determine over what subgraphs a trie-like graph should be constructed, over which the algorithm will be recursively executed.

In Fig. 11, 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}. Again, the repeated node v_2 is involved as the virtual root. (For a better understanding, the spans from v_2 are also shown here.)

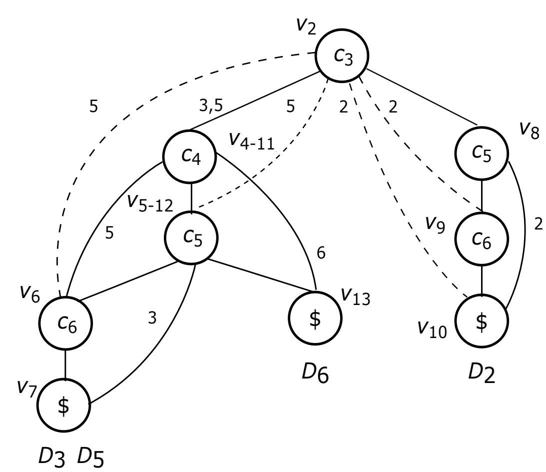

FIG. 11: A trie-like graph.

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. 12, we can see a repeated nodes of Case 1: v_{5-12}. Then, a recursive call of the algorithm will be conducted, generating an upBound is $\{v_7, v_{13}\}$, as shown in Fig. 13(a). Similar to the above discussion, we will construct the corresponding trie-like subgraph, which is just a single node as shown in Fig. 13(b). Adding the corresponding virtual node v_{5-12}, and virtual edge $v_{5-12} \xrightarrow{3,6} v_{7-13}$, we will get a path as shown in Fig. 13(c), by which we will find a largest subset of conjunctions $\{D_3, D_6\}$, satifiable by a certain truth assignment. But 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 virtual edge from v_{5-12} to v_7 (in Fig. 13(c)) is labeled with $\{3, 6\}$ and D_5 should be removed.

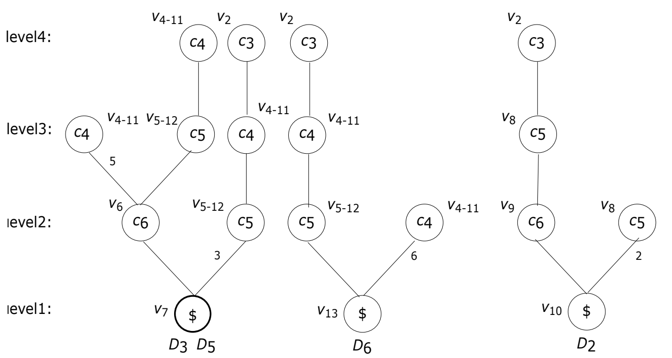

FIG. 12: A naive bottom-up search of G.

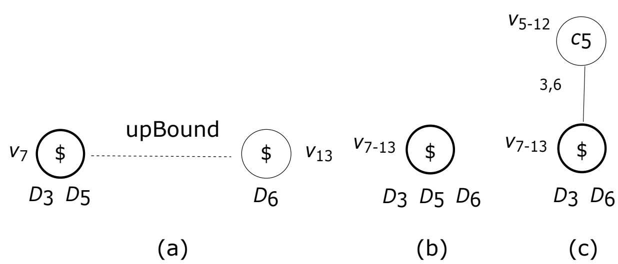

FIG. 13: Illustration for trie-like subgraph construction.

When we encounter the second repeated node v_2 at level 4 in Fig. 12, we will create another set of *RS*s shown below:

- $RS^{v_2,v_1}_{\{3,6\}}$ = $\{v_{4-11}\}$
(due to the span $v_1 \xrightarrow{3,6} v_{4-11}$),

- $RS^{v_2,v_2}_{\{5,2\}}[c_5]$ = $\{v_{5-12}, v_8\}$
(due to the span $v_2 \xrightarrow{5} v_{5-12}$ and the tree edge $v_2 \rightarrow v_8$),

- $RS^{v_2 v_2}_{\{5,2\}}[c_6]$ = $\{v_6, v_9\}$
(due to the spans $v_2 \xrightarrow{5} v_6$ and $v_2 \xrightarrow{2} v_9$).

Since $|RS^{v_2,v_1}_{\{3,6\}}|$ = 1, it will not be further considered in the subsequent computation.

In terms of $RS^{v_2,v_2}_{\{5,2\}}[c_5]$ and $RS^{v_2,v_2}_{\{5,2\}}[c_6]$, we will construct an upBound $\{v_{5-12}, v_8\}$, and create a trie-like graph as shown in Fig. 14(a). The only repeated node in this graph is v_{5-12-8}. With respect to v_{5-12-8}, two *RS*s can be figured out in terms of the spans to the nodes in this subgraph (see Fig. 14(b)):

- $RS^{v_2,v_2}_{\{2\}}[c_6]$ = $\{v_{6-9}\}$
(due to the span $v_2 \xrightarrow{5,2} v_{5-12}$),

- $RS^{v_2,v_2}_{\{5,2\}}[\$]$ = $\{v_{7-10}\}$
(due to the span $v_2 \xrightarrow{2} v_{7-10}$).

Both of these *RS*s are of size 1. Therefore, by the corresponding recursive call of *SEARCH*( ), no trie-like subgraph will be generated.

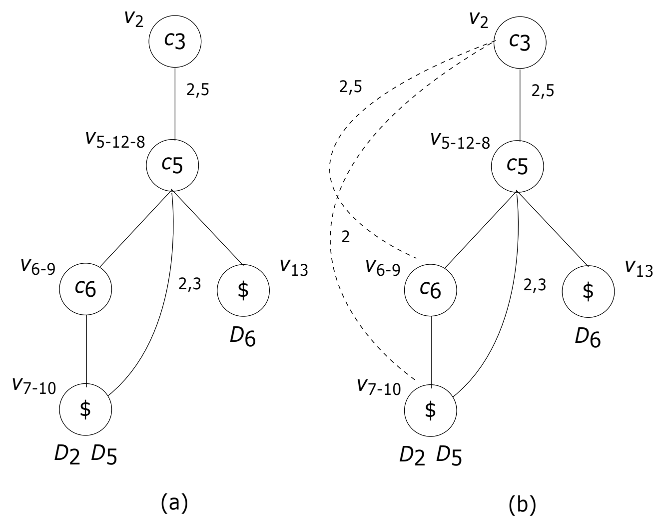

FIG. 14: Illustration for recursive execution of algorithm.

Searching the graph shown in Fig. 14(b) bottom up, we will find a new subset of conjunctions $\{D_2, D_5\}$ satisfiable by a certain truth assignment.

After we have returned back along the chain of the recursive calls described above, we will continually explore G' and encounter the last repeated node v_1 of Case 1 in G, which will be handled in a way similar to v_3 and v_2 (in Fig. 3).

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

Proposition 3. *Let G be a trie-like graph established over a logic formula in DNF. Applying SEARCH( ) to G, 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.

We then need to show that any subset of conjunctions satisfiable by a certain truth assignment can be found by the algorithm. For this purpose, consider a subset of conjunctions $D' = \{D_1, ..., D_r\}$ $(r > 1)$ which can be satisfied by a truth assignment represented by a path P. We will prove by induction on the number n_s of spans on P that our algorithm is able to find P.

Basic step. When n_s = 0, P must be a tree path in T and the claim holds. When n_s = 1, the unique span on P must cover a repeated node w of Case 1 in G. Let $u \xrightarrow{s} v$ be such a span. Denote by P' the tree path from *root* to u in T. Then, by a recursive call of *SEARCH*( ) over the trie-like subgraph constructed with respect to w we can find a sub-path P''; and P must be equal to the concantenation of P', $u \xrightarrow{s} v$, and P''.

Induction step. Assume that when n_s = k, the algorithm can find P.

Now, assume that P contains k + 1 spans s_1, s_2, ..., s_k, s_{k+1}. They must corresponds to a chain of k + 1 nested recursive calls of *SEARCH*( ). Denote by G_i the trie-like subgraph created by the (i - 1)th recursive call, where G_0 = G. Let $u \xrightarrow{s} v$ be the first span on P. Denote by P' the sub-path from the *root* of T to u, and by P'' the sub-path of P from v to the last node of P. Denote by $D_j \backslash P'$ the conjunction obtained by removing variables on P' from D_j (j = 1, ..., r). Let D'' = { $D_1 \backslash P'$, ..., $D_r \backslash P'$}. Then, the truth assignment represented by P'' satisfies D''. According to the induction hypothesis, P'' can be found by executing *SEARCH*( ). Therefore, P can also be found by *SEARCH*( ). To see this, observe the first recursive call of *SEARCH*( ) made when we meet the first repeated node of Case 1 in G', by which we will find P'' satisfying D''. Then, the concantenation of P' and P'' definitely satisfies D'. This completes the proof. $\square$

However, during the execution of *SEARCH*( ), for different repeated nodes of Case 1, the same *RS*s can be repeatedly produced, leading to some kinds of redundancy. See Fig. 15(a) for illustration.

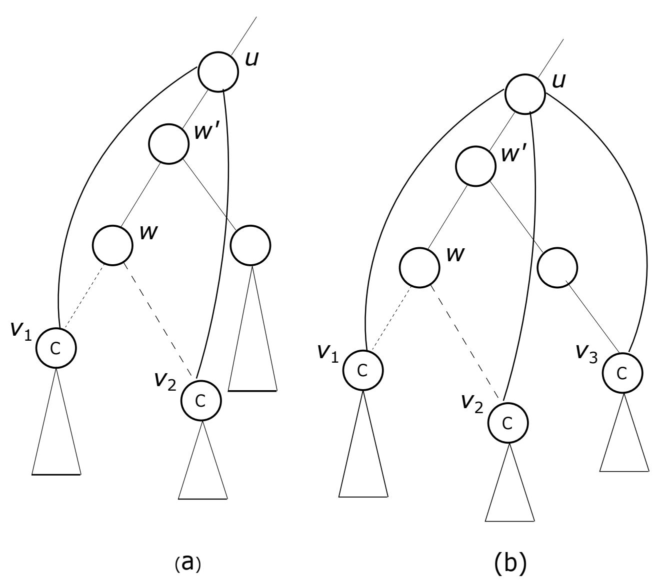

FIG. 15: Illustration for redundancy.

In this figure, special attention should be paid to w and w'. They must be two repeated nodes of Case 1 when we explore G'. With respect to w and w', their ancestor u will have two identical *RS*s:

$$RS_s^{w,u}[C] = RS_s^{w',u}[C] = \{v_1, v_2\}.$$

Thus, during the execution of *SEARCH*( ), the same trie-like subgraph will be created two times: one is for $RS_s^{w,u}$[C] and another is for $RS_s^{w',u}$[C], but with the same result to be produced.

Fortunately, this kind of redundancy can be simply removed in two ways.

In the first way, we can examine, by each recursive call, whether the input subgraph has been checked before. If it is the case, the corresponding recursive call will be suppressed.

In the second way, we create *RS*s only for those nodes appearing on part of a tree path between the current repeated node and the lowest ancestor repeated node in T. Even though we may lose some answers in this way. one of the maximum satisfiable subsets of conjunctions can always be found. See Fig. 15(b) for illustration. In this case, the *RS* of u with respect to w is different from the *RS* with respect to w'. However, when checking w, $RS_s^{w,u}$[C] will not be computed since u is beyond the segment between w and w'. Therefore, the corresponding result will not be generated. However, $RS_s^{w',u}$[C] must cover $RS_s^{w,u}$[C], revealing a larger (or same-sized) subset of conjunctions which can be satisfied by a certain truth assignment.

V. Time complexity analysis

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

The first part τ_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, τ_1 = O(nm).

The second part τ_2 is the time for constructing a trie-like graph G for D. This part of time can be further partitioned into three portions.

- τ_{21}: The time for sorting variable sequences for D_i's. It is obviously bounded by O($nm\log_2 m$).
- τ_{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).
- τ_{23}: The time for merging all p*-graphs to form a trie-like graph G. This part is also bounded by O(nm^2).

The third part τ_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 be 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 τ_3 can be characterized by the following recurrence based on an observation that for each repeated node of Case 1 a recursive call of the algorithm will be performed:

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

Here, in the above recursive equation, O(l^2m) 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 cannot be larger than O(lm).

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

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

Solving this inequality, we will get

$$\begin{aligned} \Gamma(l) &\leq d \cdot log_d\, l \cdot \Gamma(\tfrac{l}{d}) + O(l^2 m) \\ &\leq d^2 (log_d\, l)(\, log_d \tfrac{l}{d}) \Gamma(\tfrac{l}{d^2}) + \ (log_d\, l)\ l^2 m + \ l^2 m \\ &\leq \dots \dots \\ &\leq d^{\lceil log_d^l \rceil} (log_d\, l)\ (log_d(\tfrac{l}{d}))\ \dots\ (\, log_d \tfrac{l}{d^{\lceil log_d^l \rceil}}) \\ &\quad + l^2 m ((log_d\, l)(log_d(\tfrac{l}{d}))\ \dots\ (log_d \tfrac{l}{d^{\lceil log_d^l \rceil}}) + \dots + log_d\, l + 1) \\ &\leq O(l(log_d\, l)^{log_d\, l} + \ O(l^2 m (log_d\, l)^{log_d\, l}) \\ &\sim O(l^2 m\ (log_d\, l)^{log_d\, l}). \end{aligned} \tag{6}$$

Thus, the value for τ_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*

$$\begin{aligned} \Sigma_{i=1}^{3} \tau_i &= O(nm) + (O(nm\ log_2\ m) + 2 \times O(nm^2)) \\ &\quad + O(l^2 m\ (log_d\, l)^{log_d\, l}) \\ &= O(n^2 m^3 (log_d\, nm)^{log_d\, nm}). \end{aligned} \tag{7}$$

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 we have d > 1 in (7).

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.

First, we notice that in all the generated trie-like subgraphs, the number of all the repeated nodes of Case 1 is bounded by O(nm). But each repeated node may be involved in at most O(m) recursive calls (see the analysis given below) 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^2m^4).

We need to make clear that each repeated node of Case 1 can be involved at most in O(m) recursive calls. For this, we have the following analysis.

Consider a trie-like graph G shown in Fig. 16(a), in which w is a repeated node of Case 1. With respect to w, we will have the following three *RS*s:

- *RS*$_{s'}^{w,u}$[C] = $\{v_1,\ v_2\}$,

- *RS*$_{s''}^{w,u}$[D] = $\{v_3,\ v_5,\ v_6\}$,

- *RS*$_{s'''}^{w,u}$[E] = $\{v_4,\ v_7,\ v_8,\ v_9\}$,

where s', s'' and s''' are three label sets for the three *RS*s, respectively.

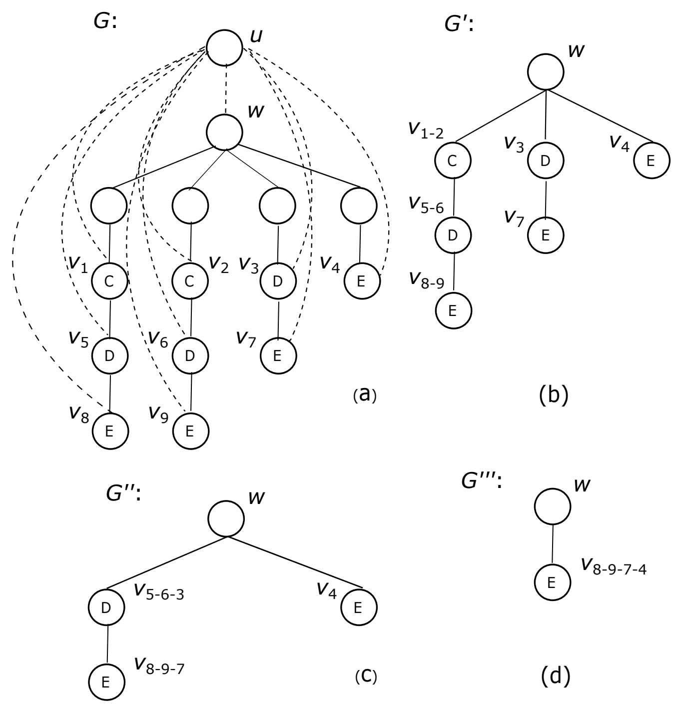

FIG. 16: Illustration forrecursive construction of trie-like subgraphs.

According to these *RS*s, we will construct a trie-like subgraph G' as shown in Fig. 16(b) and a recursive call of *SEARCH*( ) will be carried out. It is the first recursive call, in which w is involved. During this recursive execution of *SEARCH*( ), w will then be involved in a second recursive call, but on a smaller trie-like subgraph G'', whose height is one level lower than G' (see Fig. 16(c)). During the second recursive call, w will be involved in a third recursive call. For this time, the height of the corresponding trie-like subgraph is further reduced as demonstrated in Fig. 16(d).

Together with the method discussed in the previous section to avoid multiple generation of a same *RS*, the above analysis shows that any repeated node of Case 1 can be involved in at most m recursive calls of *SEARCH*( ).

Proposition 5. *Let G be a trie-like graph and w be a repeated node of Cae 1 in the corresponding layered graph. Then, w can be involved in at most m recursive calls of SEARCH( ) in the whole working process.*

Proof. Let $\{v_1, v_2, ..., v_k\}$ ($k \geq 2$) be a largest group of nodes appearing on the upBound L with respect to w with the following three properties:

- Each v_i (i = 1. ..., k) has no ancestor on L.
- $l(v_1)$ = $l(v_2)$ = ... = $l(v_k)$.
- There is not any other node u with $l(u)$ = $l(v_1)$, which is a descendant of any node on L.

Then, in the trie-like subgraph G' constructed for L, all the nodes in this group will be merged into a single node. The same claim applys to any other largest group of nodes on L satisfying the above three properties. Thus, in a next recursive call of *SEARCH*( ) involving w, the trie-like subgraph G'' to be constructed must be at least one level lower than G' since when constructing a trie-like subgraph any *RS* with $|RS|$ = 1 will not be considered. Because the height of G is bounded by m and any trie-like subgraph is constructed only once (using the method discussed in the previous section to avoid multiple generation of a same *RS*), the proposition holds. □

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^2m^4), 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 G. Searching G bottom up in a recursive way, we can find the answer efficiently.

References

[1] J. Argelich, et. al., *MinSAT versus MaxSAT for Optimization Problems,* International Conference on Principles and Practice of Constraint Programming, 2013, pp. 133-142.

[2] Y. Chen, *The 2-MAXSAT Problem Can Be Solved in Polynomial Time,* in Proc. CSCI2022, IEEE, Dec. 14-16, 2022, Las Vegas, USA, pp. 473-480.

[3] 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.

[4] 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.

[5] C. Dumitrescu, *An algorithm for MAX2SAT,* International Journal of Scientific and Research Publications, Volume 6, Issue 12, December 2016.

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

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

[8] R. Impagliazzo and R. Paturi, *On the complexity of k-sat. J. Comput.*, Syst. Sci., 62(2):367–375, 2001.

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

[10] E. Kemppainen, *Imcomplete Maxsat Solving by Linear Programming Relaxation and Rounding*, Master thesis, University of Helsinki, 2020.

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

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

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

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

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

[16] 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.

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

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

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

[20] Y. Shang, *Resilient consensus in multi-agent systems with state constraints*, Automatica, Vol. 122, Dec., 2001, 109288.

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

[22] M. Xiao, *An Exact MaxSAT Algorithm: Further Observations and Further Improvements*, Proc. of the Thirty-First International Joint Conference on Artificial Intelligence (IJCAI-22).

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