Solving the 2-MAXSAT Problem in Polynomial Time: a Proof of *P* = *NP*

YANGJUN CHEN, Department Applied Computer Science, The University of Winnpeg, Canada

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*-complete 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. Thus, the paper in fact provides a proof of *NP* = *P*.

CCS Concepts: • **Theory of computation → Propotional Satisfiability problem**.

Additional Key Words and Phrases: satisfiability problem, maximum satisfiability problem, NP-hard, NP-complete, conjunctive normal form, disjunctive normal form.

ACM Reference Format:
Yangjun Chen. . Solving the 2-MAXSAT Problem in Polynomial Time: a Proof of *P* = *NP*. xx, x (January), 23 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 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 also *NP*-complete 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

The article is a modification and extension of a conference papers: [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.

This work is supported by NSERC, Canada, 239074-01 (242523).

Author's address: Yangjun Chen, y.chen@uwinnipeg.ca, Department Applied Computer Science, The University of Winnpeg, 515 Portage Ave., Winnipeg, Manitoba, Canada, R3B 2E9 .

ACM XXXX-XXXX//1-ART
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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 2, 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 3. Next, in Section 4, 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 5.

2 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 [7], the 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 $C_j = c_{j1} \vee c_{j2}$ ($j = 1, ..., p$), 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_{j1} and c_{j2} 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 $\sigma = \{c_1 = 1, c_2 = 1, c_3 = 1\}$, 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:

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

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.

3 ALGORITHM DESCRIPTION

In this section, we discuss our algorithm. First, we present the main idea in Section 3.1. Then, in Section 3.2, a recursive algorithm for solving the problem is described in great detail. The running time of the algorithm will be analyzed in the next section.

3.1 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 (referred to 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 $c_j \in D_i$ (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:

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

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 1 for illustration.

TABLE 1. 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	$(c_2, {}^*).(c_3, {}^*).(c_4, {}^*).(c_5, {}^*)$	$\#.(c_2, {}^*).(c_3, {}^*).(c_4, {}^*).(c_5, {}^*).\$$

First, we pay attention to the variable sequence for D_2 (the second sequence in the second column of Table 1), 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 2, in which we show the appearance frequencies of all the variables in the above D.

TABLE 2. 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

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: $c_2 \longrightarrow c_3 \longrightarrow c_1 \longrightarrow c_4 \longrightarrow c_5 \longrightarrow c_6$.

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 1, 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 *sentinals* 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 $\alpha = c_0c_1 \ldots c_kc_{k+1}$ 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 going 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 the main path, 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 notation (c, *), which is used to indicate that c is optional, is not correctly implemented.

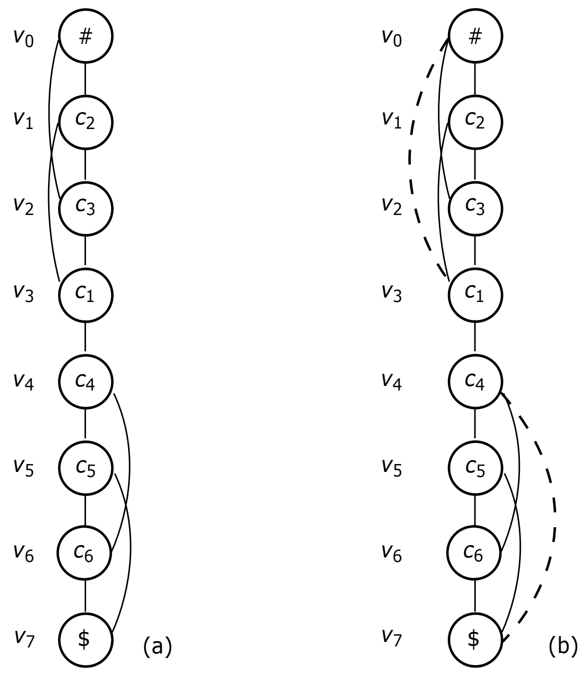

FIG. 1. A p-path and a $p*$-path.

For this reason, we introduce another concept, p^*-graphs, described as below.

Let s_1 = $<v_1, ..., v_k>$ and s_2 = $<u_1, ..., u_l>$ be two spans attached onto 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 '*'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, any path from # to \$ in P^* represents a truth assignment, under which D_i evaluate 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 evaluates 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. □

3.2 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 r (possibly empty) subsets $R_1, R_2, \ldots, R_r$ so that each R_i ($i = 1, \ldots, r$) contains all those sequences with the same first variable name. The tries: $trie(R_1)$, $trie(R_2)$, $\ldots$, $trie(R_r)$ 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 given in the third column of Table 1. 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. For example, the subset $\{D_1, D_3, D_5\}$ associated with v_7 is satisfiable under the truth assignment represented by the path from v_0 to v_7. Such a path is also called a tree 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.) More importantly, this idea can also be applied to the variable subsequences (as will be seen later), over which some dynamical tries can be recursively constructed, leading to a polynomial-time algorithm for solving the problem.

Each node v in the trie stands for a variable c, referred to as the label of v and denodeted as $l(v) = c$; and each edge e is referred to as a tree edge, labeled with a set of integers representing all the variable sequences going through e, denoted as $s(e)$. For example, $s(v_0, v_1) = \{1, 2, 3, 4, 5, 6\}$. It is because all the variable sequences given in Table 1 need to pass through this edge to reach their respective leaf nodes. In the same way, you can check all the other labels associated with tree edges.

In regard to the tree paths, we have the following lemma.

Lemma 2. *Let T be a trie created over all the variable sequences in D. Let $p = v_0 \xrightarrow{s_1} v_1 \ldots \xrightarrow{s_k} v_k$ be a root-to-leaf path in T. Let D' be the subset of conjunctions associated with v_k. Then, $R = s_1 \cap \ldots \cap s_k \cap D'$ is satisfiable by the truth assignment represented by p.*

Finally, we will associate each node v in the trie T with 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.

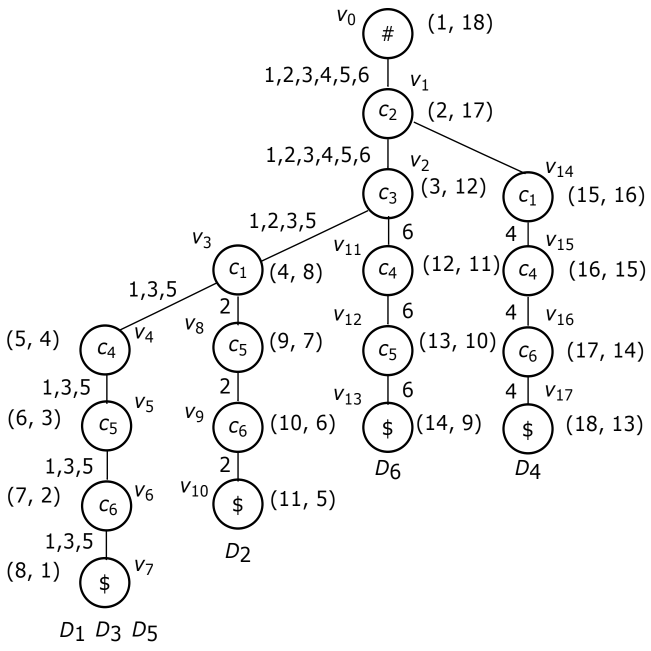

Fig. 2. A trie and tree encoding.

- 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 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. Specifically, 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 given in Table 1, 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 Fig. 3, however, the labels for all tree edges are not shown for a clear illustration.

In addition, each p^*-graph itself is considered to be a simple trie-like graph.

Concerning the paths in a trie-like graph, we have a lemma similar to Lemma 2.

Lemma 3. *Let G be a trie-like graph created over all the variable sequences in D. Let $p = v_0 \xrightarrow{s_1} v_1 \ldots \xrightarrow{s_k} v_k$ be a root-to-leaf path in G, where some edges can be spans. Let D' be the subset of conjunctions associated with v_k. Then, $R = s_1 \cap \ldots \cap s_k \cap D'$ is satisfiable by the truth assignment represented by p.*

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(m^2). Thus, a trie-like graph over n p^*-graphs has at most O(nm^2) edges.

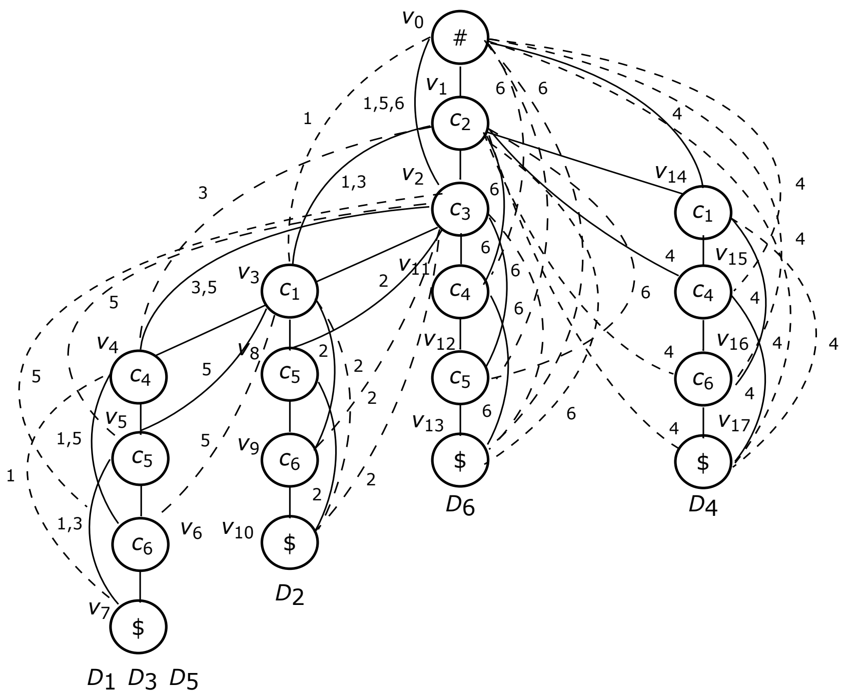

FIG. 3. A trie-like graph G.

In a next step, we will search G bottom-up level by level to seek all the possible largest subsets of conjunctions which can be satisfied by a certain truth assignment.

First of all, we call each node in T with more than one child a *branching* node. For instance, node v_3 with two children v_4 and v_8 in G shown in Fig. 3 is a branching node. For the same reason, v_2 and v_1 are another two branching nodes. Note that v_0 is not a branching node since it has only one child in T (although it has more than one child in G.)

Around the branching node, we have two very important concepts defined below.

Definition 3. (reachable subsets through spans) Let v be a branching node. 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 $G[v]$ (subgraph of G rooted at 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 also called its *anchor* node while any node in $RS_s^{v,u}[c]$ is called a *reachable* node of u.

For instance, for node v_2 in Fig. 3, which is on the tree path from *root* to v_3 (a branching node), we have two RSs 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. In fact, going through a span with the corresponding $|RS| = 1$, we cannot get any new answers. So, in the subsequent discussion, by an *RS*, we mean an *RS* with $|RS| \geq 2$.

The definition of this concept for a branching node v 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:

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

respectively due to span $v_3 \xrightarrow{5} v_5$ and tree edge $v_3 \longrightarrow v_8$ going out of v_3 with $l(v_6) = l(v_8) =$ 'c_5'; and two spans $v_3 \xrightarrow{5} v_6$ 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 $v_3 \longrightarrow v_8$ is 2 since this tree edge belongs to D_2 (see Fig. 2).

Concerning *RS*s, we have the following lemma, which is important for the construction of trie-like subgraphs.

LEMMA 4. *Let v be a branching node in G. Let u be an ancestor of u' on the tree path from root to v. If both $RS_s^{v,u}[c]$ and $RS_s^{v,u'}[c]$ exist for a certain label c, then we have $RS_s^{v,u}[c] \subseteq RS_s^{v,u'}[c]$.*

PROOF. Let $P^* = p \cup S^*$ be a p^*-graph merged into G. Assume that in P^* we have a span from a node u to some other node w. Then, for any descendant u' of u on the subpath from the child of u to the grandparent of w, we must have a span from u' to w due to the transitivity of spans. Assume that $l(w) = c$. We can immediately see that $RS_s^{v,u}[c] \subseteq RS_s^{v,u'}[c]$. □

If $RS_s^{v,u}[c] \subset RS_s^{v,u'}[c]$, we say, $RS_s^{v,u'}[c]$ is larger than $RS_s^{v,u}[c]$.

Based on the concept of reachable subsets through spans, we are able to define another more important concept, upper boundaries, 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 $\{u_1, u_2, ..., u_f\}$ $(f > 1)$ with the following properties satisfied:

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

Fig. 4 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 (a branching node), based on which 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 p^*-graphs (which then hints a recursive process to do the task). 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 the first step.

Mainly, the following operations will be carried out when encountering a branching node v.

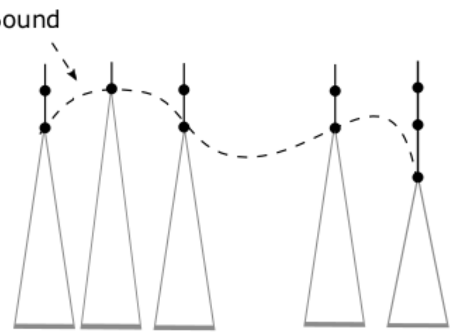

Fig. 4. Illustration for upBounds.

- Calculate all *RS*s with respect v.
- Calculate the upBound in terms of *RS*s.
- Make a recursive call of the algorithm on a subgraph which is constructed over all the p^*-subgraphs each rooted at a node on the corresponding upBound.

See the following example for illustration.

Example 1. *When checking the branching 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: $RS^{v_3,v_2}_{\{2,5\}}[c_5] = \{v_5, v_8\}$, $RS^{v_3,v_2}_{\{2,5\}}[c_6] = \{v_6, v_9\}$, $RS^{v_3,v_3}_{\{2,5\}}[c_5] = \{v_5, v_8\}$, and $RS^{v_3,v_3}_{\{2,5\}}[c_6] = \{v_6, v_9\}$. 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 in 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. 5(a)), we will first construct a trie-like graph as shown in Fig. 5(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}.

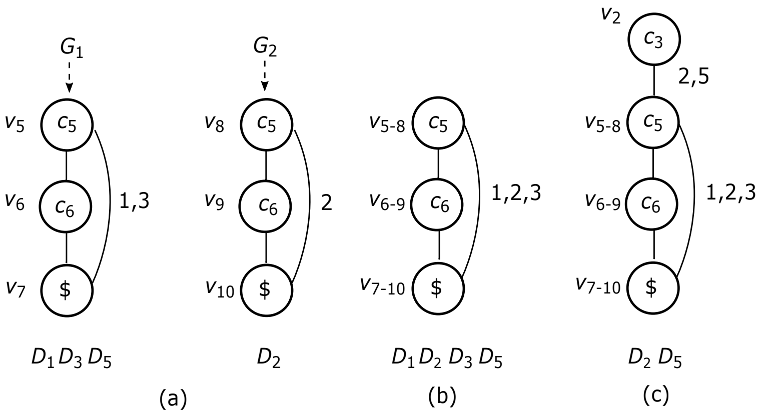

Fig. 5. Illustration for construction of trie-like subgraphs.

In addition, for technical convenience, we will add the corresponding branching node (v_3) to the trie as a virtual root, and a new edge $v_3 \xrightarrow{2,5} v_{5-8}$ as a virtual edge. See Fig. 5(c). 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 this branching node in T, which will greatly facilitate the trace of truth assignments for the corresponding satisfied conjunctions. Particularly, the label of a virtual edge $v \rightarrow u$ is set to be the label for the largest $RS_s^{v,w}$, where w is an anchor node of u. If there are more than one largest *RS*s, choose any one of them. For example, the label for the virtual edge shown in Fig. 5(c) is set to be {2, 5}. This is the label for $RS_{\{2,5\}}^{v_3,v_2}[c_5]$ (one of the two relevant *RS*s: $RS_{\{2,5\}}^{v_3,v_2}[c_5]$ and $RS_{\{2,5\}}^{v_3,v_3}[c_5]$. Both of them are of the same size.) In this way, the trace of the truth assignment for a subset of satisfied conjunctions can be very easily performed.

Now, searching the path from v_{7-10} to v_{5-8} in Fig. 5(c) bottom-up, going through the virtual node v_3 to find the corresponding anchor node v_2, and then searching the path from v_2 to v_0 in T (see Fig. 3), we will figure out a path:

$$v_0 \longrightarrow v_1 \longrightarrow v_2 \xrightarrow{2,5} v_{5-8} \longrightarrow v_{6-9} \longrightarrow v_{7-10},$$

representing a truth assignment $\{c_1 = 0, c_2 = 1, c_3 = 1, c_4 = 0, c_5 = 1, c_6 = 1\}$, satisfying $\{D_2, D_5\}$. Here, 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 label associated with the virtual edge $v_2 \rightarrow v_{5-8}$ is {2, 5} (which represent two spans: $v_2 \xrightarrow{5} v_5$, $v_2 \xrightarrow{2} v_8$ covering the branching node v_3), by which D_1 and D_3 are filtered out from $\{D_1, D_2, D_3, D_5\}$.

We remember that when generating the trie T over the main paths of the p^*-graphs created for the variable sequences shown in Table 1, 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 in fact useless. To avoid this kind of futile 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 shown below, in which a data structure R is used to accommodate the result, represented as a set of triplets of the form:

$<\alpha, \beta, \gamma>$,

where α stands for a subset of conjunctions, β for a truth assignment satisfying the conjunctions in α, and γ is the size of α. Initially, $R = \emptyset$.

Algorithm 1: *2-MAXSAT*(C)

```
Input  : a logic formula $C$ in CNF with each clause in $C$ containing at most two literals.
Output : a largest subset of clauses satisfying a certain truth assignment.
1 transform $C$ to another formula $D$ in DNF;
2 let $D = D_1 \vee ... \vee D_n$;
3 for $i = 1$ to $n$ do
4     construct a $p^*$-graph $P_i^*$ for $D_i$;
5 construct a trie-like graph $G$ over $P_1^*, ..., P_n^*$;
6 $R$ := SEARCH($G$);
7 return the result calculated in terms of $R$;
```

The input of *2-MAXSAT*( ) is a formula C in *CNF*. First, we transform it to another formula D in *DNF* (see line 1). Then, for each D_i in D, we will create its p^*-graph P_i^* (see lines 4). Next, we will contruct a trie-like graph G over all P_i^*'s (see line 5). In the last step, we call *SEARCH*(G) to produce the result (see line 6).

Algorithm 2: *SEARCH*(G)

```
    Input  : a trie-like subgraphs $G$.
    Output : a largest subset of conjunctions satisfying a certain truth assignment.
 1  if $G$ is a single $p^*$-graph then
 2      $R'$ := subset associated with the leaf node;
 3      $R$ := merge($R$, $R'$);
 4      return $R$;
 5  for each leaf node $v$ in $G$ do
 6      let $R'$ be the subset associated with $v$;
 7      $R$ := merge($R$, $R'$);
 8  let $v_1, v_2, ..., v_k$ be all branching nodes in postorder;
 9  for $i = 1$ to $k$ do
10      let $P$ be the tree path from root to $v_i$;
11      for each $u$ on $P$ do
12          calculate RSs of $u$ with respect to $v$
13      create the corresponding upBound $L$;
14      construct a trie-like graph $D$ over the subgraphs each rooted at a node on $L$;
15      $D' := \{v\} \cup D$;
16      $R'$ := SEARCH($D'$);
17      $R$ := merge($R$, $R'$);
18  return $R$;
```

The input of *SEARCH*( ) is a trie-like subgraph G. First, we will check whether G is a single p^*-graph. If it is the case, we must have found a largest subset of conjunctions associated with the leaf node, satisfiable by a certain truth assignment (see lines 1 - 4).

Otherwise, we will search G bottom up to find all the branching nodes in G. But before that, each subset of conjunctions associated with a leaf node will be first merged into R (see line 5 - 7).

For each branching node v encountered, we will check all the nodes u on the tree path from *root* to v and compute their *RS*s (see lines 8 - 12), based on which we then compute the corresponding upBound with respect to v (see line 13). According to the upBound L, a trie-like graph D will be created over a set of subgraphs each rooted at a node on L (see line 14). Then, v will be added to D as its root (see line 15). Here, we notice that $D' = \{v\} \cup D$ is a simplified representation of an operation, by which we add not only v, but also the corresponding virtual edges to D. Next, a recursive call of the algorithm is made over D' (see linee 16). Finally, the result of the recursive call of the algorithm will be merged into the global answer (see line 17).

Here, the *merge* operation used in line 3, 7, 17 is defined as below.

Let $R = \{r_1, ..., r_t\}$ for some $t \geq 0$ with each $r_i = <\alpha_i, \beta_i, \gamma_i>$. We have $\gamma_1 = \gamma_2 = ... = \gamma_t$. Let $R' = \{r'_1, ..., r'_s\}$ for some $s \geq 0$ with each $r'_i = <\alpha'_i, \beta'_i, \gamma'_i>$. We have $\gamma'_1 = \gamma'_2 = ... = \gamma'_s$. By *merge*($R$, R'), we will do the following checks.

- If $\gamma_1 < \gamma'_1$, $R := R'$.

- If $\gamma_1 > \gamma_1'$, R remains unchanged.
- If $\gamma_1 = \gamma_1'$, $R := R \cup R'$.

For simplicity, the heuristic discussed above is not incorporated into the algorithm. But it can be easily extended with this operation included.

Besides, to find a truth assignment satisfying a subset of conjunctions, we need to trace a path which may contain several spans, each corresponding to a recursive call of *SEARCH*( ).

We will represent a recursive call by a pair $<v, L>$, where v is a branching node in G, and L is the upBound with respect to v, over which a recursive call of *RESEARCH*( ) is invoked.

Then, a chain of recursive calls can be described as below:

$$<v_1, L_1> \longrightarrow <v_2, L_2> \longrightarrow ... \longrightarrow <v_k, L_k>,$$

where v_1 is a branching node in $G_0 = G$, v_i ($i = 2, ..., k$) is a branching node in G_{i-1}, the trie-like subgraph created by executing $<v_{1-1}, L_{i-1}>$, and L_i is the upBound with respect to v_i in G_{i-1}.

Denote by w_k a leaf node in G_k. Assume that D' is the subset of conjunctions associated with w_k. We will trace a path consisting of the following subpaths and spans, satisfying a largest subset of D'.

- p_i: treepaths from a child u_i of v_i to w_i in G_i ($i = k, ..., 1$), where w_i is the anchor node of u_{i+1} for $i = k$ - 1, ..., 0;
- e_i: spans connecting w_{i-1} and u_i ($i = k, ..., 1$);
- p_0: a treepath from the *root* of G to w_0.

See Fig. 6 for illustration.

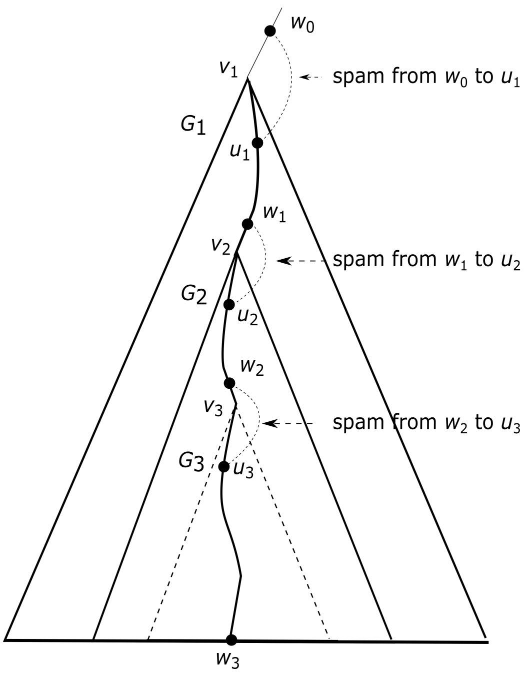

FIG. 6. Illustration for tracing truth assignments for satisfied conjunctions.

In Fig. 6, we show a chain of three recursivel calls:

$<v_1, L_1> \longrightarrow <v_2, L_2> \longrightarrow <v_3, L_3>$.

Here, we assume that v_1 is a branching node in G. By executing $<v_1, L_1>$, we will create G_1. Further, assume that v_2 is a branching node in G_1. Then, by executing $<v_2, L_2>$, we will generate G_2. Next, assume that v_3 is a branching node in G_2. We will create G_3 by executing $<v_3, L_3>$. We also assume that w_3 is a leaf node in G_3, associated with a subset D' of conjunctions.

Then, the path shown in Fig. 6 consists of three treepaths from u_i to w_i for i = 1, 2, 3, and three spans from w_i to u_{i+1} for i = 0, 1, 2, and a tree path from the *root* of G to w_0.

This path represents a truth assignment satisfying $s \cap D'$, where s is the intersection of all the edge labels on p. (s can be changed to the intersection of all the labels associated with the virtual edges on p since the intersection of all the tree edge labels is equal to or contains D', as indicated by Lemma 3).

EXAMPLE 2. *When applying SEARCH( ) to the p^*-graphs shown in Fig. 3, we will encounter three branching nodes: 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 certain 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 encounter this branching node, we will make a second 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. 7.

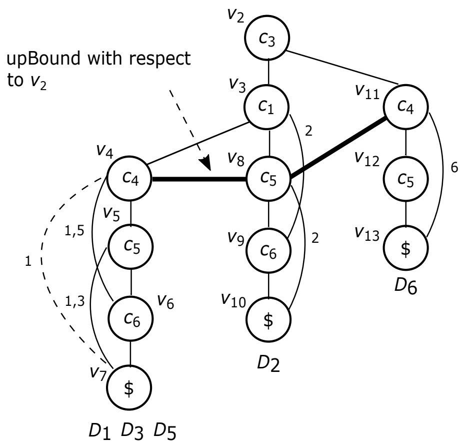

FIG. 7. Two subgraphs in $G[v_2]$ and an upBound.

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 branching node itself) have we the following four reachable subsets through spans:

- $RS^{v_2,v_2}_{\{3,5,6\}}[c_4] = \{v_4, v_{11}\}$,
- $RS^{v_2,v_2}_{\{2,5,6\}}[c_5] = \{v_5, v_8, v_{12}\}$,
- $RS^{v_2,v_2}_{\{2,5\}}[c_6] = \{v_6, v_9\}$,
- $RS^{v_2,v_2}_{\{2,6\}}[\$] = \{v_{10}, v_{13}\}$,

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 recognize the corresponding upper boundary $\{v_4, v_8, v_{11}\}$ (which is illustrated as a thick line in Fig. 7). Next, we will determine over what subgraphs a trie-like graph should be constructed, over which the algorithm will be recursively executed.

In Fig. 8, we show the trie-like graph built over the three p^*-subgraphs (rooted respectively at v_4, v_8, v_{11} on the upBound shown in Fig. 7), 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 branching node v_2 is involved as the virtual root of this trie-like subgraph. The virtual edge $v_2 \xrightarrow{3,5,6} v_{4-11}$ is labeled with $\{3, 5, 6\}$ since it stands for a span (from v_2 to v_4) labeled with $\{3, 5\}$, and a tree edge (from v_2 to v_{11}) labeled with $\{6\}$ in Fig. 3. The virtual edge $v_2 \xrightarrow{2} v_8$ is labeled with $\{2\}$ since it represents a span (from v_2 to v_8) labeled with $\{2\}$. In addition, all the spans going out of v_2 in the original graph are kept around (see Fig. 3).

By the corresponding recursive call of *SEARCH*( ), this graph will be constructed and then searched bottom up, by which we will encounter the first branching nodes: v_{5-12}. Then, a next recursive call of the algorithm will be conducted, generating an upBound $\{v_7, v_{13}\}$, as shown in Fig. 9(a). Similar to the above discussion, we will construct the corresponding trie-like subgraph, which is just a single merged node v_{7-13} as shown in Fig. 9(b). Adding the corresponding virtual root v_{5-12}, and virtual edge $v_{5-12} \xrightarrow{1,3,6} v_{7-13}$ (representing a span $v_{5-12} \xrightarrow{1,3} v_7$ and a tree edge $v_{5-12} \xrightarrow{6} v_{13}$), we will get a path as shown in Fig. 9(c), by which we will find a largest subset of conjunctions $\{D_3, D_6\}$, satifiable by a certain truth assignment: $c_1 = 0$, $c_2 = 1$, $c_3 = 1$, $c_4 = 1$, $c_5 = 1$, $c_1 = 0$. This truth assignment can be figured by tracing the corresponding path:

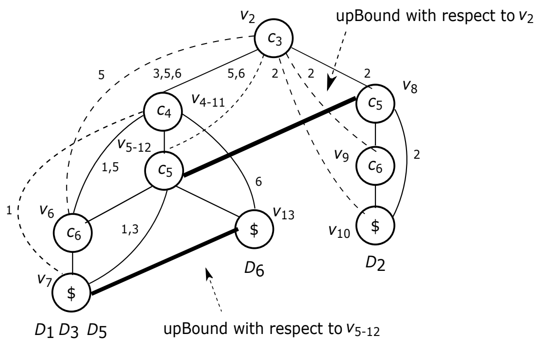

FIG. 8. A trie-like graph.

$$v_0 \longrightarrow v_1 \longrightarrow v_2 \xrightarrow{3,5,6} v_{4-11} \longrightarrow v_{5-12} \xrightarrow{1,3,6} v_{7-13}.$$

Special attention should be paid to the leaf node of the path shown in Fig. 9(c). It is associated with $\{D_3, D_6\}$, instead of $\{D_1, D_3, D_5, D_6\}$. It is because the intersection of all the labels associated with the virtual edges is $\{3, 5, 6\} \cap \{1, 3, 6\} = \{3, 6\}$ and D_1, D_5 should be removed.

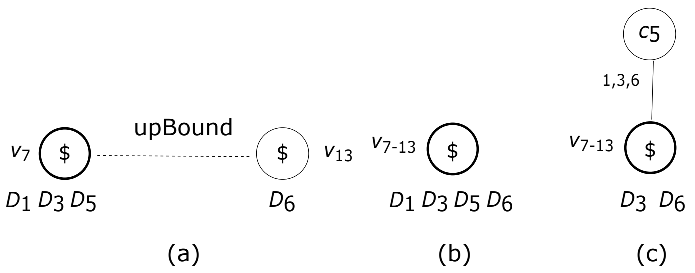

FIG. 9. Illustration for construction of a trie-like subgraph.

Continuing the search of the graph shown in Fig. 8, we will encounter its second branching node v_2 , by which another set of RSs will be created:

- $RS^{v_2,v_1}_{\{3,6\}} = \{v_{4-11}\}$
(due to the span $v_1 \xrightarrow{3,6} v_{4-11}$, which corresponds to two spans in Fig. 3: $v_1 \xrightarrow{3} v_4$ and $v_1 \xrightarrow{6} v_{11}$),

- $RS^{v_2,v_2}_{\{2,5,6\}}[c_5] = \{v_{5-12}, v_8\}$
(due to the span $v_2 \xrightarrow{5,6} v_{5-12}$ and the tree edge $v_2 \xrightarrow{2} v_8$ in Fig. 8),

- $RS^{v_2 v_2}_{\{2,5\}}[c_6] = \{v_6, v_9\}$
(due to the spans $v_2 \xrightarrow{5} v_6$ and $v_2 \xrightarrow{2} v_9$ in Fig. 8).

Since $|RS^{v_2,v_1}_{\{3,6\}}| = 1$, it will not be further considered in the subsequent computation.
However, in terms of $RS^{v_2,v_2}_{\{2,5,6\}}[c_5]$ and $RS^{v_2,v_2}_{\{2,5\}}[c_6]$, we will construct an upBound $\{v_{5-12}, v_8\}$ (see Fig. 8), and create a trie-like graph as shown in Fig. 10(a). Add the virtual node and the vitual edge

as shown in Fig. 10(b), where the label associated with the virtual edge is set to be the same as for $RS^{v_2,v_2}_{\{2,5,6\}}[c_5]$. The only branching node in this graph is v_{5-12-8}. With respect to v_{5-12-8}, v_2 has two *RS*s in terms of two spans respectively to two nodes (v_{6-9} and v_{7-10}) in this subgraph (see Fig. 10(c). Also see Fig. 8 to know how these two spans are created):

- $RS^{v_{5-12-8},v_2}_{\{2,5\}}[c_6] = \{v_{6-9}\}$
(due to the span $v_2 \xrightarrow{2,5} v_{6-9}$ in Fig. 10(c)),

- $RS^{v_{5-12-8},v_2}_{\{2\}}[\$] = \{v_{7-10}\}$
(due to the span $v_2 \xrightarrow{2} v_{7-10}$ in Fig. 10(c)).

Both of these *RS*s are of size 1. Therefore, they will simply be ignored.
For v_{5-12-8} itself, we have the following *RS*:

- $RS^{v_{5-12-8},v_{5-12-8}}_{\{1,2,3,6\}}[\$] = \{v_{7-10}, v_{13}\}$.

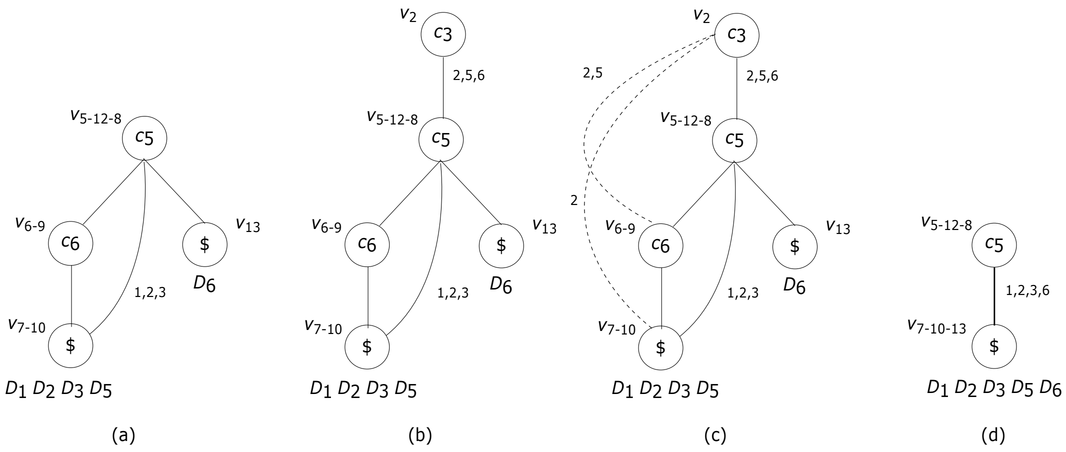

Fig. 10. Illustration for recursive execution of algorithm.

According to this *RS*, we will construct the corresponding trie-like graph, as shown in Fig. 10(d), in which the virtual node is v_{5-12-8} and the label of the virtual edge is $\{1, 2, 3, 6\}$. By tracing the corresponding path:

$$v_0 \longrightarrow v_1 \longrightarrow v_2 \xrightarrow{2,5,6} v_{5-12-8} \xrightarrow{1,2,3,6} v_{7-10-13}.$$

we will get a truth assignment: $c_1 = 0$, $c_2 = 1$, $c_3 = 1$, $c_4 = 0$, $c_5 = 1$, $c_6 = 0$, satisfying a subset $\{D_2, D_6\}$. It is because $\{2, 5, 6\} \cap \{1, 2, 3, 6\} = \{2, 6\}$ and D_1, D_3 D_5 are filtered out from the subset associated with the leaf node in Fig, 10(d).

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

Concerning the correctness of Algorithm 2, we have the following proposition.

PROPOSITION 2. *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 branching 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', the span $u \xrightarrow{s} v$, 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 encounter the first branching node in G', by which we will find P'' satisfying D''. Then, the concantenation of P' and P'' definitely satisfies D'. This completes the proof. □

3.3 Further improvement

The algorithm discussed in the previous subsection can be greatly improved in two ways. First, we can remove a lot of useless recursive calls of *SEARCH*( ) by imposing some extra controls. Secondly, any repeated recursive call can also be effectively avoided by checking same trie-like subgraphs repeatedly encountered.

- *Reducing recursive calls*

Consider Fig. 11(a). In this figure, we assume that w and w' are two branching nodes in G. Then, with respect to w and w', their ancestor u will have two identical *RS*s:

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

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

However, if we create *RS*s only for those nodes appearing on part of a tree path, i.e., the segment between the current branching node and the lowest ancestor branching node in T, this kind of redudancy can be avoided with possible lose of some answers. But the correctness of the algorithm is not affected since one of the maximum satisfiable subsets of conjunctions can always be found. See Fig. 11(b) for illustration. For this figure, the *RS* of u with respect to w is different from the *RS* with respect to w'. However, when checking w, $RS_s^{w,u}[\mathrm{C}]$ will not be computed since u is beyond the segment between w and w'. Therefore, the corresponding result will not be generated. However,

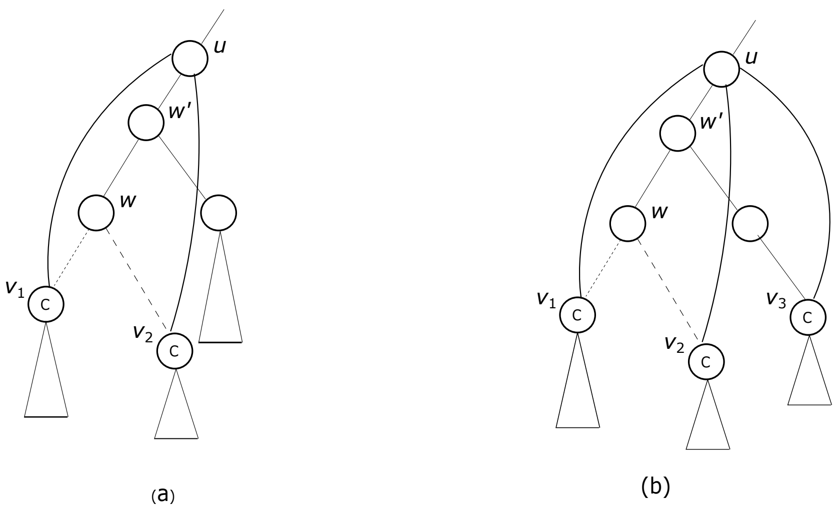

Fig. 11. Illustration for redundancy.

$RS_s^{w',u}$[C] must cover $RS_s^{w,u}$[C], implying a larger (or same-sized) subset of conjunctions which can be satisfied by a certain truth assignment.

- *Avoiding repeated recursive calls*

Now we consider Fig. 11(b) once again. Denote by G_1 the trie-like graph made over the subtrees respectively rooted at v_1 and v_2, and by G_2 the trie-like graph made over the subtrees respectively rooted at v_1, v_2, and v_3. It is possible that G_1 and G_2 contain some common branching nodes. Therefore, repeated recursive calls on the same trie-like subgraphs can be possibly conducted. To avoid this kind of redundancy, we can examine, by each recursive call, whether the input subgraph has been checked before. If it is the case, the corresponding recursive call should be simply suppressed. This obviously does not impact the correctness of the algorithm since a recursive call on a same subgraph will find only the same satisfiable subset of conjunctions (but with possible different assignments of variables since the trie-like subgraph may be reached through different spans). For this purpose, we will maintain a hash array with each entry used to store the result obtained by a recursive call on a certain trie-like subgraph. Specifically, for each recursive call <v, L> (this notation was first introduced before Example 2 to describe the chains of recursive calls), we will store the result in the address *hash*(L). Thus, to examine whether an input subgraph has been checked before, we need only a constant time.

4 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.

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 $O(m)$ 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 $O(n^2m^4)$.

However, we need to make clear that each branching node 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. 12(a), in which w is a branching node. With respect to w, we will have the following three *RS*s:

- $RS_{s'}^{w,u}[\text{C}] = \{v_1, v_2\}$,

- $RS_{s''}^{w,u}[\text{D}] = \{v_3, v_5, v_6\}$,

- $RS_{s'''}^{w,u}[\text{E}] = \{v_4, v_7, v_8, v_9\}$,

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

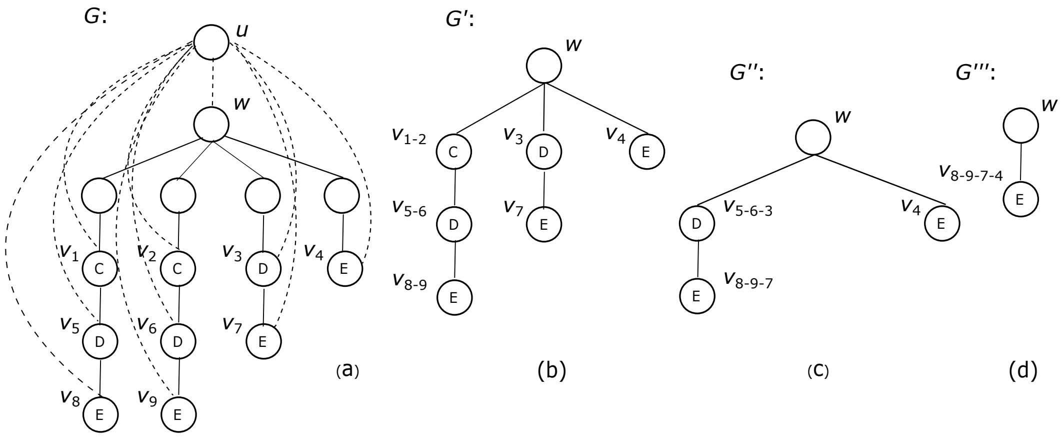

FIG. 12. Illustration for recursive construction of trie-like subgraphs.

According to these *RS*s, we will construct a trie-like subgraph G' as shown in Fig. 12(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. 12(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. 12(d).

Together with the method discussed in the previous section to avoid repeated recursive calls on of a same trie-like subgraph, the above analysis shows that any branching node can be involved in at most m recursive calls of *SEARCH*( ). In general, we have the following proposition.

Proposition 3. *Let G be a trie-like graph and w be a branching node of Cae 1 in the corresponding layered graph. Then, w can be involved in at most m recursive calls of SEARCH( ) (Algorithm 3) 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 satisfying the following three properties:

- Each v_i $(i = 1. ..., k)$ has no ancestor appearing 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 recursive calls on a same trie-like subgraph), the proposition holds. □

Proposition 4. *Let G be a trie-like graph over a formula in DNF containing n conjunctions with m variables. The time complexity of Algorithm SEARCH(G) is bounded by* $O(n^2m^4)$.

Proof. From Proposition 3, we can see that in the whole working process at most $O(nm) \times m$ trie-like subgraphs can be generated. Thus, at most $O(nm) \times m$ recursive calls can be carried out since any repeated recursive call on a same trie-like subgraph can be simply and effectively avoided. Therefore, the time complexity of *SEARCH*(G) is bounded by $O(nm) \times m \times O(nm^2) = O(n^2m^4)$. □

5 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* with each clause containing at most 2 literals. 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.