Contemporary Mathematics
http://ojs.wiserpub.com/index.php/CM/

UNIVERSAL WISER PUBLISHER

Research Article

An Efficient Algorithm for Solving the 2-MAXSAT Problem

Yangjun Chen

Department of Applied Computer Science, University of Winnipeg, Manitoba, Canada
Email: y.chen@uwinnipeg.ca

Received: 30 June 2023; **Revised:** 18 October 2023; **Accepted:** 1 November 2023

Abstract: In the maximum satisfiability (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, in which every clause contains at most two literals. In this paper, we discuss an efficient algorithm to solve this problem. Its worst-case time complexity is bounded by $O(n^2m^3(\log_2 nm)^{\log_2 nm})$. In the case that $\log_2 nm$ is bounded by a constant, our algorithm is a polynomial algorithm. In terms of Garey and Johnson, any satisfiability instance can be transformed to a 2-MAXSAT instance in polynomial time. Thus, our algorithm may lead to a proof of $P = NP$.

Keywords: satisfiability problem, maximum satisfiability problem, *NP*-hard, *NP*-complete, conjunctive normal form, disjunctive normal form

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, 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 [1]. The problem is *NP*-complete even when every clause in C has at most three literals [2]. The maximum satisfiability (MAXSAT) problem is an optimization version of satisfiability that seeks a truth assignment to maximize the number of satisfied clauses [3]. This problem is *NP*-hard even for its restricted version, the so-called 2-MAXSAT problem, in which every clause in C has at most two literals [4]. Its application can be seen in an extensive bibliography [5-10].

Over the past several decades, a lot of research on the MAXSAT has been conducted. Almost all of them are the approximation methods [3, 7, 11-13, 21], such as (1-1/e)-approximation and 3/4-approximation [7], as well as the method based on integer linear programming [9]. The only algorithms for an exact solution are discussed in [20, 21]. The worst-case time complexity of [21] is bounded by $O(b^2 2^m)$, where b is the maximum number of occurrences of any variable in the clauses of C, while the worst-case time complexity of [20] is bounded by $\max\{O(2^m), O^*(1.2989^n)\}$. Both algorithms use the traditional branch-and-bound method to solve the satisfiability problem. Its main idea is to search for a solution by letting a variable (or a literal) be 1 or 0. In terms of [7], any algorithm based on branch-and-bound runs in $O^*(c^m)$ time with $c \geq 2$.

In this paper, we discuss an efficient algorithm to solve the 2-MAXSAT problem, working in a quite different way.

DOI: https://doi.org/10.37256/cm.5320243304

Its time complexity is bounded by $O(n^2m^3(\log_2 nm)^{\log_2 nm})$, where n and m are the number of clauses and the number of variables in C, respectively. In the case that $\log_2 nm$ is bounded by a constant, the time complexity of our algorithm is polynomial. In terms of Garey and Johnson [8], any satisfiability instance can be transformed to a 2-MAXSAT instance in polynomial time. This shows a possibility of proving $P = NP$. For example, for a large problem with $n = 32$ and $m = 64$, $\log_2 nm^{\log_2 nm} = 11^{11}$ is still a manageable constant. But the running time of any existing method is $> O(2^{64})$, which is obviously prohibitively high.

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, each containing at most two 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 two 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 ($i \in \{1, ..., n\}$), 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 an almost polynomial time.

The organization of the rest of this paper is as follows: 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 our algorithm in Section 3. Section 4 is devoted to the analysis of the time complexity of the 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 that 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 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 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 other words, we will find a maximum integer j such that j clauses in C can be satisfied under a certain truth assignment $\sigma : j = \max\{k \mid$ there exists σ such that k clauses in C are satisfied under $\sigma\}$.

In terms of [6], 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 [16]. We may also use a heuristic method to find an approximate solution to the problem, as described in [8].

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

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

$$D_{i1} = c_{i1} \wedge x_i,$$

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

Let $D = D_{11} \vee D_{12} \vee D_{21} \vee D_{22} \vee ... \vee D_{n1} \vee D_{n2}$.

Then, given an instance of the 2-MAXSAT problem defined over a variable set V and a collection C of n clauses, we can construct a logic formula D in DNF over the set $V \cup X$ in polynomial time, where $X = \{x_i \mid i = 1, ..., n\}$. D has $m = 2n$ conjunctions.

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

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

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

"⇒" 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 [11], 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, (1) set both 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 $\tilde{\sigma}$.

"⇐" We now suppose that a truth assignment $\tilde{\sigma}$ for D with $q \geq n^*$ conjunctions in D is 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 least 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, into a problem to find the maximum number of conjunctions in a logic formula in DNF. As an example, consider the following logic formula in CNF:

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

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:

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

According to Proposition 1, D should also have at least $n^* = 3$ conjunctions that evaluate to *true* under some truth assignment. On the other hand, if D has at least three 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 an almost polynomial time algorithm to find a maximum set of satisfied conjunctions in any logic formula in DNF, not only restricted to the case that each conjunction contains up to two 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 a 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 variable sequence. For this purpose, we introduce a new notation:

$$\left(c_j, ^*\right) = c_j \vee \neg c_j = true$$

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:

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

Doing this enables us to represent each D_i as a variable sequence, but with all the negative literals 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.

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 eliminated. In the same way, you can check all the other variable sequences.

Now it is easy for us to compute the appearance frequencies of different variables in the variable sequences, by which each (c, *) is counted as a single appearance of c while any negative literals are not considered, as illustrated in Table 2, in which we show the appearance frequencies of all the variables in the above D.

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

Table 1. Conjunctions represented as sorted variable sequences

Conjunction	Variable sequences	Sorted variable sequences
D_1	$c_1 \cdot (c_2,^*) \cdot (c_3,^*) \cdot c_4 \cdot (c_5,^*) \cdot (c_6,^*)$	$\#.(c_2,^*) \cdot (c_3,^*) \cdot c_1 \cdot c_4 \cdot (c_5,*) \cdot (c_6,^*) \cdot \$$
D_2	$(c_1,^*) \cdot c_2 \cdot c_3 \cdot (c_5,^*) \cdot (c_6,*)$	$\# \cdot c_2 \cdot c_3 \cdot (c_1,*) \cdot (c_5,^*) \cdot (c_6,*) \cdot \$$
D_3	$(c_1,^*) \cdot c_2 \cdot (c_3,^*) \cdot c_4 \cdot c_5 \cdot (c_6,^*)$	$\# \cdot c_2 \cdot (c_3,*) \cdot (c_1,*) \cdot c_4 \cdot c_5 \cdot (c_6,^*) \cdot \$$
D_4	$(c_1,^*) \cdot (c_2,^*) \cdot (c_4,*) \cdot (c_6,^*)$	$\# \cdot (c_2,*) \cdot (c_1,*) \cdot (c_4,*) \cdot (c_6,^*) \cdot \$$
D_5	$(c_1,^*) \cdot (c_2,^*) \cdot c_3 \cdot (c_4,^*) \cdot (c_5,^*) \cdot c_6$	$\# \cdot (c_2,^*) \cdot c_3 \cdot (c_1,*) \cdot (c_4,*) \cdot (c_5,^*) \cdot c_6 \cdot \$$
D_6	$(c_2,^*) \cdot (c_3,^*) \cdot (c_4,^*) \cdot (c_5,^*)$	$\# \cdot (c_2,*) \cdot (c_3,*) \cdot (c_4,^*) \cdot (c_5,*) \cdot \$$

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

Following this general ordering, each conjunction D_i in D can be represented as a sorted variable sequence, 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 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 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 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 ($i = 1, \ldots, n$) (called a p^*-graph), under which D_i evaluates to *true*. In the following, however, we first define the simple concept of p-graphs for ease of explanation.

Definition 1 (p-graph). Let $\alpha = c_0 c_1 \ldots c_k c_{k+1}$ be a 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 $c_j (j = 0, \ldots, k+1)$ and an edge for (c_j, c_{j+1}) for each $j \in \{0, 1, \ldots, k\}$. In addition, there may be an edge from c_j to c_{j+2} for each $j \in \{0, \ldots, k-1\}$ if c_{j+1} is a pair of the form $(c, *)$, where c is a variable name.

In Figure 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, and 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).

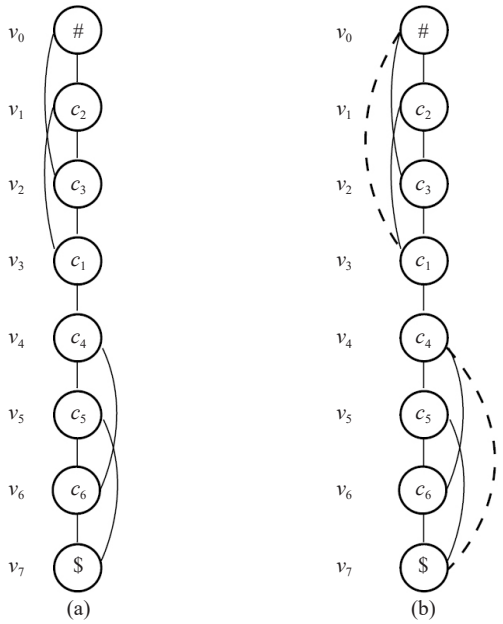

Figure 1. A *p*-path and a *p**-path

In fact, what we want is to represent all those truth assignments for each D_i (i = 1, …, n) in an efficient way, under which D_i evaluates to *true*. However, *p*-graphs fail to do so since when we go 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 u to (c, *) makes c always selected. So, the notation (c, *), which is used to indicate that c is optional, is not correctly implemented.

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

Let $s_1 = < v_1, \ldots, v_k >$ and $s_2 = < u_1, \ldots, u_l >$ be two spans attached to the same path. We say s_1 and s_2 are overlapped if $u_1 = v_j$ for some $j \in \{1, \ldots, k-1\}$ or if $v_1 = u_{j'}$ for some $j' \in \{1, \ldots, l-1\}$. For example, in Figure 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 Figure 1(b), we show the *p**-graph with respect to the *p*-graph shown in Figure 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 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 [13], denoted as T = trie(R) over $R = \{p_1, ..., p_n\}$, as follows.

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

In Figure 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 labelled with \$, representing a conjunction (or a subset of conjunctions, which can be satisfied under the truth assignment represented by the corresponding main path.)

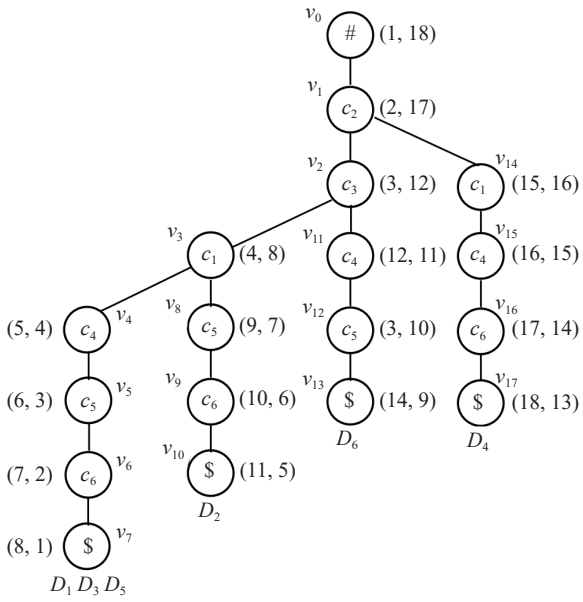

Figure 2. A trie and tree encoding

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 later), over which some dynamical tries can be recursively constructed, leading to an efficient algorithm.

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, a node in T is said to be in a high position if it is close to the root. It is said to be in a low position if it is close to a leaf node.

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.

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

For the proof of this property of any tree, see Exercise 2.3.220 in [12].

For instance, by checking the label associated with v_2 against the label for v_9 in Figure 2, we see that v_2 is an ancestor of v_9 in terms of this property. Particularly, 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 Figure 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 Figure 3) is associated with three numbers: 1, 5, and 6, indicating that the span belongs to three conjunctions: D_1, D_5, and D_6. But no numbers are associated with any tree edges.

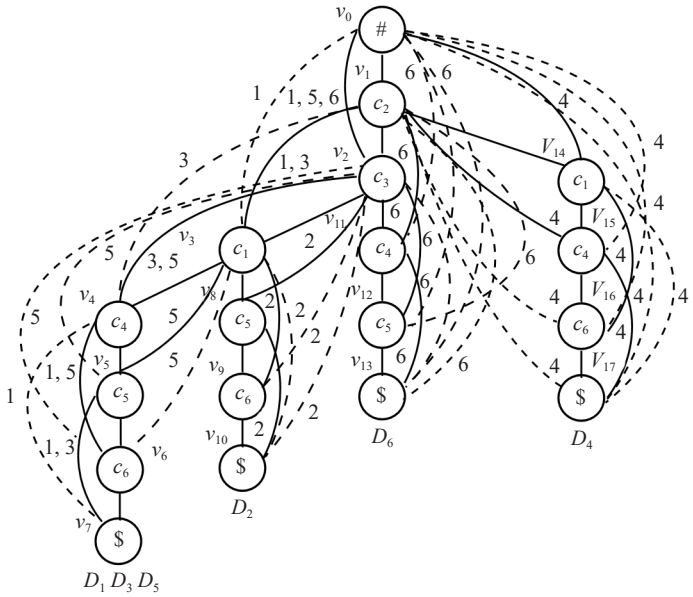

Figure 3. A trie-like graph G

From Figure 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.

In the next step, we will search G bottom-up level by level to seek all the possible largest subsets of conjunctions that 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 Figure 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 one child in T (although it has more than one child in G).

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

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

For instance, v_3 in Figure 3 is a branching node. With respect to v_3, node v_2 on the tree path from root to v_3 has two reachable subsets:

$$RS_{v_2}^{v_3}[c_5] = \{v_5, v_8\},$$

$$RS_{v_2}^{v_3}[c_6] = \{v_6, v_9\}.$$

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

In general, we are interested only in those *RS*'s (reachable subsets through spans) with $|RS_v| \geq 2$ (since any *RS* with $|RS| = 1$ only leads us to a leaf node in *T*: no larger subsets of conjunctions can be found.) So, in the subsequent discussion, by an RS_v, we mean an RS_v with $|RS_v| \geq 2$.

The definition of this concept for a 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 as a subset of nodes reachable from a span or from a tree edge. So, for v_3, we have:

$$RS_{v_3}^{v_3}[c_5] = \{v_5, v_8\},$$

$$RS_{v_3}^{v_3}[c_6] = \{v_6, v_9\},$$

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

Based on the concept of reachable subsets through spans, we are able to define another more important concept, upper boundaries, given below.

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

1) Each $u_g (1 \leq g \leq f)$ appears in some $RS_{v_i}^{v}[c] (1 \leq i \leq k)$, 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.

Figure 4 gives an intuitive illustration of this concept.

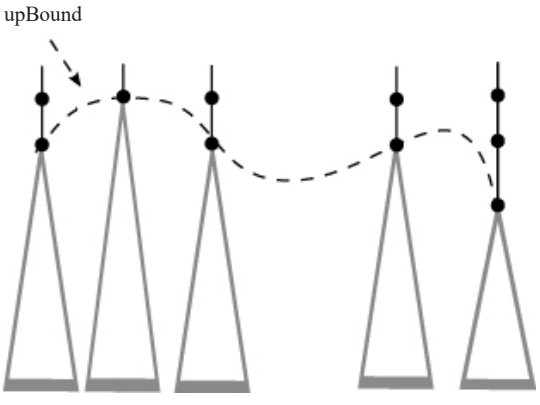

Figure 4. Illustration for upBounds

As a concrete example, consider v_5 and v_8 in Figure 3. They make up an upBound with respect to v_3 (a branching node). Then, we will construct a trie-like graph over two p*subgraphs, starting from v_5 and v_8, respectively. This can

be done by a recursive call of the algorithm itself. Here, however, v_4 is not included since the truth assignment with v_4 being set to *true* satisfies only the conjunctions associated with leaf node v_{10}. This has already been determined when the initial trie is built up. In fact, the purpose of upper boundaries is to remove all those nodes like v_4 from the subsequent computation.

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

• Calculate all *RS*s with respect to v.

• Calculate the upBound in terms of *RS*s.

• Make a recursive call of the algorithm over all the subgraphs within $G[v]$, each rooted at a node on the corresponding upBound.

The following example helps with 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. They are $RS^{v_3}_{v_2}[c_5]$, $RS^{v_3}_{v_2}[c_6]$, $RS^{v_3}_{v_3}[c_5]$, and $RS^{v_3}_{v_3}[c_6]$. In terms of these reachable subsets through spans, we will get the corresponding upBound $\{v_5$ and $v_8\}$. Node v_4 (above the upBound) will not be involved. Therefore, it will not be further considered 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 (shown in Figure 5(a)), we will first construct a trie-like graph as shown in Figure 5(b). Here, we notice that the subset associated with its unique leaf node is $\{D_2$ and $D_5\}$, instead of $\{D_1, D_2, D_3$, and $D_5\}$. It is because the number associated with span $v_2 \xrightarrow{5} v_5$ is 5, by which D_1 and D_3 are removed. But the number associated with span $v_2 \xrightarrow{2} v_8$ is 2, and therefore D_2 associated with v_{10} survives.

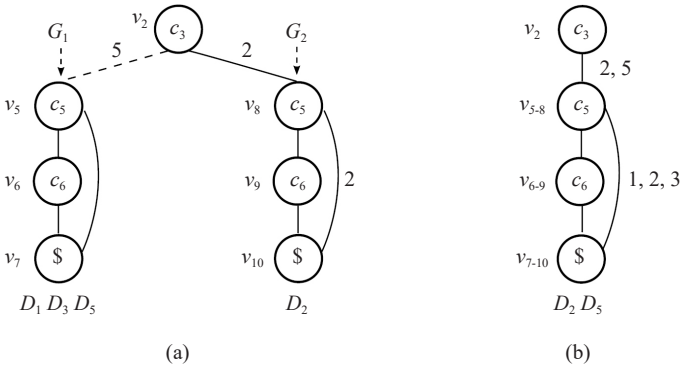

Figure 5. Illustration for recursive call of the algorithm applied to G_1 and G_2

In the trie-like graph shown in Figure 5(b), $v_{5–8}$ stands for the merge of v_5 and v_8; $v_{6–9}$ for v_6 and v_9; and $v_{7–10}$ for v_7 and v_{10}. By searching this graph, we will find the truth assignment satisfying $\{D_2$ and $D_5\}$. This truth assignment is represented by a path consisting of two parts: the tree path from root to v_2 (see Figure 3), and the path from v_2 to v_7, i.e., the span $v_2 \xrightarrow{2,5} v_5$, connected to the subpath v_5 to v_7 (see Figure 5(b)). So, the truth assignment is $\{c_1 = 0, c_2 = 1, c_3 = 1, c_4 = 0, c_5 = 1, c_6 = 1\}$.

We remember that when generating the trie T over the main paths of the p*-graphs over the variable sequences shown in Table 1, we have already found a subset of conjunctions $\{D_1, D_3$, and $D_5\}$, which can be satisfied by a truth assignment represented by the corresponding main path. This is larger than $\{D_2$ and $D_5\}$. Therefore, $\{D_2$ and $D_5\}$ should not be kept around, and this part of the computation is futile. However, we can avoid this kind of useless work by performing a pre-check: if the number of p*-subgraphs over which the recursive call of the algorithm will be invoked is smaller than the size of the partial answer already obtained, the corresponding recursive call of the algorithm should not be conducted. This check can be extended to a very powerful heuristic:

With each recursive call, we will examine whether the input subgraph has been checked before. If this is the case, the corresponding recursive call will be suppressed.

In terms of the above discussion, we design a recursive algorithm to do the task, in which 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.

Output: the largest subset of conjunctions 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 return *SEARCH*(G);

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 line 4). Next, we will construct 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).

The input of *SEARCH*( ) is a trie-like subgraph G. First, we will check whether G is a single p*-graph. If this is the case, we must have found the largest subset of conjunctions associated with the leaf node, satisfied by a truth assignment represented by a path consisting of two parts: the tree path from the root to the starting node v of the single p*-graph, and the main path from v to the leaf node of the single p*-graph. This subset should be merged into R (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 in R will be first merged into R (see lines 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, in which we add not only v but also the corresponding edges to D. Next, a recursive call of the algorithm is made over D' (see line 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 lines 3, 7, and 17 is defined as below.

Let $R = \{r_1, \ldots, r_t\}$ for some $t \geq 0$ with each $r_i = < \alpha_i, \beta_i, \gamma_i >$. We have $\gamma_1 = \gamma_2 = \ldots = \gamma_t$. Let $R' = \{r_1', \ldots, r_s'\}$ for some $s \geq 0$ with each $r_i' = < \alpha_i', \beta_i', \gamma_i' >$. We have $\gamma_1' = \gamma_2' = \ldots = \gamma_s'$. By merge($R$, R'), we will do the following checks.

• If $\gamma_1 < \gamma_{1'}, R \coloneqq R'$.

• If $\gamma_1 > \gamma_{1'}, R$ remains unchanged.

• If $\gamma_1 = \gamma_{1'}, R \coloneqq R \cup R'$.

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

The following example helps with illustration.

Algorithm 2: *SEARCH*(G)

Input: a trie-like subgraphs G.

Output: the largest subset of conjunctions satisfying a certain truth assignment.

```
if G is a single p*-graph then
    R' := subset associated with the leaf node;
    R := merge(R, R');
    return R;
for each leaf node v in G do
    let R' be the subset associated with v;
    R := merge(R, R');
let v_1, v_2, ..., v_k be all branching nodes in postorder;
for i = 1 to k do
    let P be the tree path from root to v_i;
    for each u on P do
        calculate RSs of u with respect to v_i
    create the corresponding upBound L;
    construct a trie-like graph D over all those subgraphs
    each rooted at a node on L;
    D' := {v_i} ∪ D;
    R' := SEARCH(D');
    R := merge(R, R');
return R;
```

Example 2 When applying *SEARCH( )* to the p*-graphs constructed for all the variable sequences given in Table 1, we will first construct a trie-like graph G shown in Figure 3. Searching G bottom up, we will encounter three branching nodes: v_3, v_2, and v_1.

• Initially, when creating T, a subset of conjunctions $\{D_1, D_2, \text{and } D_5\}$, is found (see Figure 2), which can be satisfied by the same truth assignment represented by the corresponding path: $c_1 = 1$, $c_2 = 1$, $c_3 = 1$, $c_4 = 1$, $c_5 = 1$, $c_6 = 1$.

• Checking v_3. As shown in Example 1, by this checking, we will find a subset of conjunction $\{D_2 \text{ and } D_5\}$ 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 $\{D_1, D_2, \text{and } D_5\}$. Thus, this result will not be kept around.

• Checking v_2. In $G[v_2]$, we have two subgraphs in $G[v_2]$, as shown in Figure 6, subgraphs rooted at v_3 and v_{11}, respectively.

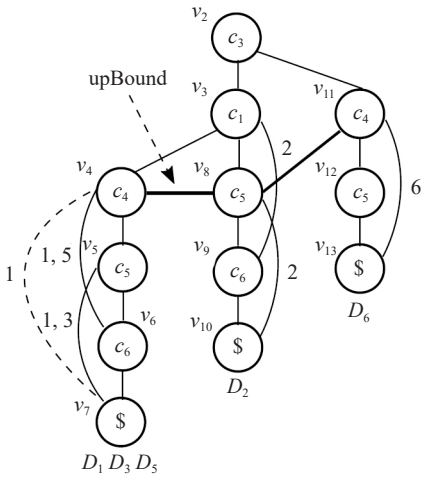

Figure 6. Two subgraphs in $G'[v_2]$ and a upBound

As with branching node v_3, we need to calculate all the relevant reachable subsets through spans for all the nodes on the tree path from the 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 Figure 3), we have

$$RS_{v_1}^{v_3}[c_4] = \{v_4, v_{11}\}$$

due to the following two spans (see Figure 3):

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

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

$$RS_{v_2}^{v_3}[c_4] = \{v_4, v_{11}\}$$

$$RS_{v_2}^{v_3}[c_5] = \{v_5, v_8, v_{12}\}$$

$$RS_{v_2}^{v_3}[c_6] = \{v_6, v_9\}$$

$$RS_{v_2}^{v_3}[\$] = \{v_{10}, v_{13}\}$$

Due to the following four groups of spans:

$$\{v_2 \xrightarrow{3,5} v_4, v_2 \xrightarrow{6} v_{11}\}.$$

$$\{v_2 \xrightarrow{5} v_5, v_1 \xrightarrow{2} v_8, v_1 \xrightarrow{6} v_{12}\}.$$

$$\{v_2 \xrightarrow{5} v_6, v_2 \xrightarrow{2} v_9\}.$$

$$\{v_2 \xrightarrow{2} v_{10}, v_2 \xrightarrow{6} v_{13}\}.$$

In terms of the reachable subsets through spans, we can establish the corresponding upper boundary {v_4, v_8, and v_{11}} (which is illustrated as a thick line in Figure 6). Then, we can determine, over what subgraphs a recursive execution of the algorithm will be conducted.

In Figure 7(a), we show the trie-like graph built over the three p*-subgraphs (starting, respectively, from v_4, v_8, and v_{11} on the upBound shown in Figure 6), in which $v_{4\text{-}11}$ stands for the merging of v_4 and v_{11}, and $v_{5\text{-}12}$ for the merging of v_5 and v_{12}. Especially, v_2 itself needs to be involved as a branching node, working as a bridge between the newly constructed trie-like graph and the rest part of G. (See operation $D' = \{v_i\} \cup D$ in line 15 of Algorithm 2.)

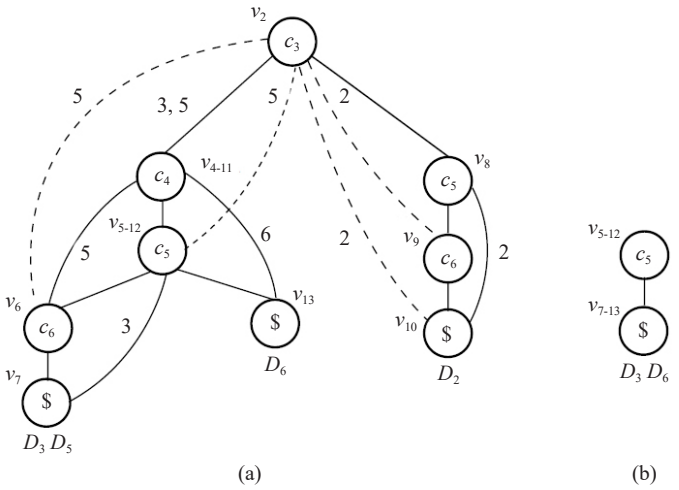

Figure 7. A trie-like graph

By a recursive call of *SEARCH( )*, we will construct this graph and then search this graph bottom up, by which we will find two branching nodes: $v_{5\text{–}12}$ and v_2. By checking $v_{5\text{-}12}$, we will find

$$RS_{v_{5\text{–}12}}^{v_{5\text{–}12}}[\$] = \{v_7, v_{13}\}.$$

The corresponding upBound is {v_7 and v_{13}}. Accordingly, we will find a single path as shown in Figure 7(b), by which we will find the largest subset of conjunctions {D_3 and D_6}, which can be satisfied by a certain truth assignment. We notice that the subset associated with this path is {D_3 and D_6}, instead of {D_3, D_5, and D_6}. It is because the span from $v_{5\text{–}12}$ to v_7 (in Figure 7(a)) is labelled with 3, and D_5 should be removed.

By checking v_2, we will have

$$RS_{v_2}^{v_2}[c_5] = \{v_{5\text{–}12}, v_8\}.$$

(due to the span $v_2 \xrightarrow{5} v_{5\text{–}12}$ and the tree edge $v_2 \to v_8$.)

$$RS_{v_2}^{v_2}[c_6] = \{v_6, v_9\}.$$

(due to the spans $v_2 \xrightarrow{5} v_6$ and $v_2 \xrightarrow{2} v_9$.)

Accordingly, the corresponding upBound is {$v_{5\text{–}12}$, and v_8}. Then, by the recursive execution of the algorithm,

we will create a trie-like graph as shown in Figure 8(a). The only branching node is $v_{5\text{-}12\text{-}8}$. Checking this node, we will finally get a single path as shown in Figure 8(b), showing the largest subset of conjunctions that can be satisfied by a certain truth assignment.

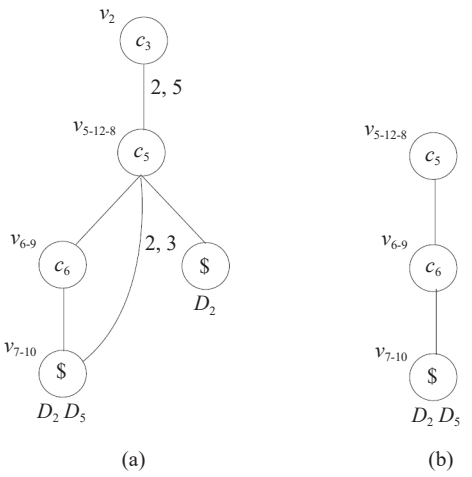

Figure 8. Illustration for the recursive execution of the algorithm

In the whole working process, a simple heuristic can be used to improve efficiency. Let α be the size of the largest subset of conjunctions found up to now, which can be satisfied by a certain truth assignment. Then, any recursive call of the algorithm over a smaller than α subset of p*-subgraphs will be suppressed.

After v_2 is checked, the next branching node is v_1, which will be handled in a way similar to v_3 and v_2.

4. Time complexity analysis

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

The first part, denoted as τ_1, is the cost to transform $C = C_1 \ldots C_n$ to $D = D_{11} \vee D_{12} \vee D_{21} \vee D_{22} \vee \ldots \vee D_{n1} \vee D_{n2}$. Obviously, τ_1 is bounded by O(n).

The second part, denoted as τ_2, is the time for computing the frequencies of variable appearances in D. Since in this process each variable in a D_i is accessed only once, $\tau_1 = \mathrm{O}(nm)$.

The third part, denoted as τ_3, is the time for constructing a trie-like graph G for D. This part of time can be further partitioned into three portions.

• τ_{31}: The time for sorting variable sequences for D_i's. It is obviously bound by O($nm\log_2 m$).

• τ_{32}: 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 the cost is bounded by O(nm^2).

• τ_{33}: The time for merging all p*-graphs to form a trie-like graph G, which is also bounded by O(nm^2).

The fourth part, denoted as τ_4, 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 $l = nm$. Assume that the average outdegree of a node in T is d. Then, the average time complexity of τ_4 can be characterized by the following recurrence:

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

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 boundaries, together with the cost for generating local trie-like subgraphs for each recursive call of the algorithm. We notice that the size of all the *RS*s together is bounded by the number of spans in G, which is $O(lm)$.

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

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

Solving this inequality, we will get

$$\begin{aligned}
\Gamma(l) &\le d \cdot \log_d l \cdot \Gamma\left(\frac{l}{d}\right) + O\left(l^2 m\right) \\
&\le d^2 \left(\log_d l\right)\left(\log_d \frac{l}{d}\right)\Gamma\left(\frac{l}{d^2}\right) + \left(\log_d l\right) l^2 m + l^2 m \\
&\le \ldots\ldots \\
&\le d^{\left\lceil \log_d^l \right\rceil}\left(\log_d l\right)\left(\log_d\left(\frac{l}{d}\right)\right)\ldots\left(\log_d \frac{l}{d^{\left\lceil \log_d^l \right\rceil}}\right) \\
&\quad + l^2 m\left(\left(\log_d l\right)\left(\log_d\left(\frac{l}{d}\right)\right)\ldots\left(\log_d \frac{l}{d^{\left\lceil \log_d^l \right\rceil}}\right) + \ldots + \log_d l + 1\right) \\
&\le O\left(l\left(\log_d l\right)^{\log_d l} + O\left(l^2 m\left(\log_d l\right)^{\log_d l}\right)\right. \\
&\sim O\left(l^2 m\left(\log_d l\right)^{\log_d l}\right).
\end{aligned} \tag{6}$$

Thus, the value for τ_4 is $\Gamma(l) \sim O(l^2 m(\log_d l)^{\log_d l})$.

From the above analysis, we have the following proposition.

Proposition 2 The average running time of our algorithm is bounded by

$$\begin{aligned}
\sum_{i=1}^{4} \tau &= O(n) + O(nm) + \left(O\left(nm \log_2 m\right) + 2 \times O\left(nm^2\right)\right) \\
&\quad + O\left(l^2 m\left(\log_d l\right)^{\log_d l}\right) \\
&\quad + O\left(n^2 m^3 \left(\log_d nm\right)^{\log_d nm}\right)
\end{aligned} \tag{7}$$

But we remark that if the average outdegree of a node in T is < 2, we can use a brute-force method to find the answer in polynomial time. Hence, we claim that the worst-case time complexity is bounded by $O(l^2 m(\log_2 l)^{\log_2 l})$ since

$(\log_d l)^{\log_d l}$ decreases as d increases.

5. Conclusions

In this paper, we have presented a new method to solve the 2-MAXSAT problem. The time complexity of the algorithm is bounded by $O(n^2m^3(\log_2 nm)^{\log_2 nm})$, 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, and d is the average outdegree of a node in a trie established over a set of conjunctions that are generated from the clauses in C. 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 at least n^* conjunctions satisfied by a truth assignment for U. To find a truth assignment that maximizes the number of satisfied conjunctions in D, a graph structure called p^*-graph is introduced to represent each conjunction in D. In this way, all the conjunctions in D can be represented as a trie-like graph. Searching G bottom up, we can find the answer efficiently.

In our future work, we will make a detailed analysis of the impact of the heuristics discussed in Section 3.2. It seems that by using the heuristics, any repeated recursive call can be effectively avoided. If that is the case, the number of recursive calls for each branching node will be bounded by $O(m)$ since the height of the trie-like graph G is bounded by $O(m)$. Thus, the worst-case time complexity of our algorithm should be bounded by $O(n^2m^4)$. It is because we have at most $O(nm)$ branching nodes, and for each recursive call, we need $O(nm^2)$ time to construct a dynamical trie. So, the total running time will be $O(nm) \times O(m) \times O(nm^2) = O(n^2m^4)$.

Conflict of interest

The author declares no competing financial interest.

References

[1] Cook SA. The complexity of theorem-proving procedures. In: *Proceedings of the 3rd Annual ACM Symposium on the Theory of Computing*. ACM Digital Library; 1971. p.151-158. Available from: https://doi.org/10.1145/800157.805047.

[2] Even Y, Itai A, Shamir A. On the complexity of timetable and multi-commodity flow problem. In: *16th Annual Symposium on Foundations of Computer Science (sfcs 1975)*. USA: IEEE; 1976. Available from: https://doi.org/10.1109/SFCS.1975.21.

[3] Johnson MS. Approximation algorithm for combinatorial problems. *Journal of Computer and System Sciences.* 1974; 9(3): 256-278. Available from: https://doi.org/10.1016/S0022-0000(74)80044-9.

[4] Garey MR, Johnson DS, Stockmeyer L. Some simplified NP-complete graph problems. *Theoretical Computer Science*. 1976; 1(3): 237-267. Available from: https://doi.org/10.1016/0304-3975(76)90059-1.

[5] Djenouri Y, Habbas Z, Djenouri D. Data mining-based decomposition for solving the MAXSAT problem: Toward a new approach. *IEEE Intelligent Systems.* 2017; 32(4): 48-58. Available from: https://doi.org/10.1109/MIS.2017.3121546.

[6] Kohli R, Krishnamurti R, Mirchandani P. The minimum satisfiability problem. *SIAM Journal on Discrete Mathematics*. 1994; 7(2): 275-283. Available from: https://doi.org/10.1137/S0895480191220836.

[7] Kügel A. Natural Max-SAT Encoding of Min-SAT. In: *Proceeding of the Learning and Intelligence Optimization Conference*. Paris, France: LION 6; 2012.

[8] Li CM, Zhu Z, Manya F, Simon L. Exact MINSAT solving. In: *Proceeding of 13th International Conference Theory and Application of Satisfiability Testing*. Edinburgh, UK; 2010. p.363-368.

[9] Li CM, Zhu Z, Manya F, Simon L. Optimizing with minimum satisfiability. *Artificial Intelligence*. 2012; 190: 32-44.

[10] Richard A. A graph-theoretic definition of a sociometric clique. *Journal of Mathematical Sociology*. 1974; 3(1): 113-126.
[11] Argelich J, Li CM, Manyà F, Zhu Z. MinSAT versus MaxSAT for optimization problems. In: Schulte C. (ed.) *Principles and practice of constraint programming*. Berlin: Springer; 2013. p.133-142. Available from: https://doi.org/10.1007/978-3-642-40627-0_13.
[12] Dumitrescu C. An algorithm for MAX2SAT. *International Journal of Scientific and Research Publications*. 2016; 6(12): 360-364. Available from: http://www.ijsrp.org/research-paper-1216.php?rp=P606089.
[13] Krentel MW. The complexity of optimization problems. *Journal of Computer and System Sciences*. 1988; 36(3): 490-509. Available from: https://doi.org/10.1016/0022-0000(88)90039-6.
[14] Papadimitriou C. *Computational complexity.* Addison-Wesley; 1994.
[15] Kemppainen E. *Incomplete MaxSAT solving by linear programming relaxation and rounding*. Master thesis. University of Helsinki; 2020.
[16] Xiao M. An exact MaxSAT algorithm: Further observations and further improvement. In: *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence (IJCAI-22)*. IJCAI; 2022. p.1887-1893. Available from: https://doi.org/10.24963/ijcai.2022/262.
[17] Zhang H, Shen H, Manyà F. Exact algorithms for MAX-SAT. *Electronic Notes in Theoretical Computer Science*. 2003; 86(1): 190-203.
[18] Impagliazzo R, Paturi R. On the complexity of k-SAT. *Journal of Computer and System Sciences*. 2001; 62(2): 367-375. Available from: https://doi.org/10.1006/jcss.2000.1727.
[19] Knuth DE. *The art of computer programming*. Vol. 1. Reading: Addison-Wesley; 1969.
[20] Knuth DE. *The art of computer programming*. Vol. 3. Reading: Addison-Wesley; 1975.