例題生成系の安全性について[†]

岩間 一雄　宮野 英次

九州大学工学部

NP完全問題等の複雑さの高い問題に対して，平均時間的には効率良く解くと主張されるアルゴリズムが数多く開発されている．それらの評価は通常解析的になされるが，実際的立場の研究者からは，実際に計算機実験によっても行なわれるべきだとの指摘がある．本稿では，充足可能問題（SAT）に対するこのようなアルゴリズムの評価に用いるべき例題生成はいかにあるべきかについて論じ，いくつかの例題生成アルゴリズムを示す．特に，出力例題の安全性，つまり例題生成アルゴリズムを悪用した解法の作りづらさ，について主に理論的側面から考察を加える．

On Securities of Instance-Generation Systems

Kazuo Iwama and Eiji Miyano

Department of Computer Science and Communication Engineering
Kyushu University

Fukuoka 812, Japan

For intractable problems such as NP-complete problems, a lot of algorithms, which are claimed to run fast on average, have been developed. Usually their performances have been demonstrated by mathematical analysis but, especially for practical purposes, it is also important to investigate them by actually running the algorithms. To this goal, we discuss how to generate instances of the satisfiability problem. In this paper, our main purpose is to discuss securities of such instance-generation systems. The system is "secure" if it is hard to generate efficient algorithms by exploiting the generation system.

[†]科学研究費 01302059，02302047，02650278 の補助を受けた．

1. Problems and Results

Suppose that there is a SAT algorithm M and we wish to test its performance experimentally, i.e., by measuring M's average run-time for a certain set I of instances. Note that M's answer to each instance (CNF predicate) is *yes* (a truth assignment satisfying the predicate exists) or *no* (no such assignments exit). When we construct such a set I, we usually mix *yes*-instances and *no*-instances in a somewhat intentional fashion, as we do so in giving decision problems to students. An instance generator (a deterministic algorithm) G is thus needed : The input to G consists of $answer \in \{yes, no\}$ and a binary string $r \in \{0,1\}^*$ and the output is a CNF predicate f. f must (not) be satisfiable if $answer$ is yes (no). G uses the string r to select one out of the many predicates whose satisfiability meets the given $answer$.

Desirable properties for the generator G should include : (i) G is fast : G is said to run in polynomial time if every instance f of length n is generated within $n^{O(1)}$ steps. (ii) G generates instances under a reasonable probability distribution. This condition will be discussed in future articles, not in this paper. (iii) G can generate a variety of instances. Let $I(G)$ ($\overline{I}(G)$) denote the set of *yes*-instances (*no*-instances) generated by G. If $I(G)$ and/or $\overline{I}(G)$ are small subsets of the whole *yes*- and/or *no*-instances, they might no longer keep the problem's intrinsic intractability. Two disjoint sets A and B is called *P-inseparable*[3], denoted by $A \simeq B$, if no set L with the property $A \subseteq L \subseteq \overline{B}$ is in P unless P=NP. G is said to be *secure* if $I(G) \simeq \overline{I}(G)$. We assume that the algorithm G is public. So, if $I(G) \not\simeq \overline{I}(G)$, i.e., if $I(G)$ and $\overline{I}(G)$ are P-separable, then one could cheatingly develop very fast SAT algorithms which works only for $I(G) \cup \overline{I}(G)$. It would be ideal if $I(G)$ and $\overline{I}(G)$ are the whole sets but one can think of its difficulty under the condition (i).

Our main objective is to seek such a generator G as above which runs in polynomial time and is *probably* secure by several reasons. It is not difficult to show that

• If G *is* secure then G is an almost perfect oneway function[2].

• If G runs in polynomial time then both $I(G)$ and $\overline{I}(G)$ are in NP.

• We can achieve the best possible $I(G)$, which is the set of all satisfiable CNF predicates (denoted by I_{sat}).

Therefore the problem is how close we can make $\overline{I}(G)$ to $I_{\overline{sat}}$, the set of all unsatisfiable predicates which is known to be co-NP complete. This appears to be equivalent to how hard we can make the set $\overline{I}(G)$. Since $\overline{I}(G)$ cannot surpass NP, what we can do seems to be limited; a typical possibility is to show that $\overline{I}(G)$ is NP-complete.

The result in this paper is slightly better: Our selection as $\overline{I}(G)$ is called a *provable* set, denoted by I_{prv}. Let $I_{\overline{prv}} = I_{\overline{sat}} - I_{prv}$. Then it is shown that $I_{sat} \simeq I_{\overline{prv}}$ and $I_{prv} \simeq I_{\overline{prv}}$. It should be noted that $I_{sat} \not\simeq I_{\overline{prv}}$ (and $I_{prv} \not\simeq I_{\overline{prv}}$ also) implies that $I_{sat} \not\simeq I_{prv}$, which means G is not secure. To prove the converse ($I_{sat} \not\simeq I_{prv}$ implies $I_{sat} \not\simeq I_{\overline{prv}}$ or $I_{prv} \not\simeq I_{\overline{prv}}$) seems to be extremely hard. If we could, then it implies that G is secure. The symbol $\simeq$ is adopted in order to suggest "is similar to" and similarity is transitive. Due to this intuition, $I_{sat} \simeq I_{\overline{prv}}$ and $I_{prv} \simeq I_{\overline{prv}}$ implies I_{sat} and I_{prv} are "similar", or hard to distinguish. As for $I(G)$, it is perfect, i.e., we can achieve $I(G) = I_{sat}$.

2. Backgrounds

Many algorithms have been developed for hard (typically NP-complete) combinatorial problems, which are claimed to run fast on average (see [7][11] for SAT). However, the analysis of their average complexities usually involves complicated mathematics, which makes it difficult to convince possible users of those algorithms. Experimental tests must have merits in this sense. The problem has been how to generate the proper input on which the algorithm is actually run.

For example, researchers in the field of logic design commonly use a fixed bench-mark set of instances, which clearly involves such a danger that algorithms can be tuned up to those fixed instances. Generating instances (as a string) at purely random is also common, which prevents the above cheating. However, it includes other defects that might be more serious: The pure random method of course generates only the instances that follow a certain mathematical (truly even) distribution. In the case of SAT, after evaluating several easy parameters, such as the numbers of variables, clauses and literals in each clause, one can guess the answer almost always correctly if he knows that the instance is purely random. This could reduce a lot of both practical and theoretical interests. Another problem is that we cannot know the answer of each generated instance (unless we actually solve it). That means it is hard to check the answer of the algorithm under test, which could again incite serious cheating.

Thus we are naturally led to the current way of generating instances. It has another significant aspect, namely, applications of the theory of oneway functions. Oneway functions are widely recognized as important in developing cryptosystems[9][12][6] but in almost no there areas. The instance generation seems to be a simpler and more direct application. It does not need the tricky trapdoor systems[13] or the strong pseudo-random sequences[1][10][4]. The notion of uniformity[8][5] is not so serious either, since we do not have to produce always hard instances. All we have to do is, if simply stated, to find, out of a co-NP set, an NP subset that is as large as or as hard as possible.

3. Generation Algorithms

A *generator* is a Turing machine M. The input to M is a binary sequence $r \in \{0,1\}^*$. M has the output tape and the contents on it after M halts, $M(r)$, is called an output of M. M must halt for all input sequences. A language L (a set of instances) is said to be *generated* by M if $L = \{M(r) \mid r \in \{0,1\}^*\}$. M is said to be $T(n)$ time-bounded if for every input sequence an output y is produced within $T(|y|)$ steps.

Note that the input of the generator under this definition does not include the answer of the output string (instance). We regard that the generator in the sense of Sec.2 consists of two (independent) generators under the new definition, one for *yes*-instance, and the other for *no*-instances. The input r may be generated inside M at random. Note that M must produce some output string (instance) whatever r is given (or generated as a random sequence).

A *literal* is a logic variable x or its negation $\overline{x}$. A *clause* is a sum of literals. A (*CNF*) *predicate* is a product of clauses. A specific assignment of *true* and *false* to all the logic variables is sometimes called a *cell*. It is said that a clause A *covers* a cell T if the assignment denoted by T makes A *false*. A clause A is said to *cover* a clause B if all the cell covered by B are covered by A. (Intuitively, A is larger than B.) The following four operations for clauses appear frequently : A clause A is *splited into* $(A+x)(A+\overline{x})$ *using* variable x. Its inverse operation is called *merge*. A literal x' is *added* to a clause A, which results in a smaller clause $(A+x')$. Also a literal x' is *removed* from $(A+x')$, which makes the literal larger.

Now we present a generator for all satisfiable CNF predicates:

Generator SAT-GEN

Step1. A single cell T_{ans} is created at random. ("at random" means using the input string appropriately.)

Step2. A clause T is also created at random.

Step3. If T does not cover T_{ans} then T is included in the instance. Otherwise, a single literal x' consisting T is selected at random and it is changed to its negation, i.e., $\overline{x}$ if $x' = x$ and x if $x' = \overline{x}$. Such a T' is also included in the instance.

Step4. Steps 2 and 3 are repeated until the input string has been spent.

Theorem 1. SAT-GEN generates I_{sat}.

Proof. Obvious. T_{ans} is at least one satisfactory truth assignment. □

SAT-GEN clearly runs in polynomial time. If we can take time, we can also generate the whole $I_{\overline{sat}}$.

Generator $\overline{\text{SAT}}$-GEN

Step1. Select a variable x at random and let $f_{now} = x\overline{x}$.

Step2. One of the following (2-1) to (2-4) is randomly chosen and executed:

(2-1) Select a clause A in f_{now} and select a variable x both at random. f_{now} is modified by splitting A into $(A + x)(A + \overline{x})$.

(2-2) Select a clause A in f_{now} and a literal x' in A at random. f_{now} is modified by removing x' from A.

(2-3) Select, again in some random fashion, a pair of clauses A and B in f_{now} such that A covers B (if any). We remove the (smaller) clause B from f_{now}.

(2-4) Construct a random clause A and add A into f_{now}.

Step3. Repeat Step2 until the input string runs out and halt with f_{now} at that time on the output tape.

Theorem 2. $\overline{\text{SAT}}$-GEN generates $I_{\overline{sat}}$.

Proof. We can use a common principle called iterated consensus for generating prime implicants. Details are omitted. □

One can see that $\overline{\text{SAT}}$-GEN does not run in polynomial time. To modify it so as to run in polynomial time is easy:

Generator PRV-GEN

Everything is the same as $\overline{\text{SAT}}$-GEN but (2-3) is removed.

Now we define I_{prv}. Consider the following two operations: (i) A literal is added to a clause A. (ii) Two clauses $(A + x)(A + \overline{x})$ is merged into A. A predicate f is called *provable* if, for some variable x, $x\overline{x}$ is implied by applying above (i) and (ii) repeatedly. I_{prv} is the set of all provable predicates.

Theorem 3. PRV-GEN generates I_{prv}.

Proof. Obvious. □

Recall that our goal is to claim that I_{prv} is reasonably large or reasonably hard within the limit of NP. Here are now our main theorems.

Theorem 4. I_{prv} and $I_{\overline{prv}}$ are P-inseparable.

Theorem 5. I_{sat} and $I_{\overline{prv}}$ are P-inseparable.

We prove Theorem 4 in the next section. Proof of Theorem 5 is easier than Theorem 4 and may be omitted. Note that the NP-completeness of I_{prv} is an immediate corollary of Theorem4. Namely, it is at least hard to show the intractability of the given instance simply by reversing PRV-GEN. Although details are omitted, the negation of Theorem 4 (and Theorem 5) would imply the fact that I_{sat} and I_{prv} are P-separable. There are several open questions:

(1) Are I_{sat} and I_{prv} P-inseparable?

(2) Are there similar generators for other problems such as the Hamilton circuit problem?

We are now developing actual instance generators based on SAT-GEN and $\overline{\text{SAT}}$-GEN. Those generators accept, as their input, (i) the number of variables, (ii) the number of clauses and (iii) the probability that each literal appears in a clause. They produces *yes*- and *no*-instances that meet the input values (if possible) using a random sequence generated inside. It is not so easy to adjust the output so as to meet the parameter values, which will be reported in future papers.

4. Proof of Theorem 4

Suppose that f is an instance of the 3SAT problem. We describe how to construct a (alway unsatisfiable) CNF predicate g from f such that g is provable if f is satisfiable and g is not provable otherwise. Roughly speaking, g uses a special variable σ and $\sigma\overline{\sigma}$ is the only one possibility implied by the literal-addition and merge operations.

For simplicity of description, we take the following example as f

$$f = (\overline{\alpha_1} + \overline{\alpha_2} + \alpha_3)(\overline{\alpha_1} + \alpha_2 + \alpha_4)$$

$$(\alpha_1 + \overline{\alpha_4} + \alpha_5)(\alpha_2 + \alpha_3 + \overline{\alpha_5})$$
$$(\alpha_3 + \alpha_4 + \overline{\alpha_5})(\overline{\alpha_2} + \overline{\alpha_3} + \alpha_5)$$
$$(\alpha_1 + \overline{\alpha_2} + \alpha_4)(\alpha_2 + \alpha_3 + \alpha_5)$$

The clauses appearing in g are divided into four groups, G_1 through G_4. G_1 consists of the following single clause:

$$\begin{aligned}(&x_1 + x_2 + x_3 + x_4 + x_5 + y_1 + y_2 + y_3 + y_4 + y_5\\ &+\overline{z_{11}} + \overline{z_{12}} + \overline{z_{13}} + \overline{z_{17}} + \overline{z_{21}} + \overline{z_{22}}\\ &+\overline{z_{24}} + \overline{z_{26}} + \overline{z_{27}} + \overline{z_{28}} + \overline{z_{31}} + \overline{z_{34}}\\ &+\overline{z_{35}} + \overline{z_{36}} + \overline{z_{38}} + \overline{z_{41}} + \overline{z_{42}} + \overline{z_{45}}\\ &+\overline{z_{47}} + \overline{z_{53}} + \overline{z_{54}} + \overline{z_{55}} + \overline{z_{56}} + \overline{z_{58}}\\ &+a_1 + a_2 + a_3 + a_4 + a_5 + a_6 + a_7 + a_8 + \overline{\sigma})\end{aligned}$$

Associated with the five variables in f, we prepare x_1 through x_5, y_1 through y_5. a_1 through a_8 are introduced since f includes eight clauses. As for z_{ij}, we need, for example, four z_{1j} $(z_{11}, z_{12}, z_{13}, z_{17})$ since the first variable α_1 appears in the first, second, third and seventh clauses of f. As mentioned before, $\overline{\sigma}$ plays an important role.

The second group G_2 is as follows

$$\begin{aligned}&(\overline{x_1} + \sigma)(\overline{x_2} + \sigma) \cdots (\overline{x_5} + \sigma)(\overline{y_1} + \sigma)(\overline{y_2} + \sigma) \cdots (\overline{y_5} + \sigma)\\ &(z_{11} + \sigma)(z_{12} + \sigma)(z_{13} + \sigma)(z_{17} + \sigma)(z_{21} + \sigma) \cdots (z_{58} + \sigma) \quad (1)\end{aligned}$$

The following observation will be helpful: Suppose hypothetically that all a_1 to a_8 are removed from G_1. Then, using the clauses in G_2, G_1 can be reduced into $\overline{\sigma}$. For example, x_1 of G_1 can be removed by adding literals $x_2, x_3, \cdots, \overline{z_{58}}$ to $(\overline{x_1} + \sigma)$ first and then merging it with G_1.

The third group G_3 of literals are prepared to delete a_1 through a_8 as mentioned above. Associated with $\alpha_1, \overline{\alpha_1}, \cdots, \alpha_5, \overline{\alpha_5}$ of f, the following clauses are prepared:

$$\begin{aligned}\alpha_1 \ &: \ (U(1) + \sigma + z_{13} + z_{17})(x_1 + \overline{z_{13}} + \overline{a_3})\\ &\quad (x_1 + \overline{z_{17}} + \overline{a_7}) && (2)\\ \overline{\alpha_1} \ &: \ (U(1) + \sigma + z_{11} + z_{12})(y_1 + \overline{z_{11}} + \overline{a_1})\\ &\quad (y_1 + \overline{z_{12}} + \overline{a_2}) && (3)\\ \alpha_2 \ &: \ (U(2) + \sigma + z_{22} + z_{24} + z_{28})(x_2 + \overline{z_{22}} + \overline{a_2})\\ &\quad (x_2 + \overline{z_{24}} + \overline{a_4})(x_2 + \overline{z_{28}} + \overline{a_8}) && (4)\\ \overline{\alpha_2} \ &: \ (U(2) + \sigma + z_{21} + z_{26} + z_{27})(y_2 + \overline{z_{21}} + \overline{a_1})\\ &\quad (y_2 + \overline{z_{26}} + \overline{a_6})(y_2 + \overline{z_{27}} + \overline{a_7}) && (5)\\ \alpha_3 \ &: \ (U(3) + \sigma + z_{31} + z_{34} + z_{35} + z_{38})(x_3 + \overline{z_{31}} + \overline{a_1})\\ &\quad (x_3 + \overline{z_{34}} + \overline{a_4})(x_3 + \overline{z_{35}} + \overline{a_5})(x_3 + \overline{z_{38}} + \overline{a_8}) && (6)\\ \overline{\alpha_3} \ &: \ (U(3) + \sigma + z_{36})(y_3 + \overline{z_{36}} + \overline{a_6}) && (7)\\ \alpha_4 \ &: \ (U(4) + \sigma + z_{42} + z_{45} + z_{47})(x_4 + \overline{z_{42}} + \overline{a_2})\\ &\quad (x_4 + \overline{z_{45}} + \overline{a_5})(x_4 + \overline{z_{47}} + \overline{a_7}) && (8)\\ \overline{\alpha_4} \ &: \ (U(4) + \sigma + z_{43})(y_4 + \overline{z_{43}} + \overline{a_3}) && (9)\\ \alpha_5 \ &: \ (U(5) + \sigma + z_{53} + z_{56} + z_{58})(x_5 + \overline{z_{53}} + \overline{a_3})\\ &\quad (x_5 + \overline{z_{56}} + \overline{a_6})(x_5 + \overline{z_{58}} + \overline{a_8}) && (10)\\ \overline{\alpha_5} \ &: \ (U(5) + \sigma + z_{54} + z_{55})(y_5 + \overline{z_{54}} + \overline{a_4})\\ &\quad (y_5 + \overline{z_{55}} + \overline{a_5}) && (11)\end{aligned}$$

Take a look at, for example, predicate (4). $(x_2 + \overline{z_{24}} + \overline{a_4})$ means that α_2 appears in the forth clause of f. $(y_2 + \overline{z_{26}} + \overline{a_6})$ in (5) means $\overline{\alpha_2}$ appears in the sixth clause of f. Thus x_i corresponds to α_i and y_i to $\overline{\alpha_i}$.

$U(1), U(2), \cdots, U(k-1), U(k)$ are clauses

$$u_1, \ \ \overline{u_1} + u_2, \ \cdots, \ \ \overline{u_1} + \overline{u_2} + \cdots + \overline{u_{(k-2)}} + u_{(k-1)},$$
$$\overline{u_1} + \overline{u_2} + \cdots + \overline{u_{(k-2)}} + \overline{u_{(k-1)}},$$

respectively. Namely,

$$(F + U(1))(F + U(2)) \cdots (F + U(k)) \qquad (12)$$

can be modified to F by repeated merge operations.

One can see that

$$(U(1) + x_1 + \overline{a_3} + \overline{a_7} + \sigma) \ \text{ and } \ (U(1) + y_1 + \overline{a_1} + \overline{a_2} + \sigma) \qquad (13)$$

can be implied from predicates (2) and (3), respectively. Now we introduce the following forth group G_4 of clauses:

$$\begin{aligned}&(U(1) + \overline{x_1} + \sigma)(U(1) + \overline{y_1} + \sigma)(U(1) + a_1 + \sigma)\\ &(U(1) + a_2 + \sigma)(U(1) + a_3 + \sigma)(U(1) + a_7 + \sigma) \quad (14)\end{aligned}$$

Using those clauses, two predicates in (13) are both reduced into $(U(1) + \sigma)$. The rest of G_4 is similar. For example,

$$(U(2) + \overline{x_2} + \sigma)(U(2) + \overline{y_2} + \sigma)$$

$$(U(2)+a_1+\sigma)(U(2)+a_2+\sigma)$$
$$(U(2)+a_4+\sigma)(U(2)+a_6+\sigma)$$
$$(U(2)+a_7+\sigma)(U(2)+a_8+\sigma)$$

are prepared for (4) and (5). Then both are reduced into

$$(U(2)+\sigma)$$

Thus G_3 are finally reduced to

$$(U(1)+\sigma)(U(2)+\sigma)(U(3)+\sigma)(U(4)+\sigma)(U(5)+\sigma)(15)$$

which is then reduced to σ by the property (12).

Here is another important observation: Recall that a_1 through a_8 are deleted using the clauses in G_3. Suppose that a_1 is deleted using $(y_2+\overline{z_{21}}+\overline{a_1})$. Then our interpretation is that the first (denoted by a_1) clause of f is satisfied by letting $\overline{\alpha_2}$ (denoted by y_2) be true, i.e., α_2 be false. Note that once $(y_2+\overline{z_{21}}+\overline{a_1})$ is used to delete a_1 then that clause disappears. Then z_{21} in the first clause of (5) can no longer be deleted and therefore $(U(2)+\sigma)$ cannot be implied from (5). If, for example, both $(y_2+\overline{z_{21}}+\overline{a_1})$ and $(x_2+\overline{z_{24}}+\overline{a_4})$ are used for this purpose, then $(U(2)+\sigma)$ cannot be implied either from (4) or (5). Namely, we can no longer imply (15) or σ. However, one can see that such a operation does not make sense under our current interpretation described above, since it means that we assigned true to both α_2 and $\overline{\alpha_2}$.

Suppose that f is satisfiable. Then there is a satisfiable truth assignment for which the confusion described above does not occur. That means we can delete all a_1 through a_8 and at the same time we can leave at least one of (2) and (3) as it was originally and similarly for (4) and (5), (6) and (7), and so on. To conclude:

Lemma 1. If f is satisfiable then g is provable.

Recall that g must be unsatisfiable whether or not f is satisfiable. To prove this is not hard: If we duplicate each clause in G_3 into two or more same clauses, then every clause in, e.g., (2) can survive. Since the duplication of clauses does not change the satisfiability of the predicate and since a predicate is clearly unsatisfiable if it is provable:

Lemma 2. g is unsatisfiable whether or not f is satisfiable.

Now the rest of the proof is to show that if f is not satisfiable than g is not provable. Two technical lemmas are presented first:

Lemma 3. Suppose that x' (a clause consisting of a single literal) is implied from a predicate D. Then, if x or $\overline{x}$ appears nowhere in D, D is unsatisfiable.

Proof. Similarly as in the case of clauses, a predicate A is said to cover a predicate B if A becomes false for all the assignments that make B false. One can see easily that if a predicate A is modified to B by the addition or merge operation then A covers B. Thus, if x' is implied from D, D covers x', which means D must be false for all the assignments making x' false. Since D does not include x or $\overline{x}$, it must be false for all possible assignments. □

Lemma 4. Suppose that $E_1,\cdots,E_m$ and $F_1,\cdots,F_n$ are clauses that do not include a literal x or $\overline{x}$. Then if x is implied from $(E_1+x)(E_2+x)\cdots(E_m+x)(F_1)(F_2)\cdots(F_n)$, then the predicate $E_1E_2\cdots E_mF_1F_2\cdots F_n$ must be unsatisfiable.

Proof. Similarly as Lemma 3. □

Now suppose that f is not satisfiable. We first focus our attention only to the special variable σ and prove $\sigma\overline{\sigma}$ cannot be implied.

Let G_{31} be a set of clauses in G_3 that do not include σ or $\overline{\sigma}$. Then we can claim by Lemma 3 that σ (or $\overline{\sigma}$) cannot be implied from G_{31}, since G_{31} is satisfied by assigning $x_1=\cdots=x_5=y_1=\cdots=y_5=\mathit{true}$. Therefore, if $\overline{\sigma}$ could be implied, then it could be done so from $G_1\cup G_2\cup G_{31}$. (Any clause including σ clearly does not help.) Now one can see, by Lemma 4, that in order to imply σ we need all the clauses in $G_1\cup G_2$ and at least eight clauses including each of $\overline{a_1}$ through $\overline{a_8}$. A crucial point is, as mentioned before, that if f is not satisfiable, we can no way select those eight clauses so that the rest of G_{31} will include complete (2) or (3), complete (4) or (5), and so on. Now we have the following lemma.

Lemma 5. If two clauses, one in (2) and the other in (3) or one in (4) and the other in (5) or $\cdots$, are removed, then $G_3 \cup G_4$ is satisfiable.

Proof. Suppose, for example, that the third clause in (2) and the second clause in (3) are removed. Then, $z_{11} = z_{17} = 1$ satisfies the first clauses of both (2) and (3). $z_{12} = z_{13} = 0$ satisfies all the other clauses in (2) and (3). Furthermore, $u_1 = 0$ satisfies all the clauses including $U(2), \cdots, U(5)$. Finally, $x_1 = y_1 = 0$ and $x_2 = \cdots = x_5 = y_2 = \cdots = y_5 = 1$ satisfies all the other clauses. □

Thus we can conclude that if $\overline{\sigma}$ is implied then σ cannot. Now here is our final lemma.

Lemma 6. If $\sigma\overline{\sigma}$ cannot be implied, then $x\overline{x}$ cannot be implied either for any variable x other than σ.

Proof. The complete proof is tedious. We only give a short observation on variable x_1. Similarly as before, the set of clauses not including x or $\overline{x}$ is satisfiable. It should be noted that $\overline{x_1}$ appears only once in the whole g, i.e., $(\overline{x_1} + \overline{\sigma})$ in G_2. Then one can see that in order to imply $\overline{x_1}$, we essentially need to imply σ exactly as before. After that, it is not so hard to show that x_1 can never be implied from the rest of clauses. □

5. Proof of Theorem 5

Proof. Suppose that A is an instance of the SAT problem. We describe how to construct a CNF predicate B from A such that B is satisfiable if A is satisfiable and B is unsatisfiable and not provable otherwise. The idea is to construct B using a simple unprovable predicate.

Let X_i be a CNF predicate such that:

$$\begin{aligned} X_i &= X_{i1}X_{i2}\cdots X_{i8} \\ &= (x_{i1} + x_{i3} + x_{i4})(\overline{x_{i1}} + \overline{x_{i2}} + x_{i3}) \\ &\quad (x_{i2} + x_{i3} + \overline{x_{i4}})(x_{i1} + \overline{x_{i2}} + \overline{x_{i4}}) \\ &\quad (\overline{x_{i1}} + \overline{x_{i3}} + \overline{x_{i4}})(x_{i1} + x_{i2} + \overline{x_{i3}}) \\ &\quad (\overline{x_{i2}} + \overline{x_{i3}} + x_{i4})(\overline{x_{i1}} + x_{i2} + x_{i4}) \end{aligned}$$

Lemma 6. X_i is unsatisfiable and unprovable.

Proof. It is obvious that X_i is unsatisfiable. We prove that X_i is not provable. Take a look at arbitrary two clauses. Then it turns out that the relations of those two clauses meets one of the following conditions: (a) All literals appearing in one clause are completely different from the literals of the other, like $(x_{i1} + x_{i3} + x_{i4})$ and $(\overline{x_{i1}} + \overline{x_{i3}} + \overline{x_{i4}})$. (b) There is a variable x such that one clause includes x and the other clause includes $\overline{x}$, and further more each clause includes different variables, like $(x_{i1} + x_{i3} + x_{i4})$ and $(\overline{x_{i1}} + \overline{x_{i2}} + x_{i3})$. In the case of (a), the two clauses cannot be merged directly even if any literals are added. In the other case, some literals must be added to each clause in order to merge the two clauses. However, if a literal is added to any clauses of X_i, then X_i becomes a satisfiable predicate since every cell is covered by only one clause. □

Suppose that the given instance A is $A_1A_2\cdots A_n$, where A_i is the ith clause of A. Now B can be constructed using the above unprovable predicate X_i as follows:

$$\begin{aligned} B &= (A_1 + X_{11})(A_1 + X_{12})\cdots(A_1 + X_{18}) \\ &\quad (A_2 + X_{21})(A_2 + X_{22})\cdots(A_2 + X_{28}) \\ &\quad \vdots \\ &\quad (A_n + X_{n1})(A_n + X_{n2})\cdots(A_n + X_{n8}) \end{aligned}$$

Let Φ and Ψ_i be the set of all variables appearing in A and X_i, respectively. Note that $\Phi \cap \Psi_i = \emptyset$ and $\Psi_i \cap \Psi_j = \emptyset$ $(i \neq j)$. One can see that a satisfiable truth assignment for A also satisfies B, i.e., B is satisfiable if A is so.

Lemma 7. Suppose that $F_1, \cdots, F_m$ are clauses whose variables belong to a variable set F and $G_1, \cdots, G_m$ are the same whose variable set is G. Suppose also that $F \cap G = \emptyset$. Then if the predicate $(F_1+G_1)(F_2+G_2)\cdots(F_m+G_m)$ is provable then both predicates $(F_1)(F_2)\cdots(F_m)$ and $(G_1)(G_2)\cdots(G_m)$ must be provable.

Proof. Suppose that a clause $(F_{ij} + G_{ij})$ is implied from $(F_i + G_i)$ and $(F_j + G_j)$by applying the two operations (the addition and the merge operation).Recall that variables in F_{ij} and those in G_{ij} are elements of the set F and G respectively. Taking $F \cap G = \emptyset$ into consideration, it can be regarded that the clause $(F_{ij} + G_{ij})$ is implied by the merge operation after adding some literals in F to F_i and F_j and adding those in G to G_i and G_j. For example, the two operations are applied to $(f_1+f_2+f_3+g_1+g_2+g_3)$ and $(\overline{f_3}+f_4+g_3+g_4)$ ($f_i \in F$, $g_i \in G$) as follows:

$$\begin{aligned}
& (f_1 + f_2 + f_3 + g_1 + g_2 + g_3)(\overline{f_3} + f_4 + g_3 + g_4) \\
\to\ & (f_1 + f_2 + f_3 + f_4 + g_1 + g_2 + g_3 + g_4) \\
& (f_1 + f_2 + \overline{f_3} + f_4 + g_1 + g_2 + g_3 + g_4) \\
\to\ & (f_1 + f_2 + f_4 + g_1 + g_2 + g_3 + g_4)
\end{aligned}$$

However, this implication is simulated by the following independent implications:

$$\begin{aligned}
& (f_1 + f_2 + f_3)(\overline{f_3} + f_4) \\
\to\ & (f_1 + f_2 + f_3 + f_4)(f_1 + f_2 + \overline{f_3} + f_4) \\
\to\ & (f_1 + f_2 + f_4) \\
& (g_1 + g_2 + g_3)(g_3 + g_4) \\
\to\ & (g_1 + g_2 + g_3 + g_4 + g_5)(g_1 + g_2 + g_3 + g_4 + \overline{g_5}) \\
\to\ & (g_1 + g_2 + g_3 + g_4)
\end{aligned}$$

Thus, one can see that the process of implying ϕ from $(F_1 + G_1)(F_2 + G_2) \cdots (F_m + G_m)$ can be simulated by a sequence of above implications in a step-by-step fashion. Now we can conclude that ϕ can be implied both from $(F_1)(F_2) \cdots (F_m)$ and from $(G_1)(G_2) \cdots (G_m)$. □

One can see that B is unsatisfiable if A is unsatisfiable, since X_i is so. Then we can claim by Lemma 6 and 7 that B is unprovable if A is unsatisfiable. □

参考文献

[1] M. Blum and S. Micali. How to generate cryptographically strong sequences of pseudo-random bits. *SIAM J. Comput.*, Vol. 13, No. 4, pp. 850–864, 1984.

[2] O. Goldreich and L. Levin. A hard-core predicate for all one-way functions. In *Proc. 21st ACM Symposium on Theory of Computing*, pp. 25–32, 1989.

[3] J. Grollmann and A. Selman. Complexity measures for public-key cryptosystems. *SIAM J. Comput.*, Vol. 17, No. 2, pp. 309–335, 1988.

[4] J. Hastad. Pseudo-random generators under uniform assumptions. In *Proc. 22nd ACM Symposium on Theory of Computing*, pp. 395–415, 1990.

[5] R. Impagliazzo and L. Levin. No better ways to generate hard np instances than picking uniformly at random. In *Proc. 31st IEEE Symp. on Foundations of Computer Science*, pp. 812–821, 1990.

[6] R. Impagliazzo, L. Levin, and M. Luby. Pseudo-random generation from one-way functions. In *Proc. 21st ACM Symposium on Theory of Computing*, pp. 12–24, 1989.

[7] K. Iwama. Cnf satisfiability test by counting and polynomial average time. *SIAM J. Comput.*, pp. 385–391, 1989.

[8] L. Levin. Average case complete problems. *SIAM J. Comput.*, Vol. 15, pp. 285–286, 1986.

[9] R. Merkle and M. Hellman. Hiding information and signatures in trapdoor knapsacks. *IEEE Trans. Imform. Theory*, Vol. IT-24, pp. 525–530, 1978.

[10] N. Nisan and A. Wigderson. Hardness vs. randomness. In *Proc. 29th IEEE Symp. on Foundations of Computer Science*, pp. 2–12, 1988.

[11] P. Purdom and C. Brown. The pure literal rule and polynomial average time. *SIAM J. Comput.*, pp. 943–953, 1985.

[12] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Comm. ACM*, Vol. 21, pp. 120–126, 1978.

[13] A. C. Yao. Theory and applications of trapdoor functions. In *Proc. 23rd IEEE Symp. on Foundations of Computer Science*, pp. 80–91, 1982.