PDMI PREPRINT — 20/2004

AUTOMATED GENERATION OF SIMPLIFICATION RULES FOR SAT AND MAXSAT[1]

Alexander S. KULIKOV
St. Petersburg State University
Department of Mathematics and Mechanics
St. Petersburg
Russia

`http://logic.pdmi.ras.ru/~kulikov`

December 22, 2004

Abstract

Currently best known upper bounds for many NP-hard problems are obtained by using divide-and-conquer (splitting) algorithms. Roughly speaking, there are two ways of splitting algorithm improvement: a more involved case analysis and an introduction of a new simplification rule. It is clear that case analysis can be executed by computer, so it was considered as a machine task. Recently, several programs for automated case analysis were implemented. However, designing a new simplification rule is usually considered as a human task. In this paper we show that designing simplification rules can also be automated. We present several new (previously unknown) automatically generated simplification rules for the SAT and MAXSAT problems. The new approach allows not only to generate simplification rules, but also to find good splittings. To illustrate our technique we present a new algorithm for $(n,3)$-MAXSAT that uses both splittings and simplification rules based on our approach and has worst-case running time $O(1.272021^N L)$, where N is the number of variables and L is the length of an input formula. This bound improves the previously known bound $O(1.324719^N L)$ of Bansal and Raman.

[1]Supported in part by grant No. 1 of the 6th RAS contest-expertise of young scientist projects (1999) and Grant #NSh-2203-2003-1 of the President of Russian Federation for Leading Scientific School Support.

1 Introduction

The *splitting method* (i.e., estimating the complexity of divide-and-conquer algorithms by recurrent inequalities) is a powerful tool for proving upper bounds for NP-hard problems. Currently best known upper bounds for many NP-hard problems are obtained by using exactly this method (SAT [10], MAXSAT [3], XSAT [1], MAXCUT [6], Vertex Cover [8]). Formally, a splitting algorithm splits an input instance of a problem into several simpler instances further simplified by certain simplification rules, such that by finding the solution for all of them one can construct the solution for the initial instance in polynomial time.

In general, there are two ways of splitting algorithm improvement: a more involved case analysis and an introduction of a new simplification rule (in Sect. 4 we give examples of both these ways). In [7] we showed how case analysis can be automated. In this paper we present a procedure for generating simplification rules for the SAT and MAXSAT problems. This procedure is capable to generate all known simplification rules that modify input formula by means of assigning a value to a variable (such as, for example, pure literal, dominating unit clause) and also several new (previously unknown) rules.

Our approach is based on defining a partial ordering on assignments to variables of a formula. This approach allows us not only to generate simplification rules, but also to find good splittings. In the end of the paper we present a new algorithm for $(n, 3)$-MAXSAT that uses both splittings and simplification rules based on our approach. The automated proof of the running time of this algorithm is available at `http://logic.pdmi.ras.ru/~kulikov`.

Motivation. We have already mentioned that one of the two main ways of splitting algorithm improvement is an introduction of a new simplification rule to it. This fact already motivates the need for designing new simplification rules. Also sometimes a new simplification rule does not allow to prove a better upper bound on the running time of a splitting algorithm, but it allows to significantly reduce the case analysis included in this algorithm. In general, a new simplification rule for, say, the SAT problem can improve the running time of a certain SAT solver, not necessarily based on the splitting method.

Organization of the paper. In Sect. 2 we give basic definitions. Section 3 contains the list of all known simplification rules that modify an input formula by means of assigning a value to a literal or replacing a literal by another literal. In Sect. 4 two examples of splitting algorithm improvement are given. In Sect. 5 we present new simplification rules. Section 6 presents a new algorithm for $(n, 3)$-MAXSAT, which uses both simplifications and splittings based on the approach of the paper. Finally, Sect. 7 indicates several further directions.

2 General Setting

2.1 Basic Definitions

Let V be a set of Boolean variables. The negation of a variable v is denoted by $\bar{v}$, by $\bar{V}$ we denote the set $\{\bar{v} \mid v \in V\}$. *Literals* are the members of the set $W = V \cup \bar{V}$. If w denotes a literal $\bar{l}$, then $\bar{w}$ denotes a literal l. A *clause* is the disjunction of a finite set of literals that does not contain simultaneously any variable together with its negation. The empty clause is interpreted as *False*. A *formula in CNF* is the conjunction of a finite set of clauses. The empty formula is interpreted as

True. It is often conveniently to consider a clause simply as a set of literals and a formula as a set of clauses. The *length of a clause* is the number of its literals. The *length of a formula* is the sum of the lengths of all its clauses. We say that *a literal l occurs* in a clause or in a formula, if this clause or this formula contains the literal l. However, we say that *a variable v occurs* in a clause or in a formula, if this clause or this formula contains the literal v, or it contains the literal $\bar{v}$. An *(i,j)-literal* is a literal occurring in a formula i times positively and j times negatively.

An *assignment* is a finite subset of W that does not contain any variable together with its negation. Informally speaking, if an assignment I contains a literal l, it means that l has the value *True* in I. To obtain a formula $F[I]$ from a formula F and an assignment $I = \{l_1, \ldots, l_s\}$, we first remove from F all clauses containing the literals l_i, and then delete all occurrences of the literals $\bar{l_i}$ from the other clauses.

2.2 The SAT and MAXSAT Problems

The SAT problem is: given a formula in CNF, check whether this formula is satisfiable, i.e., whether there exists an assignment of Boolean values to all variables of the formula that makes it *True*. For example, the formula $(xy)(\bar{x})$ is obviously satisfiable, while the formula $(xy)(\bar{x})(x\bar{y})$ is not. In the MAXSAT problem one is asked to find a maximal possible number of simultaneously satisfied clauses of a given formula. For example, this number is equal to 3 for the formula $(xy)(\bar{x}\bar{y})(\bar{x}y)(x\bar{y})$.

Both SAT and MAXSAT are very well-known NP-hard problems, that have been attacked by scientists from all over the world for a long time. The currently best known upper bounds for SAT are $O(1.238823^K)$ and $O(1.073997^L)$ [10], and for MAXSAT are $O(1.341294^K)$ [3], $O(1.105729^L)$ [2], where K denotes the number of clauses in the input formula, L denotes the length of the input formula. Note that there are no known upper bounds of the form $O(c^N)$, where $c < 2$ is a constant and N is the number of variables in the input formula, for these problems.

2.3 Estimation of the Size of a Splitting Tree

Kullmann and Luckhardt introduced in [11] a notion of a splitting tree. One can consider an execution of a splitting algorithm as a tree whose nodes are labeled with CNF-formulas such that if a node is labeled with a CNF-formula F, then its sons are labeled with simplified formulas $F[I_1], F[I_2], \ldots, F[I_k]$ for assignments $I_1, I_2, \ldots, I_k$. To each formula F of this tree we attach a non-negative integer $\mu(F)$, which denotes the *complexity of F*. We use the following measures of complexity in our framework:

1. $\mu(F) = N(F)$ is the number of variables in F;
2. $\mu(F) = K(F)$ is the number of clauses in F;
3. $\mu(F) = L(F)$ is the length of F.

The tree is a *splitting tree* if, for each node, the complexity of the formula labelling this node is strictly greater than the complexity of each of the formulas labelling its sons.

Let us consider a node in our tree labeled with a formula F_0. Suppose its sons are labeled with formulas $F_1, F_2, \ldots, F_m$. The *splitting vector* of this node is the m-tuple $(t_1, t_2, \ldots, t_m)$, where t_i are positive numbers not exceeding $\mu(F_0) - \mu(F_i)$. The *characteristic polynomial* of this splitting vector is defined as follows: $h(x) = 1 - \sum_{i=1}^{m} x^{-t_i}$. The only positive root of this polynomial is called the *splitting number* and denoted by $\tau(t_1, t_2, \ldots, t_m)$. The *splitting number of a tree T* is the largest of the splitting numbers of its nodes. We denote it by $\tau_{max,T}$.

Now we have all the necessary definitions to present a lemma proved by Kullmann and Luckhardt in [11] that allows to estimate the number of leaves in a splitting tree using its splitting number.

Lemma 1. Let T be a splitting tree, and let its root be labeled with a formula F_0. Then the number of leaves in T does not exceed $(\tau_{max,T})^{\mu(F_0)}$.

Let us show how this lemma helps to estimate the running time of a splitting algorithm. As we already mentioned a splitting algorithm at first simplifies an input formula and then makes several recursive calls of itself for formulas with smaller complexity. Clearly, the total running time of such an algorithm is the total running time of all recursive calls plus the time spent to make these calls (note that we consider only splitting algorithms that at each step make a number of recursive calls bounded by a constant). Therefore, the running time is within a polynomial factor of the number of nodes (or leaves) of the recursion tree.

2.4 Simplification Rules

Throughout this paper we consider only simplification rules for SAT and MAXSAT. In case of SAT by an *optimal assignment* we mean an assignment to all variables of a formula that satisfies this formula, in case of MAXSAT we say that an assignment is *optimal* if this assignment satisfies the maximal possible number of clauses of a formula. Note that in case of MAXSAT any CNF formula has an optimal assignment, while in case of SAT only satisfiable formulas have such an assignment.

We say that a simplification rule is applicable to a formula F if it can replace F by another formula F' in polynomial time such that both following conditions hold:

1. the complexity of F' is smaller than the complexity of F;
2. by constructing a solution for F' one can construct a solution for F in polynomial time.

Examples of simplification rules for SAT and MAXSAT are given in Sect. 3.

2.5 Class of Formulas

Let C be a clause consisting of literals $l_1, l_2, \ldots, l_k$. We define a *clause with unfixed length* as a set of clauses that contains all these literals and probably some more literals and denote it by $(l_1 l_2 \ldots l_k \ldots)$. We call the literals $l_1, l_2, \ldots, l_k$ the *basis* of this clause. For example, $(l \ldots)$ is the set of all clauses containing the literal l. In the following we use the word "clause" to refer both to clauses in its standard definition and to clauses with unfixed length.

Similarly we define a *class of formulas*. Let $C'_1, \ldots, C'_k$ be clauses (some of them may have unfixed lengths). Then a *class of formulas* is the set of formulas represented as $F = \{C_1, \ldots, C_m\}$, such that the following conditions hold:

1. $m \geq k$,
2. for $1 \leq i \leq k$, $C'_i \subseteq C_i$ (as sets of literals), if C'_i is a clause with unfixed length, and $C_i = C'_i$ otherwise,
3. for $1 \leq i \leq k$, $k+1 \leq j \leq m$, C_j does not contain any variable from the basis of C'_i.

We denote this set by $C'_1, \ldots, C'_k \ldots$ and call the clauses $C'_1, \ldots, C'_k$ the *basis* of this class of formulas. We define two functions $CorSet$ and $CorClause$ (for corresponding set and corresponding clause) based on this definition: $CorSet(F, \mathcal{F}) = \{C_1, \ldots, C_k\}$, $CorClause(C'_i, F, \mathcal{F}) = C_i$.

The notion of class of formulas allows to explain the fact that a formula contains occurrences of certain literals. For example, to show that a formula F contains a $(3,2)$-literal one can write the following: $F \in (x \ldots)(x \ldots)(x \ldots)(\bar{x} \ldots)(\bar{x} \ldots) \ldots$. However, if we want to express the fact that F contains two $(1,1)$-literals that occur together in a clause, we have to write the following: $F \in (xy \ldots)(\bar{x} \ldots)(\bar{y} \ldots) \ldots$ or $F \in (xy \ldots)(\bar{x}\bar{y} \ldots) \ldots$.

For a formula F and a subset S of its variables we define $CorClass(F, S)$ as a class of formulas resulting from F by replacing all its variables that are not in S by "$\ldots$". Clearly, $F \in CorClass(F, S)$. For example, $CorClass((xyz)(\bar{x})(\bar{y}zu)\ (\bar{u}x)(\bar{z}), \{z, u\}) = (z \ldots)(zu \ldots)(\bar{u} \ldots)(\bar{z}) \ldots$.

Note that in most situations we can work with a class of formulas in the same way as with a CNF formula. For example, if we eliminate from the basis of a class of formulas all clauses that contain a literal x and all occurrences of the literal $\bar{x}$ from the other clauses, we obtain the class of formulas resulting from all formulas of the initial class by setting the value of x to $True$. Also it is easy to see that if after assigning a Boolean value to a variable of a class of formulas or applying a (considered in this paper) simplification rule to it, its complexity measure decreases by Δ, then the complexity measure of each formula of this class decreases *at least* by Δ.

For a class of formulas $\mathcal{F}$ and a clause C we define a class of formulas $\mathcal{F} + \{C\}$ as a class resulting from $\mathcal{F}$ by adding the clause C to its basis. Similarly we define a clause with unfixed length $C + \{l\}$, where C is a clause with unfixed length and l is a literal. By *fixing the length of a clause* $(l_1 l_2 \ldots l_k \ldots)$ we mean replacing it by the clause $(l_1 l_2 \ldots l_k)$.

We say that a simplification rule is applicable to a class of formulas, if this rule is applicable to every formula of this class. For example, each formula of the class $(x)(x\bar{y} \ldots)(y \ldots) \ldots$ contains a pure literal (i.e., a literal that does not occur negated in the formula). Similarly, we say that a class of formulas has a splitting if each formula of this class has this splitting. For example, every formula of the class $(xy)(\bar{x}y\bar{t} \ldots)(\bar{y}u)(u\bar{z})(\bar{x}t)(\bar{u}\bar{y}z) \ldots$ always has the $(1,2)$-splitting (on variable x) w.r.t. the number of variables (t is the other variable, which is eliminated), the $(2,2)$-splitting (on y) w.r.t. the number of clauses, and $(4,3)$-splitting (on z) w.r.t. the length of the formula. However, when we deal with specific problem we can find even better splittings (i.e., with smaller splitting numbers) since simplification rules can reduce a formula after splitting.

3 Known Simplification Rules for SAT and MAXSAT

In this section we list all known simplification rules for SAT and MAXSAT, that replace an input formula F either by a formula $F[l]$ or by a formula $F[l_1 = l_2]$, where l, l_1, l_2 are literals of F. We do not give all details of these rules, for their detailed description (as well as for the proof of correctness) see [2], [10], [9]. In the following F denotes an arbitrary CNF formula.

1. *Pure literal rule.* If F contains a literal l that does not occur in F negated, then replace F by $F[l]$. Valid for SAT and MAXSAT.

2. *Unit clause elimination rule.* If $(l) \in F$, then replace F by $F[l]$. Valid for SAT.

3. *Dominating unit clause rule.* If F contains a literal l, such that the literal $\bar{l}$ occurs in F not more times than the literal l occurs as a unit clause, then replace F with $F[l]$. Valid for MAXSAT.

4. *Almost dominating unit clause rule.* If F contains a literal l, such that l occurs in F as a unit clause k times and $\bar{l}$ occurs in $k+1$ clauses, such that two of them contain a pair of complementary literals (it means that at least one of these two clauses is *always* satisfied), then replace F with $F[l]$. Valid for MAXSAT.

5. *Rare variable rule.* If F contains a literal a, such that a occurs in F in two 2-clauses and $\bar{a}$ occurs only in one unit-clause, then replace F by $F[a = \bar{b}]$, where $(ab) \in F$. Valid for MAX-2-SAT (a version of MAXSAT, where each clause of an input formula contains at most two literals).

6. *Black-and-white literals rule.* Let P be a binary relation between literals and formulas in CNF, such that for a variable x and a formula F, at most one of $P(x, F)$ and $P(\bar{x}, F)$ holds. If each clause of F that contains a literal l satisfying $P(l, F)$ contains also at least one literal l_0 satisfying $P(\bar{l_0}, F)$, then replace F by $F[\{l | P(\bar{l}, F)\}]$. Valid for SAT. Of course, in an algorithm first this relation should be clearly defined and secondly its definition should provide a polynomial running time of this rule.

7. *Satisfying assignment rule.* If $F \in \mathcal{F}$ and A is an assignment to all variables of $\mathcal{F}$ that satisfies all clauses of $\mathcal{F}$, then replace F by $F[A]$. Valid both for SAT and MAXSAT. For this rule to run in polynomial time, it is reasonable to try to find such classes of formulas $\mathcal{F}$ with the number of variables bounded by a constant (that is, to consider $\mathcal{F} = CorClass(F, S)$ for $|S| \leq 2$).

In the examples of this paper we need also the following two simplification rules (these rules modify an input formula not by assigning a value to a literal, but by replacing several clauses of a formula by its resolvents):

1. *Almost common clauses rule.* If $F = F' \cup \{C, D\}$, where $C \backslash \{x\} = D \backslash \{\bar{x}\}$ for a literal x, then replace F by $F' \cup \{C \backslash \{x\}\}$. Used for MAXSAT.

2. *Resolution rule.* If $F = F' \cup \{(xC), (\bar{x}D)\}$, where C, D are sets of literals and neither x nor $\bar{x}$ occurs in F', then replace F by $F' \cup \{(CD)\}$. Used for SAT and MAXSAT. Actually, this is only a particular case of the resolution rule, for more details see, for example, [10].

4 Improvements of Splitting Algorithms

In this section we give two examples of splitting algorithm improvements.

Consider a very naive algorithm for SAT, which is described in the Fig. 1. The form of this algorithm was introduced by Davis et al. [5]. It uses two "classical" simplification rules, namely the pure literal and unit clause elimination rules. Clearly, this algorithm is correct, that is, it returns the answer *True* iff the input formula is satisfiable.

Let us estimate the time complexity of this algorithm. Clearly, after the first step of SATAlg1 each literal of the formula F occurs in it both positively and negatively. It means that $K(F) - K(F[x]) \geq 1$ and $K(F) - K(F[\bar{x}]) \geq 1$, and hence the algorithm SATAlg1 has worst-case running time $O(2^K)$.

Now we improve this algorithm by introducing a new simplification rule. Let SATAlg2 be the algorithm resulting from SATAlg1 by replacing its first step by the following step:

Procedure SatAlg1

Input: A formula F in CNF.
Output: If F is satisfiable, then *True*, otherwise *False*.
Method.

1. apply the unit clause elimination and pure literal rules to F as long as at least one of them is applicable
2. if F is empty, return *True*
3. if F contains an empty clause, return *False*
4. choose any literal x occurring in F and recursively call for $F[x]$ and $F[\bar{x}]$
5. if at least one of the recursive calls returns *True*, return *True*, otherwise return *False*

Figure 1: The naive algorithm for SAT

- apply the unit clause elimination, pure literal and resolution rules to F as long as at least one of them is applicable

The correctness of SATAlg2 is again trivial. Note, however, that the resolution rule guarantees that after the first step of SATAlg2, F does not contain not only pure literals, but also $(1,1)$-literals. Thus, the splitting vector $(1,2)$ corresponds to the fourth step of SATAlg2, which means that SATAlg2 has worst-case running time $O(1.619^K)$.

Now let us show how it is possible to improve SATAlg2 by a more involved case analysis. We define algorithm SATAlg3 as an algorithm resulting from SATAlg2 by replacing its fourth step by the following step:

- let x be a literal of F such that the total number of occurrences of l and $\bar{l}$ is at least 4; if there are no such literals let x be any literal of F; recursively call for $F[x]$ and $F[\bar{x}]$

We show that SATAlg3 has better running-time than SATAlg2. To do it we prove that at the fourth step SATAlg3 makes splittings with splitting number that does not exceed 1.466. Note that if the formula F contains a literal l, such that the total number of occurrences of l and $\bar{l}$ is at least 4, then splitting on this literal provides a splitting number not greater than 1.466, since $\tau(2,2) = 1.414\ldots$, $\tau(1,3) = 1.465\ldots$. Otherwise, F contains $(1,2)$- and $(2,1)$-literals only at the step 4 of SATAlg3. Thus, $F \in (x\ldots)(\bar{x}\ldots)(\bar{x}\ldots)\ldots$. Note that F cannot contain a unit clause, hence there is at least one more literal (call it y) in the clause with x. y is either a $(1,2)$- or $(2,1)$-literal, hence in $F[x]$ it becomes either a $(0,2)$-, or $(2,0)$-, or $(1,1)$-literal. This means, that at least one clause can be eliminated from $F[x]$ by simplification rules of SATAlg3. By noting that $K(F) - K(F[\bar{x}]) \geq 2$ we conclude that splitting on x in this case is the $(2,2)$-splitting. Thus, SATAlg3 has worst-case running time $O(1.466^K)$.

Actually there are also known several other small tricks that allow to improve splitting algorithms. Here are the examples:

1. The algorithm for MAXSAT of Chen and Kanj [3] changes (i.e., splits or applies simplification rules) an input formula only in case this change preserves certain invariant. This allows to avoid bottleneck cases.

2. Gramm et al. in [9] prove that their algorithm for MAX-2-SAT makes "good" splittings for almost all input formulas. Then they prove that each path of the tree corresponding to the execution of this algorithm cannot contain more than one of the rest formulas. This allows to prove a "good" upper bound on the running time of the algorithm.

3. Byskov et al. in [1] introduce a notion of a sparse formula. This allows to use efficiently the known fact that an optimal assignment for some formulas can be constructed in polynomial time and therefore to avoid some bottleneck cases.

5 New Simplification Rules

In this section we present several new simplification rules for the SAT and MAXSAT problems.

Usually to prove the correctness of a simplification rule that assigns the value *True* to a literal x one proves that every optimal assignment that contains the literal $\bar{x}$ can be reconstructed into an optimal assignment that contains the literal x. For example, if x is a pure literal of a formula F then any assignment $A \ni \bar{x}$ satisfies not more clauses than the assignment $A \backslash \{\bar{x}\} \cup \{x\}$. This discussion motivates the following definition.

Definition 1. Let A_1 and A_2 be assignments to all known variables of a class of formulas $\mathcal{F}$. We say that A_1 is stronger than A_2 w.r.t. $\mathcal{F}$ and write $A_1 \succ_{\mathcal{F}} A_2$, if for each formula $F \in \mathcal{F}$ and for each assignment B to all variables of F the following condition holds: if $A_2 \subset B$, then B satisfies not more clauses of F than $B \backslash A_2 \cup A_1$.

For example, in case of MAXSAT problem if $\mathcal{F}_1 = (x \ldots)(x \ldots)\ldots$, then $\{x\} \succ_{\mathcal{F}_1} \{\bar{x}\}$ (due to the pure literal rule), and if $\mathcal{F}_2 = (xy \ldots)(\bar{x})(x\bar{y} \ldots)(\bar{x})(y \ldots)(x \ldots)\ldots$, then $\{\bar{x}\} \succ_{\mathcal{F}_2} \{x\}$ (due to the almost dominating unit clause rule). The given definition is very easy and natural. However, in general, it is not clear how to check whether one of the given assignments is stronger than the other. Below we show that such a check is possible.

Definition 2. Let C be a clause and A be an assignment. We define the function $mark(C, A)$ as follows:

$$mark(C, A) = \begin{cases} \text{“}+\text{”}, & \text{if } C \text{ contains at least one literal from } A; \\ \text{“}-\text{”}, & \text{if } C \text{ has fixed length and for each literal } x \in C,\ \bar{x} \in A; \\ \text{“?”}, & \text{otherwise.} \end{cases}$$

For example, $mark((xyz\ldots), \{y\}) =$ "+", $mark((x\bar{y}), \{\bar{x}, y\}) =$ "−", $mark((x\bar{y}), \{\bar{x}, z\}) =$ "?", $mark((x\bar{y}\ldots), \{\bar{x}, y\}) =$ "?". Informally, $mark(C, A)$ just tell us whether clauses of C are satisfied by extensions of A.

The following lemma formulates several straightforward consequences of the given definition.

Lemma 2. Let A_1, A_2 be assignments to all variables of a class of formulas $\mathcal{F}$, F be a formula of this class, B be an assignment to all variables of F, such that $A_2 \subset B$. Then

- for any clause $C \in F \backslash CorSet(F, \mathcal{F})$, $mark(C, B) = mark(C, B \backslash A_2 \cup A_1)$;

- for any clause C, if $mark(C, A_1) =$ " + " (or " − "), then for any assignment A, such that $A_1 \subset A$, $mark(C, A) =$ " + " (correspondingly, " − ");
- for any clause $C \in \mathcal{F}$, if $mark(C, A_1) = mark(C, A_2) =$ "?", then $mark(C_0, B \backslash A_2 \cup A_1) = mark(C_0, B)$, where $C_0 = CorClause(C, F, \mathcal{F})$.

Theorem 1. Let A_1, A_2 be assignments to all variables of a class of formulas $\mathcal{F}$ and let

$$p_1 = |\{C \in \mathcal{F} \mid mark(C, A_1) = \text{“}+\text{”}\}| \ ,$$

$$p_2 = |\{C \in \mathcal{F} \mid mark(C, A_2) = \text{“}+\text{”}\}| \ ,$$

$$q = |\{C \in \mathcal{F} \mid mark(C, A_2) = \text{“?”}, mark(C, A_1) \neq \text{“?”}\}| \ .$$

Then, $A_1 \succ_{\mathcal{F}} A_2$ iff $p_1 \geq p_2 + q$.

Proof. To prove the "if" part consider a formula $F \in \mathcal{F}$ and an assignment B to all variables of F, such that $A_2 \subset B$. Then the required statement follows from the following (in)equalities.

$$|\{C \in F \mid mark(C, B \backslash A_2 \cup A_1) = \text{“}+\text{”}\}| -$$
$$|\{C \in F \mid mark(C, B) = \text{“}+\text{”}\}| \overset{L.2(1)}{=}$$

$$|\{C \in CorSet(F, \mathcal{F}) \mid mark(C, B \backslash A_2 \cup A_1) = \text{“}+\text{”}\}| -$$
$$|\{C \in CorSet(F, \mathcal{F}) \mid mark(C, B) = \text{“}+\text{”}\}| \overset{L.2(2)}{=}$$

$$p_1 - p_2 +$$
$$|\{C \in \mathcal{F} \mid mark(C, A_1) = \text{“?”}, mark(CorClause(C, F, \mathcal{F}), B \backslash A_2 \cup A_1) = \text{“}+\text{”}\}| -$$
$$|\{C \in \mathcal{F} \mid mark(C, A_2) = \text{“?”}, mark(CorClause(C, F, \mathcal{F}), B) = \text{“}+\text{”}\}| \overset{L.2(3)}{=}$$

$$p_1 - p_2 + |\{C \in \mathcal{F} \mid mark(C, A_1) = \text{“?”}, mark(C, A_2) \neq \text{“?”},$$

$$mark(CorClause(C, F, \mathcal{F}), B \backslash A_2 \cup A_1) = \text{“}+\text{”}\}| -$$

$$|\{C \in \mathcal{F} \mid mark(C, A_2) = \text{“?”}, mark(C, A_1) \neq \text{“?”},$$
$$mark(CorClause(C, F, \mathcal{F}), B) = \text{“}+\text{”}\}| \geq$$

$$p_1 - p_2 - |\{C \in \mathcal{F} \mid mark(C, A_2) = \text{“?”}, mark(C, A_1) \neq \text{“?”}\}| =$$
$$p_1 - p_2 - q \geq 0$$

To prove the "only if" part we show that if $p_1 < p_2 + q$, then $A_1 \not\succ_{\mathcal{F}} A_2$. So, suppose $p_1 < p_2 + q$ and consider such a formula $F \in \mathcal{F}$: for each clause $C \in \mathcal{F}$,

$$CorClause(C,F,\mathcal{F}) = \begin{cases} FixLength(C \cup \{x\}), & \text{if } mark(C,A_2) = \text{“?”} \\ & \text{and } mark(C,A_1) \neq \text{“?”}, \\ FixLength(C), & \text{otherwise,} \end{cases}$$

where x is a variable that does not occur in $\mathcal{F}$. Then an assignment $B = A_2 \cup x$ is clearly an assignment to all variables of F and it satisfies p_1 clauses of F, while an assignment $B \backslash A_2 \cup A_1$ satisfies $p_2 + q$ clauses. This means that $A_1 \not\succ_{\mathcal{F}} A_2$. □

Let us show how this theorem allows to generate simplification rules. We start with giving several examples. Suppose $F \in \mathcal{F} = (xy)(x\bar{y})(\bar{x}y)(\bar{x}\bar{y}\ldots)\ldots$. In such a case we can replace F by $F[\bar{x},\bar{y}]$, since $\{\bar{x},\bar{y}\} \succ_{\mathcal{F}} \{x,y\},\{\bar{x},y\},\{x,\bar{y}\}$. Thus, the assignment $\{\bar{x},\bar{y}\}$ satisfies *at least* three clauses from $CorSet(F,\mathcal{F})$, while any other assignment to variables x and y satisfies *at most* three of these clauses.

Consider a slightly more complicated example. Let $\mathcal{F} = (xy\ldots)(x\ldots)(\bar{x}\bar{y}\ldots)(\bar{y})\ldots$, then $\{\bar{x},y\} \succ_{\mathcal{F}} \{\bar{x},\bar{y}\}$ and $\{x,\bar{y}\} \succ_{\mathcal{F}} \{x,y\}$. It means that for any formula $F \in \mathcal{F}$, if there exists an optimal assignment for F, then there exists an optimal assignment B, such that either $\{\bar{x},y\} \subset B$ or $\{x,\bar{y}\} \subset B$. Thus, we can replace F by $F[y = \bar{x}]$.

In both given examples we constructed a *majority set* of assignments to all variables of a class of formulas. Below we formally define this notion.

Definition 3. Let $\mathcal{F}$ be a class of formulas, M_0 be the set of all possible assignments to all variables of $\mathcal{F}$. We say that $M \subset M_0$ is a *majority set* for $\mathcal{F}$ and write $M = MajorSet(\mathcal{F})$, if for any assignment $A_0 \in M_0$ there exists an assignment $A \in M$, such that $A \succ_{\mathcal{F}} A_0$.

It is easy to see that a majority set is not unique and that, in particular, M_0 (from the definition) is a majority set for $\mathcal{F}$. However, we are interested in majority sets of small size. We use a greedy algorithm for construction of such majority sets, its code is given in Fig. 2.

Now we are ready to present the new simplification rule that unifies all known simplification rules that modify an input formula by assigning a value to a literal. It is illustrated in Fig. 3. It is easy to see that the running time of this rule is $O(L)$, the correctness is trivial. We call this rule an *automated* procedure for generating simplification rules, as one can add this procedure with any constant parameter into an algorithm. Clearly, if $GSR(c_1)$ can simplify a formula, then $GSR(c_2)$ can simplify this formula too, for $c_1 < c_2$.

It is not difficult to show that if a formula can be simplified by some of the rules listed in Sect. 3 except for the black-and-while literals and satisfying assignment rules, then it can be simplified by $GSR(2)$. However, the black-and-white literals rule is, in some sense, a particular case of the satisfying assignment rule, since an assignment $A = \{l | P(\bar{l},F)\}$ clearly satisfies all clauses of a class of formulas $\mathcal{F} = CorClass(F,S)$, where S is the set of variables corresponding to the literals of A. And the satisfying assignment rule is, in turn, a particular case of GSR, since if an assignment A satisfies all clauses of a class of formulas $\mathcal{F}$, then $\{A\} = MajorSet(\mathcal{F})$.

Below we give several new simplification rules, that are particular cases of GSR.

New simplification rules for MAXSAT:

1. if $F \in \mathcal{F} = (x)(\bar{x}y\ldots)(\bar{x}\bar{y}\ldots)(\bar{x}\bar{y}\ldots)(\bar{y}\ldots)(\bar{y}\ldots)\ldots$, then replace F by $F[x]$ (since $\{\{x,y\},\{x,\bar{y}\}\} = MajorSet(\mathcal{F})$)
2. if $F \in \mathcal{F} = (xyz\ldots)(\bar{x}z\ldots)(\bar{x}z\ldots)(y\ldots)(\bar{y}\bar{z}\ldots)(z\ldots)\ldots$, then replace F by $F[x = y]$ (since $\{\{\bar{x},y,\bar{z}\},\{x,\bar{y},z\},\{x,y,z\}\} = MajorSet(\mathcal{F})$)

Procedure MSC

Input: A class of formulas $\mathcal{F}$.
Output: A majority set for $\mathcal{F}$.
Method.

1. let $M_0 = 0$
2. let M be the set of all possible assignments to all variables of $\mathcal{F}$
3. in case of SAT: remove from M all assignments A, such that $mark(C, A) =$ " $-$ " for some clause $C \in \mathcal{F}$
4. while $M \neq 0$
   (a) let A_0 be an assignment of M such that $|\{A \in M | A_0 \succ_{\mathcal{F}} A\}|$ is maximal
   (b) $M_0 = M_0 \cup \{A_0\}$
   (c) $M = M \backslash \{A \in M | A_0 \succ_{\mathcal{F}} A\}$
5. return M_0

Figure 2: The greedy algorithm for majority set construction

Procedure GSR(integer c)

Input: A CNF formula F.
Output: A simplified CNF formula F.
Method.
for each subset S of variables of F, such that $|S| \leq c$

1. let $\mathcal{F} = CorClass(F, S)$
2. let $M = MajorSet(\mathcal{F})$
3. if there is a literal x, such that it occurs in every assignment from M, then return $F[x]$
4. if there are literals x and y, such that for any assignment $A \in M$, either $\{x, y\} \subset A$ or $\{\bar{x}, \bar{y}\} \subset A$, then return $F[x = y]$

Figure 3: The general simplification rule

3. if $F \in \mathcal{F} = (x)(\bar{x}yz\ldots)(\bar{x}\bar{t}\ldots)(\bar{x}\bar{t}\ldots)(yt\ldots)(\bar{y}\bar{t}\ldots)(\bar{y}\ldots)(z\bar{t}\ldots)(\bar{z}\ldots)\ (\bar{z}\ldots)\ldots$, then replace F by $F[x]$ (since $\{\{x,\bar{y},\bar{t}\},\{x,y,\bar{t}\},\{x,\bar{y},t\}\} = MajorSet(\mathcal{F})$)

New simplification rules for SAT:

1. if $F \in \mathcal{F} = (xy)(\bar{x}\ldots)(\bar{y}\ldots)(\bar{x}y\ldots)(\bar{y}\ldots)\ldots$, then replace F by $F[x = \bar{y}]$ (since $\{\{x,\bar{y}\},\{\bar{x},y\}\} = MajorSet(\mathcal{F})$)
2. if $F \in \mathcal{F} = (xy\ldots)(xz\ldots)(\bar{x}\bar{z}\ldots)(\bar{x}\bar{z}\ldots)(\bar{x}\bar{z}\ldots)(y\ldots)(y\ldots)(\bar{y}z\ldots)(\bar{y}\ldots)\ (\bar{z}\ldots)\ldots$, then replace F by $F[x = \bar{z}]$ (since $\{\{x,\bar{y},\bar{z}\},\{x,y,\bar{z}\},\{x,\bar{y},\bar{z}\}\} = MajorSet(\mathcal{F})$)

Actually our technique allows to find not only simplifications, but also good splittings. The main idea is that for any formula $F \in \mathcal{F}$ the splitting $F[A_1]$, $F[A_2], \ldots, F[A_k]$ is correct, where $\{A_1, A_2, \ldots, A_k\} = MajorSet(\mathcal{F})$.

In the follwoing section we present a new splitting algorithm that uses both simplifications and splittings described above.

6 A New Algorithm for $(n,3)$-MAXSAT

The $(n,3)$-MAXSAT problem is a version of MAXSAT where each variable appears at most three times in an input formula. In this section we present a new $O(1.272021^N L)$-time algorithm for this problem. The automated proofs of this bound can be found at `http://logic.pdmi.ras.ru/~kulikov`. Actually two proofs are available there, the difference between them is in sets of simplification rules. The equal number of cases in both these proofs shows that the pure literal, dominating unit clause, almost dominating unit clause, and satisfying assignment rules are indeed particular cases of *GSR*. The algorithm is quite simple (compared to the previously best $O(1.324719^N L)$-time algorithm of Bansal and Raman [2]) and uses both simplifications and splittings based on the approach of the paper.

The code of the new algorithm is presented in Fig. 4, we do not give all details, as we are mainly interested in its running time. The following proof of the upper bound on the running time of NewAlg allows to understand better the structure of automated proofs generated by our framework, that is, the case analysis is executed similarly in all these proofs (for full details see [7]).

Theorem 2. The algorithm NewAlg has worst-case running time $O(1.272021^N L)$.

Proof. We recall that the pure literal, dominating unit clause, and almost dominating unit clause rules are particular cases of *GSR*. Note that if a formula F is in $(n,3)$-CNF, then so is a formula $F[l]$ (for a literal l). This property is also invariant w.r.t. the resolution and almost common clauses rules. However, the formula $F[l_1 = l_2]$ is not necessarily in $(n,3)$-CNF. Note also that if an occurrence of a variable was eliminated during splitting or application of the simplification rules, then further application of simplification rules of NewAlg eliminates all other occurrences of this variable.

We show that at the fourth step NewAlg makes only $(2,4)$-, $(3,3)$-, and $(1,7)$-splittings. This proves the theorem, since $\tau(2,4) = 1.272020\ldots$, $\tau(3,3) = 1.259921\ldots$, $\tau(1,7) = 1.255423\ldots$.

Consider a formula F at the third step of NewAlg. According to the resolution and pure literal rules, F contains neither $(0,i)$-, nor $(i,0)$-, nor $(1,1)$-literals. Thus, F contains only $(1,2)$- and $(2,1)$-literals. Let a be a $(1,2)$-literal, C_1, C_2, C_3 be clauses of F, such that $C_1 \ni a$ and $C_2, C_3 \ni \bar{a}$.

Procedure NewAlg

Input: A $(n, 3)$-CNF formula F.
Output: The maximal possible number of simultaneously satisfied clauses of F.
Method.

1. apply the resolution, almost common clauses, and $GSR(2)$ rules to F as long as at least one of them is applicable
2. if the answer is obvious, return it
3. for each subset S of variables of F, such that $|S| \leq 2$:
   (a) let $\mathcal{F} = MajorSet(F, S)$
   (b) let $\{A_1, A_2, \ldots, A_k\} = MajorSet(\mathcal{F})$
   (c) consider the splitting $F[A_1], F[A_2], \ldots, F[A_k]$
4. from the splittings considered at the previous step choose the one with smallest splitting number and make the corresponding recursive calls
5. return the answer according to the answers returned by the recursive calls

Figure 4: The new algorithm for $(n, 3)$-MAXSAT

C_2 and C_3 are not unit due to the dominating unit clause rule. Suppose, C_2 and C_3 contain only one other variable except for $\bar{a}$. Since C_2 and C_3 are not almost common, $C_2 = C_3 = (\bar{a}b)$. There are two possible cases of other occurrence of the variable that corresponds to the literal b:

1. $F \in \mathcal{F} = (a \ldots)(\bar{a}b)(\bar{a}b)(\bar{b} \ldots) \ldots$

   In this case F can be replaced by $F[a = b]$, since $\{\{a, b\}, \{\bar{a}, \bar{b}\}\} = MajorSet(\mathcal{F})$. The formula $F[a = b]$ is in $(n, 3)$-CNF, since the clauses C_2 and C_3 are satisfied in it.

2. $F \in \mathcal{F} = (a\bar{b} \ldots)(\bar{a}b)(\bar{a}b) \ldots$

   In this case F can be replaced by $F[a]$, since $\{\{a, b\}\} = MajorSet(\mathcal{F})$.

Thus, there are two different literals c and d, such that $c \in C_2$, $d \in C_3$. If C_1 contains at least three literals, then the splitting $F[a], F[\bar{a}]$ is the $(3,3)$-splitting (in $F[a]$ all the literals of C_1 are eliminated by the simplification rules, in $F[\bar{a}]$ variables that correspond to literals c and d are eliminated).

Now consider the case when C_1 contains exactly two literals. Let $C_1 = (ab)$. Suppose that either b or $\bar{b}$ occurs in C_2 or C_3. There are five cases:

1. $F \in \mathcal{F} = (ab)(\bar{a}\bar{b} \ldots)(\bar{a}\bar{b} \ldots) \ldots$

   F can be replaced by $F[a]$, since $\{\{a, \bar{b}\}\} = MajorSet(\mathcal{F})$.

2. $F \in \mathcal{F} = (ab)(\bar{a}\bar{b} \ldots)(\bar{a}b \ldots) \ldots$

   F can be replaced by $F[\bar{a}]$, since $\{\{\bar{a}, b\}\} = MajorSet(\mathcal{F})$.

3. $F \in \mathcal{F} = (ab)(\bar{a}\bar{b} \ldots)(\bar{a} \ldots)(\bar{b} \ldots) \ldots$

   F can be replaced by $F[a = \bar{b}]$, since $\{\{\bar{a}, b\}, \{a, \bar{b}\}\} = MajorSet(\mathcal{F})$. It is easy to see that $F[a = \bar{b}]$ is in $(n, 3)$-CNF.

4. $F \in \mathcal{F} = (ab)(\bar{a}\bar{b} \ldots)(\bar{a} \ldots)(b \ldots) \ldots$

   F can be replaced by $F[\bar{a}]$, since $\{\{\bar{a}, b\}\} = MajorSet(\mathcal{F})$.

5. $F \in \mathcal{F} = (ab)(\bar{a}b \ldots)(\bar{a} \ldots)(\bar{b} \ldots) \ldots$

   F can be replaced by $F[a = \bar{b}]$, since $\{\{\bar{a}, b\}, \{a, \bar{b}\}\} = MajorSet(\mathcal{F})$. It is easy to see that $F[a = \bar{b}]$ is in $(n, 3)$-CNF.

Thus, neither C_2 nor C_3 contains occurrences of the variable that corresponds to b. It means that the formula F is a member of at least one of the following classes:

1. $\mathcal{F} = (ab)(\bar{a} \ldots)(\bar{a} \ldots)(\bar{b} \ldots)(\bar{b} \ldots) \ldots$

   In this case the splitting $F[a], F[\bar{a}]$ is the $(3,3)$-splitting, since in $F[a]$ the variable that corresponds to b and also at least one other variable from the last two clauses of $\mathcal{F}$ are eliminated, and in $F[\bar{a}]$ the variables that correspond to c and d are eliminated.

2. $\mathcal{F} = (ab)(\bar{a} \ldots)(\bar{a} \ldots)(b \ldots)(\bar{b} \ldots) \ldots$ In this case the splitting $F[a, \bar{b}], F[\bar{a}, b]$ is obviously the $(2,4)$-splitting. This splitting is correct, since $\{\{a, \bar{b}\}, \{\bar{a}, b\}\} = MajorSet(\mathcal{F})$.

The remained case is when C_1 is a unit clause. Since a was any $(1,2)$-literal, at this point all $(1,2)$-literals have to appear as unit clauses. Suppose that C_2 and C_3 contain at least one common literal except for $\bar{a}$. This is the case when $F \in \mathcal{F} = (a)(\bar{a}b\ldots)(\bar{a}b\ldots)(\bar{b})\ldots.$ Then F can be replaced by $F[\bar{b}]$, since $\{\{a,\bar{b}\},\{\bar{a},\bar{b}\}\} = MajorSet(\mathcal{F})$. So, the only common literal of C_2 and C_3 is $\bar{a}$.

Suppose that at least one of the clauses C_2 and C_3 has length 2. W.l.o.g. consider the case when this clause is C_2. Then $F \in \mathcal{F} = (a)(\bar{a}b)(\bar{a}\ldots)(b\ldots)(\bar{b})\ldots.$ F can be replaced by $F[a=b]$, since $\{\{a,b\},\{\bar{a},\bar{b}\}\} = MajorSet(\mathcal{F})$. The formula $F[a=b]$ becomes an $(n,3)$-CNF formula after an application of the almost common clauses rule.

Now suppose that C_2 has length exactly 3 (and C_3 — at least 3). Then in $F[a]$ the length of C_2 becomes equal 2 and $F[a]$ can be simplified as in the previous case. Thus, $F[a], F[\bar{a}]$ is the $(2,5)$-splitting.

The only remained case is when both C_2 and C_3 have length at least 4. Then $F[a], F[\bar{a}]$ is the $(1,7)$-splitting (as C_2, C_3 do not have common literals except for $\bar{a}$).

$\square$

7 Further Directions

It would be interesting to apply the approach of the paper to obtain new simplification rules for other NP-hard problems. For example, the paper [1] contains a lot of simplification rules for the XSAT problem. We believe that many of them (and maybe other ones) can be generated by our procedure.

The described simplification rules and splittings were not sufficient for our framework for automated proofs of upper bound for NP-hard problems to improve currently best divide-and-conquer algorithms for SAT and MAXSAT. This is because the algorithms use several non-standard tricks (they are roughly described in Sect. 4) that are difficult to add in such a framework. We are going to try to introduce these tricks into the program.

Acknowledgements

The author would like to express a great gratitude to his supervisor Edward A. Hirsch for valuable comments.

References

[1] J. M. Byskov, B. A. Madsen, and B. Skjernaa. New Algorithms for Exact Satisfiability. *Theoretical Computer Science Preprint*, 2003.

[2] N. Bansal and V. Raman. Upper Bounds for MaxSat: Further Improved. *Proceedings of ISAAC'99*, 247–258, 1999.

[3] J. Chen and I. Kanj. Improved exact algorithms for MAX-SAT. *Proceedings of 5th LATIN*, Lecture Notes in Compter Science, 2286: 341–355, 2002.

[4] E. Dantsin, E. A. Hirsch, S. Ivanov, and M. Vsemirnov, Algorithms for SAT and Upper Bounds on Their Complexity. ECCC Technical Report 01-012, 2001. Electronic address: `ftp://`

ftp.eccc.uni-trier.de/pub/eccc/reports/2001/TR01-012/index.html. A Russian version appears in *Zapiski Nauchnykh Seminarov POMI*, 277: 14–46, 2001.

[5] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Comm. ACM*, 5:394–397, 1962.

[6] S. S. Fedin, A. S. Kulikov, A $2^{|E|/4}$-time Algorithm for MAX-CUT. *Zapiski nauchnykh seminarov POMI*, 293: 129–138, 2002. English translation is to appear in Journal of Mathematical Sciences.

[7] S. S. Fedin, A. S. Kulikov, Automated Proof of Upper Bounds for NP-hard problems. *Proceedings of IWPEC'04*, Lecture Notes in Computer Science, 3162: 248–259, 2004.

[8] J. Gramm, J. Guo, F. Hüffner, and R. Niedermeier. Automated Generation of Search Tree Algorithms for Hard Graph Modification Problems. *Proceedings of 11th Annual European Symposium on Algorithms*, 642–653, 2003.

[9] J. Gramm, E. A. Hirsch, R. Niedermeier, and P. Rossmanith. New worst-case upper bounds for MAX-2-SAT with application to MAX-CUT. *Discrete Applied Mathematics*, 130(2):139–155, 2003.

[10] E. A. Hirsch. New worst-case upper bounds for SAT. *Journal of Automated Reasoning*, 24(4):397–420, 2000.

[11] O. Kullmann and H. Luckhardt. Algorithms for SAT/TAUT decision based on various measures. *Preprint*, 1998. Available at http://cs-svr1.swan.ac.uk/∼csoliver/papers/html.

[12] S. I. Nikolenko and A. V. Sirotkin. Worst-case upper bounds for SAT: automated proof. *Proceedings of the 8th ESSLLI Student Session*, 225–232, 2003. Available at http://logic.pdmi.ras.ru/∼sergey.