# On Hierarchies over the Class of SLUR Formulae

Tomáš Balyo[1,2], Štefan Gurský[1,3], Petr Kučera[1,4], and Václav Vlček[1,5,⋆]

[1] Department of Theoretical Computer Science and Mathematical Logic, Charles University, Malostranské nám. 25, 118 00 Praha 1, Czech Republic
[2] biotomas@gmail.com
[3] stevko@mail.ru
[4] kucerap@ktiml.ms.mff.cuni.cz
[5] vlcek@ktiml.mff.cuni.cz

**Abstract.** Single look-ahead unit resolution (SLUR) algorithm is a nondeterministic polynomial time algorithm which for a given input formula in a conjunctive normal form (CNF) either outputs its satisfying assignment or gives up. A CNF formula belongs to the SLUR class if no sequence of nondeterministic choices causes the SLUR algorithm to give up on it. The SLUR class is reasonably large. It is known to properly contain the well studied classes of Horn CNFs, renamable Horn CNFs, extended Horn CNFs, and CC-balanced CNFs. Recently, it was shown, that a canonical representation of a Boolean function always belongs to the SLUR class [4, 5]. In this paper we extend this result by showing, that this remains true even for some representations, which are not far from the canonical one. We also generalize the hierarchy of classes of Boolean formulae which was built on top of the SLUR class in [5, 18].

**Keywords:** Boolean functions, CNF satisfiability, unit resolution

## 1 Introduction

The satisfiability problem (SAT) is to decide whether a given formula $\varphi$ in CNF has a satisfying assignment, i.e. whether for some assignment $t$ of values 0 (false) or 1 (true) to variables we have that $\varphi(t)$ evaluates to 1 (true). This problem was the first one shown to be NP-complete [3, 11]. Thus, unless P=NP, no polynomial time algorithm can solve this problem. There are, however, many classes of formulae for which polynomial SAT algorithms are known. These classes of formulae include Horn formulae [9, 13, 15], renamable Horn formulae [14, 1], extended Horn formulae [6], and CC-balanced formulae [7]. These four classes share an interesting property: the satisfiability problem for formulae from these classes can be solved by unit resolution, namely by the single look-ahead unit resolution (SLUR) algorithm [17, 10].

The SLUR algorithm works as follows. In every step it nondeterministically chooses a variable and its value and performs as many unit resolutions as possible. If the unit resolutions do not yield a contradiction, the algorithm continues with the new formula in the same way. If the contradiction (the empty clause) is derived, the algorithm tries the other value for the chosen variable and continues with the new formula. If both values lead to a contradiction using unit resolutions, the SLUR algorithm gives up. A nice property of Horn, renamable Horn, extended Horn, and CC-balanced formulae is that the SLUR algorithm never gives up on them [17]. Therefore, it is natural to define a generalization of these four classes called the SLUR class as the class of those formulae on which the SLUR algorithm never gives up.

In [4, 5] it was shown, that every Boolean function has at least one representation which belongs to the SLUR class, which is in contrast with for example Horn formulae. A Boolean function $f$ is called Horn, if there exists at least one Horn CNF representing $f$. However, if we would say, that a function $f$ is SLUR, if there exists at least one CNF representing $f$ which belongs to the SLUR class, then by the above mentioned result, every Boolean function would be SLUR. In particular, it was shown in [4, 5], that if a CNF representation $\varphi$ of $f$ contains all prime implicates of $f$, then $\varphi$ belongs to the SLUR class. In this paper we shall extend this result by relaxing this condition.

In [18, 5] a hierarchy of CNFs was built on top of the SLUR class by modifying the SLUR algorithm. Instead of choosing one variable at every step, we choose $i$ variables and check all possible assignments of Boolean values to them, if no sequence of nondeterministic choices causes this generalized algorithm to give up on a CNF, then it is said to belong to the SLUR($i$) class. In this paper we consider a similar

⋆ The first, the second, and the fourth author gratefully acknowledge the support by the Science Foundation of the Charles University (grant No. 266111).

hierarchy, which can be most easily described as a restriction of a DPLL procedure [16], which at the basis of most modern SAT solvers. In fact the SLUR algorithm can be viewed as a restricted version of DPLL procedure, which instead of backtracking more than one level up, gives up. This means, that SLUR formulae are those, which are easily solved by the DPLL procedure. If we allow backtracking not one, but two, or generally $i$ levels up before giving up, we can get another hierarchy of formulae, whose $i$-th level will be called SLUR*($i$). We shall show, that the classes of hierarchy SLUR($i$) are then properly contained in the classes SLUR*($i$).

The rest of the paper is organized as follows. In Section 2 we introduce necessary definitions and basic known results. In Section 3 we shall show, that formulae, which are not far from the canonical representation, belong to the SLUR class. In Section 4 we define new hierarchy defined on top of the SLUR class and we show its properties. Finally we close the paper with conclusions and some open questions in Section 5.

## 2 Definitions and results

A *Boolean function* $f$ on $n$ propositional variables $x_1, \ldots, x_n$ is a mapping $\{0,1\}^n \to \{0,1\}$. The propositional variables $x_1, \ldots, x_n$ and their negations $\overline{x}_1, \ldots, \overline{x}_n$ are called *literals* (*positive* and *negative literals*, respectively). An disjunction of literals

$$C = \bigvee_{i \in I} \overline{x}_i \vee \bigvee_{j \in J} x_j \tag{1}$$

is called a *clause*, if every propositional variable appears in it at most once, i.e. if $I \cap J = \emptyset$. The *degree* of a clause $C$ is the number of literals in C. For two Boolean functions $f$ and $g$ we write $f \leq g$ if

$$\forall (x_1, \ldots, x_n) \in \{0,1\}^n \ : \ f(x_1, \ldots, x_n) = 1 \quad \Longrightarrow \quad g(x_1, \ldots, x_n) = 1 \tag{2}$$

Since each clause is in itself a Boolean function, formula (2) also defines the meaning of inequalities $C_1 \leq C_2$, $C_1 \leq f$, and $f \leq C_1$, where $C_1, C_2$ are clauses and $f$ is a Boolean function.

We say that a clause $C_1$ *subsumes* another clause $C_2$ if $C_1 \leq C_2$ (e.g. the clause $\overline{x} \vee z$ subsumes the clause $\overline{x} \vee \overline{y} \vee z$). A clause $C$ is called an *implicate* of a function $f$ if $f \leq C$. An implicate $C$ is called *prime* if there is no distinct implicate $C'$ subsuming $C$, or in other words, an implicate of a function is prime if dropping any literal from it produces a clause which is not an implicate of that function.

It is a wellknown fact that every Boolean function $f$ can be represented by a conjunction of clauses (see e.g. [12]). Such an expression is called a *conjunctive normal form* (or CNF) of the Boolean function $f$. It should be noted that a given Boolean function may have many CNF representations. If two distinct CNFs, say $\phi_1$ and $\phi_2$, represent the same function, we say that they are *equivalent*, and denote this fact by $\phi_1 \equiv \phi_2$. A CNF $\phi$ representing a function $f$ is called *prime* if each clause of $\phi$ is a prime implicate of function $f$. The unique CNF consisting of all prime implicates of function $f$ is called the *canonical CNF* of $f$. A CNF $\phi$ representing a function $f$ is called *irredundant* if dropping any clause from $\phi$ produces a CNF that does not represent $f$. We shall often treat a CNF as a set of its clauses.

For a Boolean function $f$ on $n$ variables, a variable $x$, and a value $e \in \{0,1\}$ we denote by $f[x := e]$ the Boolean function on $n-1$ variables which results from $f$ by assigning the value $e$ to variable $x$. Similarly, for a CNF $\varphi$ we denote by $\varphi[x := e]$ the CNF which results from $\varphi$ by substituting $e$ for all appearances of $x$ (and $1-e$ for all appearances of $\overline{x}$) in $\varphi$. A partial assignment is a mapping $p : V \mapsto \{0, 1, *\}$, where $V$ is the set of variables and the value $*$ means an unspecified value. Alternatively, we will also identify a partial assignment with a set of literals $S$ where $x \in S$ if $p(x) = 1$, $\overline{x} \in S$ if $p(x) = 0$, and $\{x, \overline{x}\} \cap S = \emptyset$ if $p(x) = *$.

Two clauses $C_1$ and $C_2$ are said to be *resolvable* if they contain exactly one complementary pair of literals, i.e. if there exists exactly one propositional variable that appears uncomplemented in one of the clauses and complemented in the other. That means that we can write $C_1 = \tilde{C}_1 \vee x$ and $C_2 = \tilde{C}_2 \vee \overline{x}$ for some propositional variable $x$ and clauses $\tilde{C}_1$ and $\tilde{C}_2$ which contain no complementary pair of literals. The clauses $C_1$ and $C_2$ are called *parent clauses* and the disjunction $R(C_1, C_2) = \tilde{C}_1 \vee \tilde{C}_2$ is called the *resolvent* of the parent clauses $C_1$ and $C_2$. Note that the resolvent is a clause (does not contain a propositional variable and its negation). A clause which consists of a single literal is called a *unit clause* and resolution in which one of the parent clauses is a unit clause is called *unit resolution*.

The following is an easy lemma [2].

**Lemma 1.** *Let $C_1$ and $C_2$ be two resolvable implicates of a Boolean function $f$. Then $R(C_1, C_2)$ is also an implicate of $f$.*

Let $\varphi$ be a CNF representing Boolean function $f$, we say, that $C$ can be derived from $\varphi$ by a series of resolutions if there is a sequence of clauses $C_1, \ldots, C_k = C$ such that every $C_i, 1 \leq i \leq k$ either belongs to $\varphi$, or $C_i = R(C_{j_1}, C_{j_2})$, where $j_1, j_2 < i$. It is a wellknown fact (see e.g. [2]) that every prime implicate of $f$ can be derived from $\varphi$. We define *depth* of resolution derivation of $C$ from $\varphi$ as follows.

1. If $C \in \varphi$, then depth of resolution derivation of $C$ is 0.
2. If $C$ can be derived from $\varphi$ by a series of resolutions $C_1, \ldots, C_k = C$, where $C = R(C_i, C_j)$ with $i, j < k$, then depth of the derivation of $C$ is maximum of depths of resolution derivations of $C_i$ and $C_j$ increased by 1.
3. If $C$ cannot be derived from $\varphi$ by a series of resolutions, then we define depth of resolution derivation of $C$ as infinity.

Note, that depth depends on a particular series of resolutions. We say, that $C$ has *resolution depth* $d$ with respect to CNF $\varphi$, if $C$ can be derived by a series of resolutions of depth $d$ and there is no series of resolutions of depth smaller than $d$ which would derive $C$. In particular $C$ has resolution depth 0, if it belongs to $\varphi$ and $C$ has depth 1, if there are clauses $C_1, C_2 \in \varphi$ such that $C = R(C_1, C_2)$.

The wellknown satisfiability problem is defined as follows.

| Satisfiability (SAT) |
|---|
| **Instance :** A formula $\varphi$ in CNF. |
| **Question :** Is there an assignment $t$ to variables of $\varphi$ such that $\varphi(t) = 1$? |

Although this problem is NP-complete in general [3, 11], there are many classes of Boolean formulas, for which SAT problem can be solved in polynomial time. One of these classes is defined using the SLUR (or single-look ahead unit resolution) algorithm [17, 10]. The basic operation used by this algorithm is unit propagation. Function *unitprop*($\varphi$) for a given formula $\varphi$ in CNF returns a pair of values $(\varphi', t)$, where $\varphi'$ is the CNF formula that results from repeatedly performing *unit resolution* until no unit clauses remain in the formula, and $t$ is the partial assignment which satisfies unit clauses found and eliminated during unit propagation. It is known, that unitprop can be implemented in time linear in the length of formula $\varphi$ [8].

**Algorithm 1. SLUR($\varphi$)**

**Input:** A CNF formula $\varphi$ with no empty clause
**Output:** A satisfying partial truth assignment for the variables in $\varphi$, or "unsatisfiable", or "give up".

1: $(\varphi, t) := unitprop(\varphi)$
2: **if** $\varphi$ contains the empty clause **then return** "unsatisfiable" **endif**
3: **while** $\varphi$ is not empty
4: **do**
5: Select a variable $v$ appearing as a literal of $\varphi$
6: $(\varphi_1, t_1) := unitprop(\varphi \wedge \overline{v})$
7: $(\varphi_2, t_2) := unitprop(\varphi \wedge v)$
8: **if** both $\varphi_1$ and $\varphi_2$ contain the empty clause **then return** "give up" **endif**
9: **if** $\varphi_1$ contains the empty clause
10: **then**
11: $(\varphi, t) := (\varphi_2, t \cup t_2)$
12: **else if** $\varphi_2$ contains the empty clause
13: **then**
14: $(\varphi, t) := (\varphi_1, t \cup t_1)$
15: **else**
16: Arbitrarily do one of the following:
17: $(\varphi, t) := (\varphi_1, t \cup t_1)$
18: $(\varphi, t) := (\varphi_2, t \cup t_2)$
19: **enddo**
20: **return** $t$

We shall say, that CNF $\varphi$ is a SLUR CNF, if for all possible sequences of nondeterministic choices in steps 5 and 16, the SLUR algorithm does not give up. If the SLUR algorithm gives up on a CNF $\varphi$ then we shall say that the SLUR algorithm *gives up on a CNF $\varphi$ with a variable $v$ and an assignment $t$*, if $v$ is the last selected variable in step 5 before giving up and $t$ is the assignment to the variables at the time of giving up (not including value of $v$ and values assigned by unit propagation which follows after the selection of $v$). Finally, the SLUR class is the set of all SLUR CNFs.

It was shown in [17] class of SLUR CNFs contains some of well known classes such as (hidden) Horn, (hidden) extended Horn and Balanced formulae.

### 2.1 SLUR($i$)

In this section we recall definition of hierarchy of CNFs defined in [18, 5]. For each fixed $i \geq 1$ we define a class SLUR($i$) as follows. Instead of selecting a single variable on line 5, the parametrized version of the SLUR($\varphi$) algorithm (let us denote it by SLUR($i$,$\varphi$)) nondeterministically selects $i$ variables, and instead of running unit propagation after substituting the two possible values for the selected variable on lines  and  it runs unit propagation on all possible $2^i$ assignments. If all assignments produce the empty clause in the first iteration (after selecting the first $i$-tuple of variables) SLUR($i$,$\varphi$) returns "unsatisfiable". If all assignments produce the empty clause in any of the subsequent iterations SLUR($i$,$\varphi$) gives up. If at least one of the assignments does not produce the empty clause SLUR($i$,$\varphi$) nondeterministically chooses one of such assignments and continues in the same manner. The class SLUR($i$) is then defined as the class of CNFs $\varphi$ on which SLUR($i$,$\varphi$) never gives up regardless of the choices made. Note that the SLUR class is a strict subset of SLUR(1) since every CNF on which the SLUR algorithm gives up after selecting the first variable is not in the SLUR class but belongs to SLUR(1).

It is obvious from the definition, that SLUR($i$,$\varphi$) provides for every fixed $i$ a polynomial time SAT algorithm with respect to the length of the input CNF $\varphi$ (of course, the time complexity grows exponentially in $i$). It is also clear from the definition that every CNF on $n$ variables belongs to SLUR($n$), and hence the hierarchy (i.e. the infinite union of SLUR($i$) classes) contains all CNFs. It was shown in [18, 5], that the SLUR($i$) hierarchy does not collapse, i.e. for every $i \geq 1$ we have SLUR($i$) $\subsetneq$ SLUR($i+1$).

### 2.2 CANON($i$)

The last definition we will need for our main result is the following.

**Definition 1.** *Let $\varphi$ be a CNF and let $f$ be a Boolean function it represents. We say that CNF $\varphi$ belongs to class CANON($i$), where $i \geq 0$, if every prime implicate of $f$ has resolution depth at most $i$ with respect to $\varphi$.*

Note, that if $\varphi \in$ CANON(0), then $\varphi$ contains all prime implicates of $f$ (where $f$ is a Boolean function represented by $\varphi$). If $\varphi$ is moreover prime, it is in fact the canonical representation of $f$. It was shown in [4, 5], that every $\varphi \in$ CANON(0) belongs to the SLUR class. In Section 3 we shall show, that this remains true for CANON(1), but it is not true for CANON(2).

## 3 CANON(1) is a subclass of SLUR

The main result of this paper is an extension to the results in [4, 5] where it is shown, that every canonical formula is in the SLUR class. In fact, it was shown there, that every $\varphi \in$ CANON(0) belongs to the SLUR class, i.e. it is not important for $\varphi$ to be prime, but that it contains all prime implicates of $f$, where $f$ is a Boolean function represented by $\varphi$. We shall start with showing that class CANON(1) is closed under partial assignment.

**Lemma 2.** *Let $\varphi \in$ CANON(1) and let $x$ be any variable of $\varphi$. Then both $\varphi[x := 0]$ and $\varphi[x := 1]$ are also in CANON(1).*

*Proof.* We shall show only the case $x := 0$, the case $x := 1$ is similar. Let us denote $\varphi' = \varphi[x := 0]$, our goal is to prove that $\varphi'$ is in CANON(1), i.e. each of its prime implicates is either in $\varphi'$ or can be derived from it in one resolution step. Let $f$ denote the function represented by $\varphi$ and let $f'$ denote the function represented by $\varphi'$.

Let us fix an arbitrary prime implicate $C'$ of $f'$ and let us show, that $C' \in \varphi'$ or there are two clauses $C_1, C_2 \in \varphi'$ such that $C' = R(C_1, C_2)$. That is exactly the property required for $\varphi'$ to belong to CANON(1) and thus by this our proof will be completed.

Because $\varphi' = \varphi[x := 0]$, we can observe, that $C' \vee x$ is an implicate of $f$. Indeed, let $v$ be an arbitrary assignment satisfying $f$ and let us show, that $C' \vee x$ is satisfied by $v$, too. If $v(x) = 0$, then $v$ satisfies $f'$ and thus $C'(v) = 1$. If $v(x) = 1$, then clearly $(C' \vee x)(v) = 1$. This implies, that there has to be a prime implicate $C$ of $f$ such that

$$C \leq C' \vee x \tag{3}$$

It follows, that $C[x := 0] \leq (C' \vee x)[x := 0] = C'$ and because $C'$ is a prime implicate of $f'$, while $C[x := 0]$ is an implicate of $f'$, we get, that in fact

$$C[x := 0] = C'. \tag{4}$$

If $C \in \varphi$, then clearly $C' = C[x := 0] \in \varphi$.

Let us now assume that $C \notin \varphi$. From our assumption that $\varphi \in$ CANON(1) it follows, that there are $C_1, C_2 \in \varphi$ such that $C = R(C_1, C_2)$. We will divide the proof into several cases.

1. Clause $C$ does not contain variable $x$.
   (a) If the resolution step does not use variable $x$ as a conflict one, then also $x \notin C_1, C_2$, which immediately means that both $C_1 = C_1[x := 0]$ and $C_2 = C_2[x := 0]$ are present in $\varphi'$ including their conflict variable. We can therefore do the same resolution step as before and get $C = C' = R(C_1, C_2)$.
   (b) If on the other hand the resolution step uses $x$ as a conflict variable, then we can write $C_1 = A \vee x$, $C_2 = B \vee \bar{x}$, and $C = R(C_1, C_2) = A \vee B$. for some clauses $A, B$, which do not have a conflict. After substitution to $x$ we get $C_1[x := 0] = A$ and $C_2[x := 0] = 1$. It follows that $C_1[x := 0] = A \leq A \vee B = C \leq C' \vee x$, where the last inequality follows from (3). Now since $C_1 \in \varphi$, we get that $C_1[x := 0] = A \in \varphi'$ and because $C'$ is a prime implicate of $f'$, it must be the case that in fact $C' = A \in \varphi'$.
2. It remains to consider the case when $C$ contains $x$, but $C \notin \varphi$. This case is in fact similar to the case when $C$ does not contain $x$ and it was derived by a resolution which did not use $x$ as a conflict variable. Let us again assume, that $C = R(C_1, C_2)$, where $C_1, C_2 \in \varphi$. Therefore $C_1[x := 0], C_2[x := 0]$ are two resolvable clauses which belong to $\varphi'$ and we have that $C' = R(C_1[x := 0], C_2[x := 0])$.

□

Let us make a simple observation about unsatisfiable clauses from CANON(1).

**Lemma 3.** *If $\varphi \in$ CANON(1), then either $\varphi$ contains an empty clause, or an empty clause is generated during unitprop($\varphi$).*

*Proof.* Let $f$ denote the function represented by $\varphi$. If $\varphi$ is unsatisfiable, then $f$ has the only prime implicate and it is an empty clause, let us denote it $\emptyset$. Due to the assumption that $\varphi \in$ CANON(1), we get that either $\emptyset \in \varphi$, or $\emptyset = R(C_1, C_2)$, where $C_1, C_2 \in \varphi$. In the latter case the only possibility how an empty clause can be generated in one resolution step is, if $C_1 = x$ and $C_2 = \overline{x}$ for some variable $x$ (or symmetrically $C_1 = \overline{x}$ and $C_2 = x$). This means, that an empty clause would be generated during unit propagation. □

Now we have everything ready to prove the main result of this section.

**Theorem 2.** *If $\varphi$ is in CANON(1) then it is also in the SLUR class.*

*Proof.* If $\varphi$ is unsatisfiable, then by Lemma 3 Algorithm 1 (SLUR) would correctly recognize it after unit propagation in step 1. Let us assume, that $\varphi$ is satisfiable. Inductive use of Lemma 2 ensures, that at every step of the algorithm every formula considered belongs to CANON(1). This is because every formula originates from $\varphi$ by partial assignment, this is also true for unit propagation, note also, that using $\varphi \wedge \overline{v}$ corresponds to $\varphi[v := 0]$ and using $\varphi \wedge v$ corresponds to $\varphi[v := 1]$. If at the beginning of the while cycle formula $\varphi$ is satisfiable, then one of $\varphi_1$ and $\varphi_2$ is satisfiable and if one of them is unsatisfiable, it contains an empty clause by Lemma 3. Thus at the beginning of the next cycle $\varphi$ is again satisfiable and at the end the SLUR algorithm finds satisfying assignment. □

Note, that there is a formula $\varphi \in$ CANON(2) which is not SLUR. As an example of such formula we can take

$$\varphi = (x \vee y \vee a) \wedge (x \vee \overline{y} \vee b) \wedge (\overline{x} \vee y \vee c) \wedge (\overline{x} \vee \overline{y} \vee d). \tag{5}$$

It can be checked, that all other implicates which can be derived by resolution from $\varphi$ are the following:

$$(x \vee a \vee b), (y \vee a \vee c), (\overline{y} \vee b \vee d), (\overline{x} \vee c \vee d), (a \vee b \vee c \vee d) \tag{6}$$

Here the last clause has resolution depth 2, and the remaining clauses have resolution depth 1, thus $\varphi \in$ CANON(2). However, it is not SLUR. If the SLUR algorithm first chooses $a$, $b$, $c$, and $d$ and sets them all to 0, then it gets a complete quadratic CNF, which is unsatisfiable and thus the SLUR algorithm would give up. This implies, that it is not true, that CNFs from CANON(2) would all be SLUR.

Above CNF $\varphi$ has another interesting property. It is the only prime and irredundant CNF representing the same function $f$. And it is also the only one, which is not SLUR, it suffices to add any other implicate from list (6) to $\varphi$ to make it SLUR. If e.g. we add $(x \vee a \vee b)$ to $\varphi$, then the SLUR algorithm would recognize an unsatisfiable formula during unit propagation after setting $a$, $b$, $c$, and $d$ to 0 and thus it would not give up. If $x$ or $y$ would be assigned a value before $a$, $b$, $c$, or $d$, or if one of $a$, $b$, $c$, or $d$ would be assigned value 1, then SLUR algorithm would get a satisfiable quadratic formula, which is SLUR. This is an example of a function, that does not have a prime and irredundant SLUR representation.

## 4 SLUR*($i$)

Let us return to SLUR($i$) hierarchy, which was defined in [18, 5]. Let us consider the following formula

$$\varphi = (x \vee \overline{y})(\overline{x} \vee y)(x \vee y \vee a \vee b)(x \vee y \vee \overline{a} \vee b)(x \vee y \vee a \vee \overline{b})(x \vee y \vee \overline{a} \vee \overline{b}) \tag{7}$$

It can be observed, that this formula is not in the SLUR(2) class, if we choose $x$ and $y$ and then we choose assignment $x = y = 0$, then the SLUR(2) algorithm gives up as it is left with a complete and thus unsatisfiable quadratic formula. Here SLUR(2) actually does not take any advantage from the fact that it can choose two variables at once, because it chooses two equivalent variables. If however after choosing a value for $x$, the SLUR(2) algorithm would be allowed to perform unit propagation, then it would not choose $y$ as the second variable and it would recognize, that in case $x = 0$ the rest is an unsatisfiable formula. This example leads us to hierarchy consisting of classes SLUR*($i$), in which the algorithm also chooses $i$ variables at each step, but it performs unit propagation between each of these choices rather then after all of them. Formally, SLUR*($i$) class is defined using Algorithm 4.

**Definition 2.** *Function $F$ is a member of SLUR*(i) class if Algorithm 4 does not return "give up" for any of nondeterministic choices made during its run.*

Firstly we will show the recursive function (Algorithm 3) that takes care of searching the $i$-decision assignments and then the SLUR*($i$) algorithm.

**Algorithm 3. test($\varphi$, $k$)**

**Input:** A CNF formula $\varphi$, number of decision $k$ which remain to be made by the algorithm.
**Output:** A partial assignment which have not led to an empty clause after $k$ decisions, or UNSAT if no such assignment exists.

1: $(\varphi, t) := unitprop(\varphi)$
2: **if** $\varphi$ contains an empty clause **then return** UNSAT **endif**
3: **if** k=0 **then return** empty assignment **endif**
4: $e$ :=an undetermined literal (positive or negative)
5: $t'_1 := test(\varphi_1, k-1)$
6: **if** previous test did not return UNSAT **then return** $t \cup t_1 \cup t'_1$ **endif**
7: $t'_2 := test(\varphi_2, k-1)$
8: **if** previous test did not return UNSAT **then return** $t \cup t_2 \cup t'_2$ **endif**
9: **return** UNSAT

**Algorithm 4. SLUR*($i$, $\varphi$)**

**Input:** A CNF formula $\varphi$ without any empty clause
**Output:** A partial truth assignment satisfying $\varphi$, "unsatisfiable", or "give up".

1: $(\varphi, t) := unitprop(\varphi)$

```
2:  if φ contains an empty clause then return "unsatisfiable" endif
3:  while φ is not empty
4:  do
5:    t' :=test(φ, i)
6:    if previous test returned UNSAT
7:    then
8:      if it is the first run of the while cycle
9:      then
10:       return "unsatisfiable"
11:     else
12:       return "give up"
13:     endif
14:   endif
15:   t := t ∪ t'
16: enddo
17: return t
```

Note, that all nondeterminism is now stored in step 4 of procedure *test* (Algorithm 3). In this step by choosing literal instead of a variable we also give no preference to whether the first value tested will be 1 or 0. The test procedure is in fact a DPLL procedure (see [16] for details), in which we bound our search by given depth. If $i$ is a fixed constant, algorithm SLUR*($i$) runs in polynomial time, though it is naturally exponential with increasing $i$.

It is easy to show that the SLUR*(i) hierarchy does not collapse, i.e. for every $i \geq 1$ we have

$$\text{SLUR*}(i) \subsetneq \text{SLUR*}(i+1),$$

in fact exactly the same argumentation as for the previous SLUR($i$) hierarchy can be used [5, 18]. Due to space limitation, we will omit the details here. It can be immediately seen, that every CNF $\varphi$ on $n$ variables belongs to SLUR*(i). It can be also observed that by doing unit propagation before each choice, we do not loose anything and thus

$$\text{SLUR}(i) \subseteq \text{SLUR*}(i)$$

for every $i \geq 1$, in particular SLUR $\subseteq$ SLUR*(1). The example formula defined in (7) at the beginning of this section shows, that in fact SLUR(2) $\subsetneq$ SLUR*(2), this example can in fact be generalized to show the following lemma. (Note, that in case $i = 1$ we do not gain anything and thus SLUR(1) = SLUR*(1)).

**Lemma 4.** *For every $i > 1$ we have SLUR($i$) $\cap$ SLUR*(2) $\neq \emptyset$.*

*Proof.* Let $i > 1$ be a fixed constant and let us consider the following formula:

$$\begin{aligned}\varphi = &(y_1 \vee \overline{y_2}) \wedge (y_2 \vee \overline{y_3}) \wedge \ldots \wedge (y_{i-1} \vee \overline{y_i}) \wedge (y_i \vee \overline{y_1}) \\ &\wedge(y_1 \vee \ldots \vee y_i \vee a \vee b) \wedge (y_1 \vee \ldots \vee y_i \vee \overline{a} \vee b) \\ &\wedge(y_1 \vee \ldots \vee y_i \vee a \vee \overline{b}) \wedge (y_1 \vee \ldots \vee y_i \vee \overline{a} \vee \overline{b})\end{aligned}$$

This formula is logically equivalent to

$$(y_1 \leftrightarrow y_2 \leftrightarrow \ldots \leftrightarrow y_i) \wedge \{y_1 \vee \ldots \vee y_i \vee [(a \vee b) \wedge (a \vee \bar{b}) \wedge (\bar{a} \vee b) \wedge (\bar{a} \vee \bar{b})]\}.$$

If the SLUR($i$) algorithm chooses at first the $i$-tuple $y_1, \ldots, y_i$, and then it sets all these variables to 0, then it will get an unsatisfiable complete quadratic formula on variables $a$ and $b$, which means, the SLUR($i$) algorithm will give up.

On the other hand Algorithm 4 which performs unit propagation after each pick of a variable will assign equivalent values to all $y_1, \ldots, y_i$ variables after it will come across the first one of them. So it can use the remaining step to deal with the rest of the formula. Again, no problem can arise, if the first chosen variable is $a$ or $b$. This means that formula $\varphi$ belongs to SLUR*(2). □

The following is now an easy corollary.

**Corollary 1.** *For every $i > 1$ we have that SLUR($i$) $\subsetneq$ SLUR*($i$).*

Let us now return to the classes CANON($i$). Let $\varphi$ be CNF defined in (5), we have seen that this CNF belongs to CANON(2), but it is not SLUR, now we can even observe, that $\varphi$ does not belong to SLUR(2), this is because if the SLUR(2) algorithm chooses first $a$ and $b$ and sets them to 0, then $c$ and $d$ and sets them to 0, what remains is a complete unsatisfiable quadratic formula. Moreover, in this case unit propagation after choosing value for $a$ or $c$ does not help and thus $\varphi$ does not even belong to SLUR*(2). By concatenating copies of $\varphi$ by disjoint union, we could in fact get an example of a formula showing the following lemma (we omit formal proof due to the space limitations).

**Lemma 5.** *For every $i \geq 1$ we have that $CANON(2) \cap SLUR^*(i) \neq \emptyset$.*

## 5 Conclusion

In this paper we concentrated on SLUR formulae and their possible generalizations. We have related newly defined hierarchies CANON($i$) and SLUR*($i$) to hierarchy SLUR($i$) defined in [18, 5]. We leave as an open question, whether there is some natural generalization of SLUR class which would contain hierarchy CANON($i$) and whether there is a satisfiability algorithm for each of these classes, which would be polynomial for a fixed $i$. Following definition of CANON($i$), there is such an algorithm based on resolution, so what we would like to see, is a satisfiability algorithm based on unit propagation and the SLUR algorithm.

## References


1. B. Aspvall. Recognizing disguised nr(1) instances of the satisfiability problem. *Journal of Algorithms*, 1(1):97 – 103, 1980.
2. Hans K. Buning and T. Letterman. *Propositional Logic: Deduction and Algorithms*. Cambridge University Press, New York, NY, USA, 1999.
3. Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC '71: Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158, New York, NY, USA, 1971. ACM.
4. O. Čepek and P. Kučera. Various notes on SLUR formulae. *Proceedings of the 13th Czech-Japan Seminar on Data Analysis and Decision Making in Service Science*, pp. 85 – 95, Otaru, Japan, 2010.
5. Ondřej Čepek, Petr Kučera and Václav Vlček. Properties of SLUR formulae (sent to SOFSEM 2011).
6. V. Chandru and J. N. Hooker. Extended horn sets in propositional logic. *J. ACM*, 38(1):205–221, 1991.
7. Michele Conforti, Gérard Cornuéjols, and Kristina Vuskovic. Balanced matrices. *Discrete Mathematics*, 306(19-20):2411–2437, 10 2006.
8. Mukesh Dalal and David W. Etherington. A hierarchy of tractable satisfiability problems. *Information Processing Letters*, 44(4):173–180, 12 1992.
9. W.F. Dowling and J.H. Gallier. Linear time algorithms for testing the satisfiability of propositional horn formulae. *Journal of Logic Programming*, 3:267 – 284, 1984.
10. John Franco and Allen Van Gelder. A perspective on certain polynomial-time solvable classes of satisfiability. *Discrete Appl. Math.*, 125(2-3):177–214, 2003.
11. M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, San Francisco, 1979.
12. M.R. Genesereth and N.J. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann, Los Altos, CA, 1987.
13. A. Itai and J.A. Makowsky. Unification as a complexity measure for logic programming. *Journal of Logic Programming*, 4:105 – 117, 1987.
14. Harry R. Lewis. Renaming a set of clauses as a horn set. *J. ACM*, 25(1):134–135, 1978.
15. M. Minoux. Ltur: A simplified linear time unit resolution algorithm for horn formulae and computer implementation. *Information Processing Letters*, 29:1 – 12, 1988.
16. D. Putnam, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.
17. John S. Schlipf, Fred S. Annexstein, John V. Franco, and R. P. Swaminathan. On finding solutions for extended horn formulas. *Inf. Process. Lett.*, 54(3):133–137, 1995.
18. Václav Vlček. Třídy s efektivně řešitelným SATem (master thesis, in Czech). *Charles university in Prague*, 2009.