Charles University in Prague

Faculty of Mathematics and Physics

DOCTORAL THESIS

Václav Vlček

Classes of Boolean Formulae with Effectively Solvable SAT

Department of Theoretical Computer Science
and Mathematical Logic

Supervisor of the doctoral thesis: doc. RNDr. Ondřej Čepek, Ph.D.

Study programme: Computer Science

Specialization: Theoretical Computer Science (I1)

Prague 2013

I would like to thank to my supervisor Ondřej Čepek for kind, helpful and encouraging approach and lots of constructive ideas to the preliminary versions of this thesis. I would also like to thank to other attendants of the Seminar on Boolean functions for their participation on the joint research, namely Martin Babka, Tomáš Balyo, Štefan Gurský and Petr Kučera.

I dedicate this thesis to my parents.

Název práce: Třídy booleovských formulí s efektivně řešitelným SATem

Autor: Václav Vlček

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí disertační práce: doc. RNDr. Ondřej Čepek, Ph.D., Katedra teoretické informatiky a matematické logiky

Abstrakt: Práce studuje třídy booleovkských formulí pro které je problém splnitelnosti řešitelný v polynomiálním čase. Zaměřuje se na třídy založené jednotkové rezoluci; popisuje třídy unit refutation complete formulí, unit propagation complete formulí a specialně se zaměřuje na třídu SLUR. Shrnuje její vlastnosti a poslední výsledky dosažené v této oblasti. Hlavním výsledkem je coNP-úplnost testování zda daná formule patří do třídy SLUR. V závěru je třída SLUR rozvinuta do několika různých hierarchií a jsou studovány jejich vlastnosti a vzájemný vztah vzhledem k inkluzi.

Klíčová slova: booleovské formule, splnitelnost, SAT, single lookahead, jednotková rezoluce

Title: Classes of Boolean Formulae with Effectively Solvable SAT

Author: Václav Vlček

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: doc. RNDr. Ondřej Čepek, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: The thesis studies classes of Boolean formulae for which the well-known satisfiability problem is solvable in polynomially bounded time. It focusses on classes based on unit resolution; it describe classes of unit refutation complete formulae, unit propagation complete formulae and focuses on the class of SLUR formulae. It presents properties of SLUR formulae as well as the recently obtained results. The main result is the coNP-completness of membership testing. Finally, several hierarchies are built over the SLUR class and their properties and mutual relations are studied.

Keywords: Boolean formulae, satisfiability, SAT, single lookahead, unit resolution

Contents

Introduction

A problem of testing satisfiability of a Boolean formula (SAT) is one of the best known NP-complete problems. In fact it was the first one proven to be NP-complete [9]. Since Boolean formulae are widely used as a natural encoding language for many problems of computer science, and contemporary SAT solvers are rather effective when solving the problem, Boolean functions and testing their satisfiability appears in many areas such as planning, scheduling, automated theorem proving, program verification, etc.

There are two main directions in contemporary research to tackle the SAT problem effectively. The goal of the first one is to design a general SAT solver which works fast on most of formulae, but does not guarantee a polynomial time bound (achieving such a bound is considered very unlikely thanks to the NP-completeness of the problem). The other direction is to isolate a subclass of formulae and design a specific algorithm for this class that is guaranteed to work in polynomial time. This approach is well justified in a situation when we deal with a problem which naturally leads to formulae of specific properties, or if we are able to choose the target encoding language (as for example in the case of knowledge compilation).

In this dissertation we will mainly focus on the second approach. We will concentrate on the class of Single Lookahead Unit Resolution formulae (SLUR) [31, 18]. It is a class which contains many other known classes such as Horn formulae [17, 32], hidden Horn [2, 27], extended Horn [7] and CC-balanced [8] formulae. The class is interesting from the point of view of the general SAT solvers as well, because most of these solvers are guaranteed to work fast on SLUR formulae.

In Chapter 1 we will define the necessary notation, sum up some general results about Boolean formulae, and also describe the well-known resolution method for obtaining a canonical formula. In Chapter 2 we focus on a special case of resolution called unit resolution, which is a key part of almost any contemporary general SAT solver. We also present the algorithm for deciding the satisfiability of formulae from the SLUR class. We will also describe a method of clause learning [28] which speeds up the performance of general SAT solvers considerably, and which will bring us to the definitions of unit refutation complete formulae [16] and the propagation complete formulae [6]. In Chapter 3 the SLUR class is defined and some general results about this class are derived. The main result about the SLUR class is that the recognition of a formula membership is a coNP-complete problem. On the other hand, a sufficient condition is proven there: every formula which allows to generate every prime implicate by at most one resolution step belongs to the SLUR class. In Chapter 4 several hierarchies built over the SLUR class are studied. These hierarchies extend the SLUR class such that they contain more formulae, but the time needed for satisfiability testing increases exponentially depending on the layer of the hierarchy, however, the testing time remains polynomial at any fixed level. One of these hierarchies [21, 22] constitutes a hierarchy over unit refutation complete formulae as well, and as a consequence of the fact that the initial layer coincides with the SLUR class and the class of unit refutation formulae, the authors of [21, 22] showed these two classes to be equal. The dissertation finishes with a brief conclusions chapter.

1. Notation and well known results

In this section we define a collection of standard terms from the area of Boolean functions and sum up some known basic results we will need further in the text.

Boolean function on n variables is any function $f : \{0,1\}^n \to \{0,1\}$. For a Boolean function $f : \mathcal{V} \to \{0,1\}$ a mapping $v : \mathcal{V} \to \{0,1,*\}$ assigning values to variables of f is called *partial assignment*; if all variables are assigned values from $\{0,1\}$, we call v just *assignment*. The value '$*$' denotes that a given variable does not have an assigned value from $\{0,1\}$. However, the function value can be still determined by a partial assignment in the case that the function values for all extensions of the given partial assignment into an assignment are equal. By $f[v]$ we denote the fact that we substitute (partial) assignment v into function f. If v substitutes into one variable x only, we write $f[x = i]$ where $i \in \{0,1\}$ is the value assigned. For two partial assignments v_1 and v_2 expression $v_1 \cup v_2$ denotes the partial assignment we obtain by taking v_1 and adding those assignments of v_2 which substitutes to variables not assigned by v_1.

We say that Boolean function f is *satisfiable* if there is an assignment $\overrightarrow{x} \in \{0,1\}^n$ to the variables of f such that $f(\overrightarrow{x}) = 1$, in the other case we say that f is *unsatisfiable*.

As in case of real valued functions, we can define a partial ordering on Boolean functions in the following way. Let f and g be two Boolean functions on n variables, we write $f \leq g$ if for every assignment $\overrightarrow{x} \in \{0,1\}^n$ it holds $f(\overrightarrow{x}) \leq g(\overrightarrow{x})$, i.e. if every assignment which satisfies f satisfies g too.

There are many possibilities how to represent Boolean functions. As any function on a finite domain, they can be represented by a table of all possible assignments and the corresponding function values. Other common ways are to use decision trees, binary diagrams or Boolean circuits (see e.g. [33, 38] for further details on some of possible representations). Even though these representations do have many advantages, we will focus mainly on Boolean functions represented by a propositional logic formula.

It is well known (see e.g. [20]), that any Boolean function can be represented by a propositional formula in a so called *conjunctive normal form* (or *CNF*, for short). The usual proof proceeds as follows. Let us have a Boolean function given by the following function table.

$\mathbf{x_1}$	$\mathbf{x_2}$	$\mathbf{x_3}$	$\mathbf{f(x_1, x_2, x_3)}$
0	0	0	1
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

For every row for which the function value equals 0 we take a disjunction of all variables such that we put x_i into it if the corresponding column contains 0 and $\overline{x_i}$ if it contains 1. Then we connect all these disjunctions by conjunctions. For the function f from the example bellow we get formula $F_{\mathrm{CNF}} = (x_1 \vee \overline{x_2} \vee \overline{x_3}) \& (\overline{x_1} \vee x_2 \vee \overline{x_3}) \& (\overline{x_1} \vee \overline{x_2} \vee x_3)$.

We can construct yet another formula if we consider lines where f evaluates to 1, take conjunction of all variables x_i for ones and $\overline{x_i}$ for zeros and finally take a disjunction of all these conjunctions. Formulae of this form are said to be in *disjunctive normal form* (*DNF*). This formula for the example above the DNF formula is of the following form: $F_{\mathrm{DNF}} = (\overline{x_1} \& \overline{x_2} \& \overline{x_3}) \vee (\overline{x_1} \& \overline{x_2} \& x_3) \vee (\overline{x_1} \& x_2 \& \overline{x_3}) \vee (x_1 \& \overline{x_2} \& x_3) \vee (x_1 \& x_2 \& x_3)$.

Almost all results about DNFs have a corresponding result in the CNF formalism, therefore we will use CNF as they are more common. Let us define CNF formulae and the satisfiability problem formally.

1.1 Terms connected to the CNF formulae

Definition 1. Propositional variable x or its negation $\overline{x}$ is called a *literal* (the variable without negation is a *positive literal* and the other one is a *negative literal*). *Clause* is a finite disjunction of literals and formula is in *conjunctive normal form* (*CNF*) if it is formed by a finite conjunction of clauses.

By $F[x = i]$ we will denote the formula we obtain from formula F by replacing literal x by value i to and $\overline{x}$ by the opposite value. For a partial assignment $v : \mathcal{V} \to \{0, 1, *\}$ assigning values to the set $\mathcal{V}$ of variables of F, the expression $F[v]$ denotes the formula we obtain by applying assignments of corresponding values to all variables of $\mathcal{V}$ which are assigned either 0 or 1 by v.

We will restrict our attention to clauses which do not contain variable and its negation at the same time. This is not restrictive because such clauses can be found in linear time in the CNF length and every such clause is always satisfied. In contrast to that, we permit clauses without any literal (*empty clauses*, usually denoted by $\perp$) and CNF formulae without a clause (*empty formulae*). Empty clause is considered as unsatisfiable, as there is no literal in the empty disjunction which could be satisfied, while empty formula is always satisfied by definition, because there is no violated constraint in the conjunction.

Let us also note that because there is no point of using a literal more than once in a clause, and the order does not matter, we can see clauses as a set of its literals. In the same way we can see CNF formulae as sets of its clauses. We will use set and formula notation interchangeably as there is no danger of confusion.

Boolean function can have more than one CNF representations. For example, identically zero function can be represented by $x \;\&\; \overline{x}$ or by $(x_1 \vee x_2) \;\&\; (x_1 \vee \overline{x_2}) \;\&\; (\overline{x_1} \vee x_2) \;\&\; (\overline{x_1} \vee \overline{x_2})$ and in many other ways.

Probably the most studied problem in computer science is the problem of Boolean satisfiability. It was also the first proven to be NP-complete [9, 19] and it will be one of the main topics of this thesis as well. The problem is formally defined as follows.

Satisfiability (SAT)	
Instance :	A formula F in CNF.
Question :	Is there an assignment v to the variables of F such that $F(v) = 1$?

We can see clauses of a CNF as constraints given on a solution of a problem described by a Boolean formula. In order to simplify the problem, it is sometimes handy to consider the ordering '$\leq$' we have defined on Boolean function also for Boolean formulae (and as every literal and every clause are also formulae we can use the notion here as well). This notation allows as to specify some types of clauses which are clearly unnecessary in a CNF formula and can be opted out without changing the function.

Definition 2. We say that clause C_1 *subsumes* clause C_2 if $C_1 \leq C_2$. Clause C is an *implicate* of a Boolean function f if $f \leq C$, it is a *prime implicate* if there is no other implicate C' of f subsuming C.

As can be checked by considering the satisfying assignments, the fact that C_1 subsumes C_2 is in reality equivalent to the claim that every literal of C_1 is contained also in C_2 (i.e. $C_1 \subseteq C_2$). Note that every clause of a CNF formula is an implicate of the function which the formula represents. It is also easy to see that we can omit clauses which are subsumed by other clause in the formula, and the prime implicates are the clauses that cannot be shortened in terms of the number of literals without changing the represented function.

Definition 3. A Boolean formula which consists only of prime implicates is called a *prime formula.* A formula which contains exactly all the prime implicates is known as the *canonical formula.* If there is no clause which could be removed from a formula without changing the represented function then the formula is called *irredundant.*

Example 4. The following three formulae represents same function:

1. $F_1 = (x \vee \overline{y})$ & $(\overline{x} \vee y)$ & $(x \vee z)$,
2. $F_2 = (x \vee \overline{y})$ & $(\overline{x} \vee y)$ & $(x \vee z)$ & $(y \vee z)$ and
3. $F_1 = (x \vee \overline{y})$ & $(\overline{x} \vee y)$ & $(x \vee y \vee z)$.

Their equivalence is clear from the fact that first two clauses give us the equivalence of variables x and y. The first two formulae are prime and the second even canonical, whereas the third one is not prime because its last clause is subsumed by a implicates $(x \vee z)$ and $(y \vee z)$.

Canonical formulae (or more precisely formulae containing canonical formulae as subsets) have an interesting property with respect to partial assignments as stated in the following lemma. This lemma will be later generalized for a larger class of formulae in section 3.3 (Lemma 33).

Lemma 5. *[36] Let F' be the canonical formula for Boolean function f. Let $F \supseteq F'$ be a formula representing the same function f and let x be any of its variables. Then formula $F[x = i]$ that we get by assigning $i \in \{0, 1\}$ to x contains the canonical representation of the restricted function $f[x = i]$ as a subformula.*

Proof. Let C be a prime implicate of $f[x=0]$. Then $C \vee x$ is implicate of f and thus there is a prime implicate C' of f in F such that $C' \leq C \vee x$. We will treat two cases separately:

1. $x \notin C'$: Then $C' \leq C$ also holds. Because C is the prime implicate of $f[x=0]$ and C' is a prime implicate of f and because C' does not depend on x we have also $C \leq C'$. That implies $C = C'$ and because $C'[x=0] = C'$ we have proved that C is contained in $F[x=0]$.

2. $x \in C'$: Than $C' = C'' \vee x$ for some C''. Because $C'' \vee x = C' \leq C \vee x$ we have $C'' \leq C$. But we have $C'[x=0] = C'' \leq (C \vee x)[x=0] = C$ and C is a prime implicate of $f[x=0]$. Thus follows that $C'' = C$ and because $C'[x=0] = C'' = C$ we have proved that C is contained in $F[x=0]$.

We have proved that an arbitrarily chosen prime implicate C of $f[x=0]$ is contained in $F[x=0]$. The case assigning $x=1$ is similar. $\square$

In the following section we present a method how to generate the canonical formula from a formula on input. It will be clear that the canonical formula can be in general exponentially longer than the input formula if we measure the length by the number of literal occurrences in the formula.

1.2 Resolution method

Definition 6. We say that two clauses have a *conflict in a variable x*, if x occurs as a positive literal in one of them and as a negative literal in the other. Two clauses $C_1 = (A \vee x)$ and $C_2 = (B \vee \overline{x})$ having a conflict in exactly one variable x are said to have a *resolution over x* (or to *be resolvable*). Their *resolvent* is clause $C = R(C_1, C_2) = (A \vee B)$.

The concept of resolution (see e.g. [29, 30]) can be used to generate further implicates from a formula, as it is clear from the following lemma. As we will see later, we can actually find all the prime implicates of the represented function.

Lemma 7. *[10] Let C_1 and C_2 be two resolvable clauses of a Boolean formula F which have resolvent $C = R(C_1, C_2)$. Then C is an implicate of F and therefore F & C represents the same function as F.*

Proof. We will prove $F \leq C$ by considering the assignments to the variables contained in C_1 and C_2. Let x be the variable in which C_1 and C_2 have a conflict.

Let us have a satisfying assignment of F. It has to satisfy clauses C_1 and C_2, too. If there are satisfied literals in both clauses that are over variables other than x, then C is also satisfied as it contains these literals as well. If one of clauses C_1 and C_2 is satisfied thanks to the valuation of x, let say it is clause C_1. Then C_2 cannot be satisfied thanks to x, hence there must be other literal satisfying C_2 and this literal will be preserved after the resolution and satisfies C. $\square$

As we said before, we can add all generated implicates to the formula without changing the function represented. But, if we generate an implicate that is subsumed by a clause already contained in F, there is no point of adding. On the other hand, if some clause of F became subsumed, we can omit it without changing the represented function. In fact, by performing all resolutions possible and eliminating subsumed clauses the canonical formula can be generated. This procedure, called *resolution method*, was described in [29, 30] performs as follows.

Algorithm 1 Resolution method

```
Input: CNF formula $F$.
Output: Canonical CNF formula representing the same function as $F$.
  $F := F$ with subsumed clauses omitted
  while ($F$ contains resolvable clauses $C_1$ and $C_2$ such that $R(C_1, C_2)$ is not
  subsumed by any clause in $F$) do
    $F := F \cup R(C_1, C_2)$
    $F := F$ with subsumed clauses omitted
  end while
  return $F$
```

We have yet prove that the procedure finally stops and all the prime implicates are generated.

Theorem 8. *[10] Resolution method returns canonical representation of the function represented by input CNF formula.*

Proof. It is clear that the algorithm is finite, because no implicate will be added more than once and it either stays in the formula till the end of computation or it is replaced by a subsuming (shorter) clause and never added again.

We will show the fact that the output formula contains all the prime implicates by contradiction. Let F be the formula returned by the algorithm and let C be a prime implicate which is not contained in F. Let I_C be set of all the implicates of the function which are subsumed by C but are not subsumed by any implicate in F, i.e. $I_C = \{C' : (C \subseteq C') \ \& \ [(\forall C'' \in F) \ C'' \not\subseteq C']\}$. Let us note that I_C is non-empty, because it contains C. Hence, we can take a clause in I_C which contains most literals, let it be clause C^*. C^* is the longest clause subsuming C which yet is not subsumed by any clause presented in F. We will consider two cases:

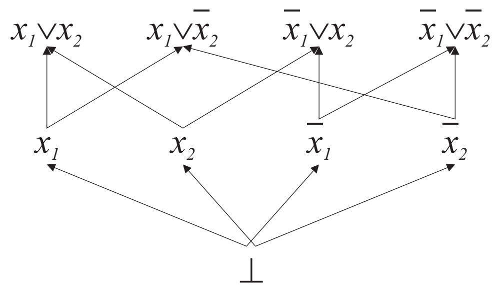
a) C^* contains all n variables appearing in F. In this case every clause of F has a conflict with C^* thanks to the $(\forall C'' \in F) \ C'' \not\subseteq C^*$ condition. Therefore, if C^* evaluates to 0, F evaluates to 1 thank to the fact that every clause contains at least one literal of opposite polarity than in C^* which satisfies it.
On the other hand, it holds that if C^* evaluates to 0, then C evaluates to 0 as well (see $C \subseteq C'$ condition). But that yields that F evaluates to 0, because C in an implicate. This gives us the contradiction.
b) C^* contains less than n variables. Let x and $\overline{x}$ be literals over a variable not contained in C^*, i.e. neither $C^* \vee x$ nor $C^* \vee \overline{x}$ is contained in I_C. It can be only so that they violate the $(\forall C'' \in F) \ C'' \not\subseteq F$ condition. Let A be a clause of F such that $A \subseteq C^* \vee x$ and similarly let B be such that

$B \subseteq C^* \vee \overline{x}$. As $C^* \in I_C$, we have that $x \in A$ and $\overline{x} \in B$. The resolvent R(A, B) of A and B is an implicate not subsumed by any clause in F which is a contradiction with the termination of the algorithm.

□

It is quite obvious the that resolution method can be used for solving the satisfiability problem. It is an easy consequence of the fact that a formula is unsatisfiable if and only if its only prime implicate is the empty clause. The drawback of this way of satisfiability testing is its complexity. There are formulae which for which the canonical representations contains exponentially many clauses and therefore the resolution algorithm performs exponentially many steps.

Example 9. Let $V = \{x_1, \dots x_n\}$ be a set of variables, and let us define a formula $F = \bigwedge_{P \subseteq V} [\bigvee_{x_i \in P} x_i \vee \bigvee_{x_i \in V \setminus P} \overline{x_i}]$. This formula is unsatisfiable, it contains all possible clauses on n variables and does not leave any room for satisfying assignments. The run of resolution algorithm for two variables is shown in Figure 1.1. Its vertices correspond to the clauses, the top layer consist of the clauses of the input formula, and each further layer contains clauses that can be derived by one resolution step from the previous (in case we always generate all clauses possible and after that we add them to the formula altogether).



$x_1 \vee x_2$
$x_1 \vee \overline{x_2}$
$\overline{x_1} \vee x_2$
$\overline{x_1} \vee \overline{x_2}$
x_1
x_2
$\overline{x_1}$
$\overline{x_2}$
$\perp$

Figure 1.1: Resolution graph of complete quadratic formula

Definition 10. Let F be a formula in CNF and let C be a clause derivable from F by series of resolutions. A *resolution graph* is an oriented connected acyclic graph where each vertex corresponds to a clause, every vertex of non-zero out-degree corresponding to clause C_R has two child nodes corresponding to C_1 and C_2 such that $C_R = R(C_1, C_2)$, every leaf (i.e. vertex with out-degree zero) corresponds to some clause of F and clause C is contained in a root. *Resolution tree* is a special case where the resolution graph is a tree.

Resolution tree can be obtained from a resolution graph by splitting shared vertices into copies and omitting vertices to which there is no path from derived clause C.

Example 11. Let now $F = (x_1 \vee \overline{y_1}) \& \dots \& (x_n \vee \overline{y_n}) \& (\overline{x_1} \vee \dots \overline{x_n})$, its length measured by number of literal occurrences clearly equals to $3n$. Each of the quadratic clauses is resolvable with clause $(\overline{x_1} \vee \dots \overline{x_n})$, which yields a resolvent of length n in which one x-variable is replaced by the corresponding y-variable. This resolvent clearly is not absorbed by any clause. The quadratic clauses cannot absorb it because each of them contains a positive literal, which is not the case of the resolvent, and therefore they obviously cannot be its subset. On the other hand, every non-quadratic clause contains n literals, because there are only resolutions which replace some x-variable by y-variable or other way round.

Hence, the canonical formula consists of all the quadratic clauses presented in the initial formula F and all clauses of length n of the form $(\overline{z_1} \vee \ldots \vee \overline{z_n})$ where $z_i \in \{x_i, y_i\}$. The overall length of the canonical formula measured by number of literal occurrences therefore equals to $2n + n \cdot 2^n$. We have that the canonical formula is $\frac{2n+n\cdot 2^n}{3n} = \Theta(2^n)$ times longer than the input formula, which means that the resolution algorithm has to perform exponentially many steps.

There are occasions when we are interested in the complexity of derivation of some concrete implicate. One of the most natural measures is the minimum length of *resolution proof of clause C from formula F*, i.e. series $C_1, C_2, \ldots, C_k = C$ where every clause in the series is either clause of formula F or is a resolvent of two preceding clauses. In this case the number k is called *length* of this resolution proof. For example, for the pigeon hole principle formulae (see [23]) the minimum length of resolution proof of $\perp$ is exponentially large in the length of the input formula.

Other possible measure of resolution complexity is the *depth of resolution.* It corresponds to the minimal possible layer in the resolution graph on which a certain clause appears. This measure has a trivial upper bound for unsatisfiable formulae [35], because the contradiction can be derived in depth no larger than the number of variables of the formula (this result is a consequence of a result of [34] stating that for any tree resolution proof there is a tree resolution proof where on any path from root to leave no variable is resolved more than once). We will, however, need this definition just for prime implicates in order to define a hierarchy of formulae we is needed further in the text.

Definition 12. Given a formula F, the depth of resolution for an implicate C is the least possible number we can get by following procedure:

1. If $C \in F$ then its depth is 0.
2. If C is derived by a series of resolutions $C_1, \ldots, C_k = C$ as a resolvent of some preceding clauses C_i and C_j, then its depth is equal to maximum of depths of C_i and C_j increased by 1.

The hierarchy of formulae we will use is then defined in the following way. It has been established in [4].

Definition 13. We say that a Boolean formula F belongs to class CANON(i) if every prime implicate of the function which F represents can be derived by a series of resolutions of depth i.

For example canonical formulae are just the formulae of CANON(0). Class CANON(1) then contains all the formulae from which each of their prime implicates can be obtained by at most one resolution step.

Sometime it can be interesting to consider restricted variants of resolution. One of such restrictions is a requirement that each time we perform a resolution step, one of the clauses is from the original formula. This variant of resolution is called *input resolution*. Other possible restriction is so called *non-merge resolution* where we require that resolved clauses do not have common literal. Yet another variant is so called *unit resolution* which will be explained thoroughly in the next section, because it is a key part of many SAT-solvers and it is also very important for definition of the classes of Boolean formulae we will focus on.

2. Unit resolution and refutation

A clause which contains just one literal is usually called a *unit clause*. If there is such a clause in a formula, then it forces a value that the corresponding variable has to have in every satisfying assignment. When our goal is to find such an assignment, we can replace every occurrence of the variable by the forced value. After that we can omit all 0 from the clauses because it cannot help with satisfying in any way. We can also withdraw all the clauses that already has became satisfied. If some of the clauses became empty, we can conclude that the formula is not satisfiable and nor was the original one because the value assigned was forced.

The whole process described above is actually equivalent to the situation of resolution when one of the clauses used is a unit clause. Such restricted variant of resolution is called *unit resolution*. If we proceed further and repeatedly perform until there is no unit clause, we get a procedure that is commonly called *unit propagation* (or *unitprop* for short). We use symbol $F \vdash_1 C$ to denote the fact that clause C can be obtained from F by unit propagation.

It is known that unit propagation can be performed in a linear time in length of the input formula [12]. Thanks to this effectiveness and thanks to the fact it speeds up computation considerably, unit resolution is used in almost every contemporary procedure for deciding SAT problem on general formulae. Such a procedure is usually called SAT-solver and we will discuss some basics in the next section.

2.1 SAT solvers

The most basic idea to test satisfiability of a formula over n variables is to try all 2^n possible assignments. As we mentioned above, this can be speed up by performing unit resolution after every assignment of a value to a variable. This usually saves up some decisions and lowers the amount of branching. The method is known as Davis-Putnam procedure [15]. It was refined into DPLL-procedure [14] which uses depth-first-search approach in order to decrease space complexity to linear in number of variables.

DPLL procedure is a keystone of almost any contemporary SAT-solver. It is formally described in Algorithm 2. The procedure called *unitprop* performs unit propagations and returns a pair consisted of the simplified formula and the assignments forced during the procedure. Some variants of DPLL also contain a step where they assign a value to the variables which occur in the whole formula just as a positive or just as a negative literal, but this step is not usually used in the modern SAT-solvers, because in practice it slows down the computation rather than speeds it up.

Example 14. In the Figure 2.1 you can see a search tree of the DPLL procedure on a formula $F = (\overline{x_1} \vee x_6 \vee x_7)$ & $(x_6 \vee \overline{x_7})$ & $(\overline{x_6} \vee x_7)$ & $(\overline{x_6} \vee \overline{x_7})$ & $(x_2 \vee x_4 \vee x_5)$ & $(\overline{x_2} \vee x_3 \vee x_4)$ & $(\overline{x_2} \vee \overline{x_4})$ & $(\overline{x_3} \vee \overline{x_5})$ & $(x_4 \vee x_5)$. The algorithm proceeds as follows (we assume that it picks the variables in the order of their numbers):

Algorithm 2 DPLL procedure

```
Input: CNF formula $F$.
Output: Satisfying assignment or unsatisfiable.
  $(F, t) :=$ unitprop($F$)
  if $F$ is empty then
    return $t$
  else
    Let $x$ be any variable from $F$
    if DPLL($F\&x$) does not return unsatisfiable then
      Add assignments returned by the DPLL call into $t$.
    else if DPLL($F\&\overline{x}$) does not return unsatisfiable then
      Add assignments returned by the DPLL call into $t$.
    else
      return unsatisfiable
    end if
  end if
```

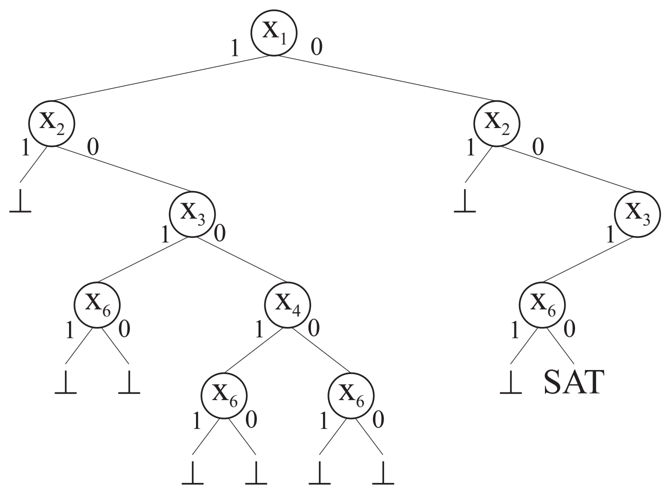


Figure 2.1: Search tree of DPLL on formula F

1. It picks the first variable x_1 and try to assign 1. After the unit resolution it gets a formula with x_1 missed out, because x_1 is presented just negatively. We get $(x_6 \vee x_7)$ & $(x_6 \vee \overline{x_7})$ & $(\overline{x_6} \vee x_7)$ & $(\overline{x_6} \vee \overline{x_7})$ & $(x_2 \vee x_4 \vee x_5)$ & $(\overline{x_2} \vee x_3 \vee x_4)$ & $(\overline{x_2} \vee \overline{x_4})$ & $(\overline{x_3} \vee \overline{x_5})$ & $(x_4 \vee x_5)$.
2. The formula still is not satisfied, so we pick x_2 and try to assign 1. The formula after the assignment looks like this: $(x_6 \vee x_7)$ & $(x_6 \vee \overline{x_7})$ & $(\overline{x_6} \vee x_7)$ & $(\overline{x_6} \vee \overline{x_7})$ & $(\mathbf{1} \vee x_4 \vee x_5)$ & $(\mathbf{0} \vee x_3 \vee x_4)$ & $(\mathbf{0} \vee \overline{x_4})$ & $(\overline{x_3} \vee \overline{x_5})$ & $(x_4 \vee x_5)$. One clause contains 1 and therefore can be missed out. Also a new unit clause $\overline{x_4}$ appeared. It forces value 0 for variable x_4. If we assign this value, we get $(x_6 \vee x_7)$ & $(x_6 \vee \overline{x_7})$ & $(\overline{x_6} \vee x_7)$ & $(\overline{x_6} \vee \overline{x_7})$ & $(x_3 \vee \mathbf{0})$ & $(\mathbf{1})$ & $(\overline{x_3} \vee \overline{x_5})$ & $(\mathbf{0} \vee x_5)$. We get yet another two unit classes which forces x_3 and x_5 to be 1. But after assignment we got the clause $(\overline{x_3} \vee \overline{x_5})$ empty and not possible to satisfy. Therefore the algorithm has to go back to the last point where it did a decision to assign value to x_2, i.e. to the formula from point 1.

3. After returning back to the decision point it tries to assign the other value 0 to the variable x_2. It leads to the formula $(x_6 \vee x_7)$ & $(x_6 \vee \overline{x_7})$ & $(\overline{x_6} \vee x_7)$ & $(\overline{x_6} \vee \overline{x_7})$ & $(\mathbf{0} \vee x_4 \vee x_5)$ & $(\mathbf{1} \vee x_3 \vee x_4)$ & $(\mathbf{1} \vee \overline{x_4})$ & $(\overline{x_3} \vee \overline{x_5})$ & $(x_4 \vee x_5)$. This time we get no unit clause and we have to do another decision.
4. After assigning 1 to x_3 we get $(x_6 \vee x_7)$ & $(x_6 \vee \overline{x_7})$ & $(\overline{x_6} \vee x_7)$ & $(\overline{x_6} \vee \overline{x_7})$ & $(x_4 \vee x_5)$ & $(\mathbf{0} \vee \overline{x_5})$ & $(x_4 \vee x_5)$ which contains a unit clause. It forces to assign 0 to x_5. This leads to $(x_6 \vee x_7)$ & $(x_6 \vee \overline{x_7})$ & $(\overline{x_6} \vee x_7)$ & $(\overline{x_6} \vee \overline{x_7})$ & $(x_4 \vee \mathbf{0})$ & $(\mathbf{1})$ & $(x_4 \vee \mathbf{0})$. And we have to assign 1 to x_4.
5. In the next step algorithm tries to assign value to the x_6 which leads immediately to a formula with empty clause. Let us note, that formula itself $(x_6 \vee x_7)$ & $(x_6 \vee \overline{x_7})$ & $(\overline{x_6} \vee x_7)$ & $(\overline{x_6} \vee \overline{x_7})$ is unsatisfiable.
6. Algorithm returns back to the point where it assigned a value to x_3 and continue with the other value. Let us just skip the computation up to the point where it returns to the root assigns 0 to x_1. At this point the algorithm gets a formula $(x_6 \vee \overline{x_7})$ & $(\overline{x_6} \vee x_7)$ & $(\overline{x_6} \vee \overline{x_7})$ & $(x_2 \vee x_4 \vee x_5)$ & $(\overline{x_2} \vee x_3 \vee x_4)$ & $(\overline{x_2} \vee \overline{x_4})$ & $(\overline{x_3} \vee \overline{x_5})$ & $(x_4 \vee x_5)$.
7. Then it picks variable x_2, try 1 and get $(x_6 \vee \overline{x_7})$ & $(\overline{x_6} \vee x_7)$ & $(\overline{x_6} \vee \overline{x_7})$ & $(\mathbf{1} \vee x_4 \vee x_5)$ & $(\mathbf{0} \vee x_3 \vee x_4)$ & $(\mathbf{0} \vee \overline{x_4})$ & $(\overline{x_3} \vee \overline{x_5})$ & $(x_4 \vee x_5)$. And that actually the interesting point, why we did examine the run of procedure so in-depth. One can notice that formula over the variables x_3, x_4 and x_5 is actually the same as in step 2 and there it was the part that leads to the contradiction (variables x_6 and x_7 were not even assigned there). The question is whether it would be possible to discover the contradiction faster, without assigning value to x_2 this time. We will discus this question bellow, but let us just briefly say, how the algorithm finishes.

... The algorithm returns back to x_2, try the valuation 1 of x_3 with variables of x_4 and x_5 forced. And at the end it will check both possibilities for x_6 where the first one, assigning 1, leads to contradiction and the second one to satisfying all the clauses. It will search no more and return the satisfying assignment: $x_1 = 0$, $x_2 = 0$, $x_3 = 1$, $x_4 = 1$, $x_5 = 0$, $x_6 = 0$, $x_7 = 0$.

As we mentioned in step 6 of the example, there might be occasions when some part of the formula is already unsatisfiable independently on some choices made before (x_1 in the example). It yielded to the emergence of so called Conflict-Driven Clause Learning SAT solvers (known as CDCL solvers, see e.g. Chapter 4 of [5]).

The idea of CDCL solvers is to analyse why the conflict occurred and than add a clause that allows us to derive it faster using unit propagation next time. In the mentioned example such a clause would be simply $\overline{x_2}$ which would prevent the solver from assigning x_2 other value than 0, as 1 inevitable leads to contradiction. In order to describe a way how to get such a clause we have to define an implication graph, as the authors of CDCL solver did [28].

Definition 15. An oriented *implication graph* corresponding to a state of algorithm consists of set of vertices V which correspond to the variable valuations performed on the path from the root and with one special vertex corresponding to contradiction. There is an edge from vertex u to vertex v, if v corresponds to an assignment that was forced as a consequence of assignment u (i.e. v assigns value to a variable which appeared as a unit clause after applying the valuation u).

Moreover for each vertex we remember whether it was chosen as one of the options in algorithm or whether it was derived as a consequence of other assignment. We also mark the number of variables chosen by the algorithm before (i.e. the number of decisions). For every edge we make a note which clause forced value of the variable (we note the corresponding clause from original formula).

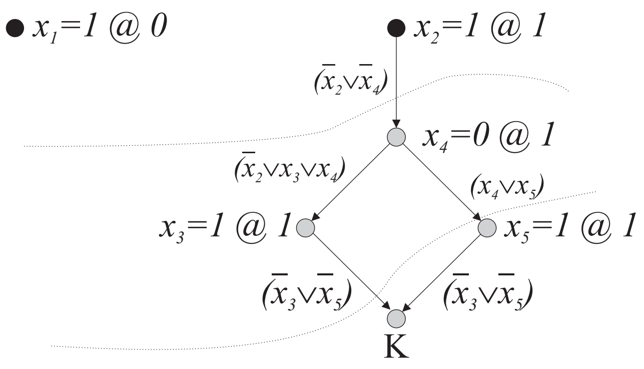

x_1=1 @ 0
x_2=1 @ 1
$(\overline{x_2} \vee \overline{x_4})$
x_4=0 @ 1
$(\overline{x_2} \vee x_3 \vee x_4)$
$(x_4 \vee x_5)$
x_3=1 @ 1
x_5=1 @ 1
$(\overline{x_3} \vee \overline{x_5})$
$(\overline{x_3} \vee \overline{x_5})$
K

Figure 2.2: Implication graph for example 14, step 2

The implication graph for the Example 14 in the step 2 is in the Figure 2.2. In order to construct a clause which speeds up the discovery of the contradiction, we take an edge cut separating vertex K from all decision points (marked in black). The literals corresponding to the vertices at the beginnings of the edge cut's edges form sufficient condition for contradiction - if they are satisfied by a partial assignment, this assignment cannot be extended into one that satisfies the formula (e.g. in the example the conditions corresponding to marked cuts are x_2 and $x_3 \& \overline{x_4}$). But we want to avoid the contradiction and add a further clause which would speed up the contradiction discovery, therefore we take negation of this condition and according to de Morgan laws we get clauses $\overline{x_2}$ and $(\overline{x_3} \vee x_4)$. Then whenever we assign all but one variables in this added clause, unit resolution forces the opposite value other than one which leads to the contradictory partial assignment.

Let us verify that these clauses are implicates of given formula F and therefore can be added to the formula without changing the represented function. Let us start with clause $(\overline{x_3} \vee x_4)$. It can be obtained as a resolution of $(\overline{x_3} \vee \overline{x_5})$ and $(x_4 \vee x_5)$ of the original formula F and therefore $(\overline{x_3} \vee x_4)$ is an implicate of F. The proof that $\overline{x_2}$ is an implicate is more complex. Let us consider clauses $(\overline{x_2} \vee \overline{x_4})$, $(\overline{x_2} \vee x_3 \vee x_4)$, $(\overline{x_3} \vee \overline{x_5})$, and $(x_4 \vee x_5)$ of F. By resolution of first two we obtain clause $(\overline{x_2} \vee x_3)$, and by resolution of second two we get $(\overline{x_3} \vee x_4)$. Resolution of these two clause gives us clause $(\overline{x_2} \vee x_4)$, which is resolvable with first considered clause $(\overline{x_2} \vee \overline{x_4})$ leading to desired clause $\overline{x_2}$. Thus $\overline{x_2}$ is an implicate of F. Note that in both cases the clauses adjacent to edges of implication graph was used.

In a similar manner, for every clause C obtained from an edge cut separating vertex K from vertices of decision, it can be proven that C is an implicate of the original formula. On the other hand, it is pointless to add all clauses we can get in this way. The ones we want to add are those, which would immediately after backtracking to the decision point lead to a derivation of a contradiction by unit resolution. Such a clause contains just one literal from that decision level and is known as an *assertion clause*. Such clauses can be found using resolutions. For more information on this topic refer Chapter 4 of [5].

We referred just the basics of current SAT solvers. Besides these techniques they also incorporate many heuristics on how to pick variables in the most efficient order, which value try first, after how long the algorithm should restart, which of the added clauses it should preserve, and so on. There are also many clever data structures to represent formulae.

Other approach then described are the SAT solvers which do not guarantee completeness (i.e. do not have to terminate), but are usually faster in discovering satisfying assignment. For example SAT solvers working on the basis of random walks. Many information about SAT solvers in general can be found in book [5].

However, no general SAT solvers do guarantee polynomial upper bound on the computation time because SAT is an NP-complete problem and there is no know polynomial algorithm for NP-complete problems. That is why there is also an approach to identify subclasses of formulae such that the satisfiability of every CNF from such a subclass can be tested in polynomial time. This is sufficient in a case we are solving specific problem that we know will generate formulae with a specific properties. In the rest of the thesis we will focus on such classes.

2.2 Unit refutation completeness

In the previous section we have seen that it is sometimes useful to add further information to a formula in order to enable fast detections of contradictions. CDCL solvers add such clauses at the moment the contradiction is discovered. However we can define a class of formulae so that we require that the formulae already contain all clauses that after applying any partial assignment we can detect the conflict by a unit resolution (if the formula obtained by partial assignment is unsatisfiable). These formulae form a class of *unit refutation complete* formulae. It was first described in [16] where it was presented as a more efficient alternative for representing knowledge than canonical formula.

Definition 16. A formula is called *unit refutable* if it is not satisfiable and there is a unit resolution proof of a contradiction. Formula F is *unit refutation complete (UC)* if for its every implicate C the formula $F \ \& \ \neg C$ is unit refutable, i.e. $F \ \& \ \neg C \vdash_1 \bot$.

Let us note that $\neg C$ is for a clause $C = (l_1 \vee \ldots \vee l_k)$ according to the de Morgan's laws equivalent to $\overline{l_1} \ \& \ldots \& \ \overline{l_k}$ which is actually forcing the partial assignments to the variables presented.

Other thing to notice is that for C empty the condition is: if $\bot$ is an implicate of F, than $F \vdash_1 \bot$, i.e. if F is unsatisfiable, it can be detected by unit propagation only. And that is equivalent to the requirement that any unsatisfiable formula from the class of unit refutation complete formulae can be recognized by a unit resolution. It may seem as rather strong condition on a class of formulae, and in some sense it actually is. In Chapter 4 we will cite a recent result of [21, 22] that the class of unit refutation complete formulae is the same as the class of SLUR formulae introduced in Chapter 3, where is also cited other result that a randomly generated formula has a diminishing probability to be in this class.

On the other hand, these classes are interesting from the practical point of view of SAT solvers, because they contain formulae which are easy to solve. In that sense they present a good candidates to encode some problem into.

For this reason they are of interest for a research area called knowledge compilation. The goal of this field is to create a representation of some set of knowledge (a database) which would afterwards allow to pose questions and answer them efficiently. In our case the database is a set of clauses and a query is a clause for which we want to test whether it is a logical consequence of the database. The requirements usually laid on the database are that it is compact to represent and allows us to resolve queries quickly. We usually require that the database which represents knowledge can be compiled quickly, but it is not as critical as the time requirement for answering queries since it is done only once in a preprocessing phase. Knowledge compilation is, though, much more general term, other representation than CNF formulas can be used; for a survey see [13].

The property required by the definition of unit refutation complete formulae is easily valid for the canonical formulae. If C is an implicate of canonical formula F, than it is subsumed by some prime implicate C' (clause of F), therefore $\neg C$ together with C suffices for derivation of contradiction.

We can see the class of unit refutation complete formulae as a trade of between the size of the representation of a given function and the complexity of computation we have to perform in order to solve a query using this function. We are not, thought, guaranteed to obtain polynomial sized representation every time (but we usually obtain smaller representations than the canonical formula). It is very unlikely to discover such a class of formulae which would guarantee small representation for every function and allow efficient procedures for generating the knowledge base and answering queries at the same time.

Various ways how to obtain a unit refutation complete formula from any formula on input were presented in [16]. We will discuss this class later in the connection to the SLUR class, as it was proven in [21, 22] that these classes are actually equivalent.

2.3 Propagation completeness

In comparison to canonical formulae the class of unit refutation complete formulae provides potentially smaller representation of a formula for which it is easy to discover contradiction after a partial assignment. The class of propagation complete formulae goes yet further. A stronger condition is required: if a partial assignment forces a value of some variable, then its value must be deducible by unit propagation.

Definition 17. A formula F is called *propagation complete (PC)* if for any set of literals $l_1, \ldots, l_k$ and any literal d the following condition holds: if d is satisfied in every assignment that satisfies $F \mathbin{\&} l_1 \mathbin{\&} \ldots \mathbin{\&} l_k$ than $F \mathbin{\&} l_1 \mathbin{\&} \ldots \mathbin{\&} l_k \vdash_1 d$.

This definition originates from [6]. There was also proven that we can obtain a PC formula from any formula by adding so called empowering implicates. Those are clauses that add power to derive a literal by unit resolution under some partial assignment that we were not able to derive earlier.

Definition 18. We say that clause $C = l_1 \vee \ldots \vee l_k$ is *empowering* for a formula F if for some of its literals l_i it holds:

1. $F \;\&\; \bigwedge_{j\in\{1,\ldots,k\}\setminus\{i\}} \neg l_j \not\vdash_1 \bot$,
2. literal l_i is satisfied in every assignment satisfying $F \;\&\; \bigwedge_{j\in\{1,\ldots,k\}\setminus\{i\}} \neg l_j$, but
3. $F \;\&\; \bigwedge_{j\in\{1,\ldots,k\}\setminus\{i\}} \neg l_j \not\vdash_1 l_i$.

It is easy to see from the definitions that every propagation complete formula is also refutation complete. Indeed, let $C = (l_1 \vee \ldots \vee l_k)$ be an implicate of F, then $F \& \bigwedge_{i=1}^{k-1} \overline{l_i} \vdash_1 \bot$ or $F \& \bigwedge_{i=1}^{k-1} \overline{l_i} \vdash_1 \bot$. It implies $F \& \bigwedge_{i=1}^{k} \overline{l_i} \vdash_1 \bot$.

The other inclusion between these two classes does not hold. Let us consider e.g. formula $F = (a \vee x_1 \vee x_2 \vee x_3) \;\&\; (\overline{a} \vee x_1 \vee x_2 \vee x_4)$. This formula is prime and has yet another prime implicate $C = (x_1 \vee x_2 \vee x_3 \vee x_4)$, as can be seen by resolution method. It is unit refutation complete, because if we add $\neg C = \overline{x_1} \;\&\; \overline{x_2} \;\&\; \overline{x_3} \;\&\; \overline{x_4}$ to F to make the unsatisfiable formula, we obtain $a \;\&\; \overline{a}$ and consequently the contradiction $\bot$ by series unit resolutions. On the other hand, F is not propagation complete because if we set x_2, x_3 and x_4 to false (by adding $\overline{x_2} \;\&\; \overline{x_3} \;\&\; \overline{x_4}$ to F), we obtain formula $(a \;\vee\; x_1) \;\&\; (\overline{a} \;\vee\; x_1))$ by unit propagation. Hence we get neither a unit clause, nor the empty clause. In fact, C is an empowering implicate of F with empowering literal x_1 (and x_2).

3. Class of SLUR formulae

A class of SLUR formulae was defined in [31, 18] by a non-deterministic algorithm. Its name SLUR is an acronym for 'Single Lookahead Unit Resolution' which states how the algorithm operates. If it is at a point where unit resolution cannot derive anything new, it chooses a variable, tries both assignments, verifies if there cannot be detected a contradiction by a unit resolution and finally chooses some branch where it does not. The class is then defined as such formulae on which this approach never fails, see a more formal definition bellow.

Algorithm 3 SLUR

Input: A CNF formula F.

Output: An assignment satisfying F, *unsatisfiable* or *give up*.

```
1:  $F$, $v$ := unitprop($F$)
2:  if $F$ contains an empty clause then
3:      return unsatisfiable
4:  end if
5:  while $F$ is not an empty set of clauses do
6:      $x$ := arbitrary variable from $F$
7:      $(F_1, v_1)$ := unitprop($F \& \overline{x}$)
8:      $(F_2, v_2)$ := unitprop($F \& x$)
9:      if both $F_1$ and $F_2$ contain an empty clause then
10:         return give up
11:     else if $F_1$ does not contain an empty clause then
12:         $F := F_2$
13:         $v := v \cup v_2$
14:     else if $F_2$ does not contain an empty clause then
15:         $F := F_1$
16:         $v := v \cup v_1$
17:     else
18:         Choose arbitrarily:
19:         1. $F := F_1$, $v := v \cup v_1$ or
20:         2. $F := F_2$, $v := v \cup v_2$
21:     end if
22: end while
23: return v
```

Definition 19. [31, 18] A formula F belongs to *SLUR* class if Algorithm 3 does not *give up* after any series of non-deterministic choices made on lines 6 and 18.

It is known, that SLUR contains many of other known classes of formulae such as Horn, hidden Horn, (hidden) extended Horn and CC-balanced formulae [31]. For overview of relationships of some well studied classes see e.g. [37] or [36].

There is also an obvious resemblance of SLUR algorithm to the DPLL procedure. Except of the non-determinism, the main difference is that DPLL procedure backtracks more than one level, whereas the SLUR algorithm just tests whether a given variable assignment leads to a contradiction by unit propagation (which is in fact equivalent to one layer backtracking).

In some sense we can see, though, SLUR formulae as a class of formulae for which the DPLL (resp. CDCL) procedure works fast. According to [18] the SLUR algorithm can be implemented in to run linear time.

3.1 Properties of SLUR formulae

Let us mention some of the basic properties of SLUR class proven in [37].

Lemma 20. *[37] The class of SLUR formulae is closed under complementing all occurrences of a variable and under partial assignments.*

Proof. The proofs are straightforward. The case of variable complementation is clear because the SLUR algorithm does not make any difference between positive and negative literal. Hence, if we take a formula from SLUR class and complement a variable SLUR algorithm would not *give up* because it does not on the original formula. We can conclude that formula remains in the class and therefore the class is closed under variable complementation.

We prove the fact that the class is closed under partial assignment for the case of one variable assignment, the rest can be done by induction. We consider two cases according to whether the value was assigned to the variable on line 1 or during performing the while cycle. In the first case the value of the variable is forced by a unit clause. If we assign the value that is forced anyway, nothing will change. If we assign the other, the formula became unsatisfiable and the contradiction is detected before entering the while cycle, hence the formula remains in the class. In the second case, when the value is assigned during the while cycle, we can assume that the value is assigned in the first run as the SLUR procedure must succeed for all orders. The partial assignment either shortens the computation by one step or it makes the formula unsatisfiable, which we detect immediately by unit resolution as follows from the assumption that the formula was SLUR (i.e. the wrong branch has been recognized by *unitprop*). Therefore we see that the formula after partial assignment remains in the SLUR class. □

Lemma 21. *[37] The class of SLUR formulae is not closed under deletion of a literal or a clause nor on complementation of a literal.*

Proof. The cases of literal and clause deletion are easy to prove. It is sufficient to consider situation when the clause or literal is necessary for forcing something else. For example formula $F = a \,\&(a \vee x \vee y)\&(x \vee \overline{y})\&(\overline{x} \vee y)\&(\overline{x} \vee \overline{y})$ is SLUR, but if we delete the first clause a or the first literal of the second clause, it will clearly fall out the SLUR class. That is because in both cases we are left with an unsatisfiable formula (for variable order a, x and y the algorithm *gives up*) which is not detected before entering the while cycle. The same obviously holds if we complement one of 'a' literals. Then it will be missed out from the formula by *unitprop* before entering the cycle and we are in the same situation again. □

In [18] it was proven that the probability that a random formula is from the SLUR class is quite low. To precisely cite this result we need to define a probabilistic space of formulae $\mathcal{M}^k_{M,n}$.

Definition 22. A random formula in $\mathcal{M}^k_{M,n}$ is a multi-set of m clauses selected uniformly, independently and with replacement where each clause contains exactly k variable distinct literals over a set of n variables.

The exact statement of [18] is formulated in the following theorem.

Theorem 23. *[18] The probability that a random formula from $\mathcal{M}^k_{M,n}$, with $\frac{m}{n} = r > \frac{2}{k(k-1)}$ and $k \geq 3$ is in the SLUR class tends to 0 for n tends to infinity.*

3.2 coNP-completenes of recognition problem

In this section we will focus on a problem of testing whether a formula is in the SLUR class. We will prove that this problem is coNP-complete. Let us present a formal definition of this problem.

SLUR membership (SLUR-m)	
Instance :	A formula F in CNF.
Question :	Does F belong to the SLUR class?

The following result is quite obvious, but forms a necessary part of SLUR-m coNP-completeness proof. It was observed in [37].

Lemma 24. *[37] Problem* SLUR-m *belongs to the coNP class.*

Proof. We will use the certificate definition of coNP membership, i.e. we will present a certificate verifiable in polynomial time which for a non SLUR formula proves that it really is not SLUR. Such a certificate is formed by the series of non-deterministic choices which, if followed by the SLUR algorithm, leads to the *give up* result. This certificate clearly has a linear size in the number of variables and can be verified by performing the SLUR algorithm under given choices by verifying that it *gives up.* ☐

We yet have to prove the other element necessary for coNP-completeness, the fact that SLUR-m is coNP-hard problem [11]. It is done by a reduction of 3D matching problem, which is known to be NP-complete [19, 24]. 3D matching problem is defined as follows.

3D matching (3DM)	
Instance :	Sets X, Y, Z of the same cardinality $\|X\| = \|Y\| = \|Z\| = q$ and a set of triples $W \subseteq X \times Y \times Z$.
Question :	Does W contain a matching of size q? I.e. is there any $M \subseteq W$ such that $\|M\| = q$ and any two different triples $E, E' \in M$ are disjoint ($E \cap E' = \emptyset$)?

The transformation will be constructed such that for any instance of 3DM we create a formula F_W such that it holds that F_W is in the SLUR class if and only if for the original problem was not possible to construct any 3D matching. The idea is to construct a formula such that if the original problem admits a matching, then the formula will be unsatisfiable but it will not be possible to detect a conflict at the beginning by unit propagation. On the other hand, if it there is no matching, then the algorithm does not *give up* for any order. The actual construction is presented bellow.

Construction 25. [11] Let us have an instance of 3DM problem: let $X = \{x_1, \dots, x_q\}$, $Y = \{y_1, \dots, y_q\}$ and $Z = \{z_1, \dots, z_q\}$ be the sets of the same cardinality q and let $W = \{E_1, \dots, E_w\}$ be a set of w triples we are choosing the matching from. We shall assume that $E_j = (x_{f(j)}, y_{g(j)}, z_{h(j)})$, where f, g, and h are functions determining which elements of X, Y, and Z belong to E_j.

· For every $i \in \{1, \dots, q\}$ let us denote $A_i = (a_i \vee \overline{a_{i+1}})$, where $a_1, \dots, a_{q+1}$ are new variables.
· For every $i \in \{1, \dots, q\}$ and $j \in \{1, \dots, w\}$ let us denote $B_i^j = (\overline{b_i^1} \vee \dots \vee \overline{b_i^{j-1}} \vee b_i^j \vee \overline{b_i^{j+1}} \vee \dots \vee \overline{b_i^w})$, i.e. B_i^j denotes a clause on variables $b_i^1, \dots, b_i^w$, in which every literal is negative except b_i^j.
· For every $i \in \{1, \dots, q\}$ and $j \in \{1, \dots, w\}$ let us denote $C_i^j = (\overline{c_i^1} \vee \dots \vee \overline{c_i^{j-1}} \vee c_i^j \vee \overline{c_i^{j+1}} \vee \dots \vee \overline{c_i^w})$, i.e. C_i^j denotes a clause on variables $c_i^1, \dots, c_i^w$, in which every literal is negative except c_i^j.
· Given a triple $E_j \in W$, let $D_j = (A_{f(j)} \vee B_{g(j)}^j \vee C_{h(j)}^j)$.
· Finally let $F_W = \bigwedge_{j=1}^w D_j \wedge (a_{q+1} \vee a_1) \wedge (\overline{a_1} \vee u) \wedge (\overline{u} \vee \overline{a_1})$.

Let us denote the sets of variables in F_W as follows:

· $\mathcal{A} = \{a_1, \dots, a_{q+1}, u\}$,
· $\mathcal{B}_i = \{b_i^1, \dots, b_i^w\}$ for every $i \in \{1, \dots, q\}$,
· $\mathcal{B} = \bigcup_{i=1}^q \mathcal{B}_i$,
· $\mathcal{C}_i = \{c_i^1, \dots, c_i^w\}$ for every $i \in \{1, \dots, q\}$,
· $\mathcal{C} = \bigcup_{i=1}^q \mathcal{C}_i$ and
· $\mathcal{V} = \mathcal{A} \cup \mathcal{B} \cup \mathcal{C}$.

Firstly we will show some properties of the formula, which we use afterwards to show the relationship of SLUR membership and the matching solution. We say that a clause is *trivial* if it is empty or it evaluates to 1, otherwise we say it is *nontrivial.* Let $F[v]$ denote a formula we get from a formula F by applying a partial assignment v.

Lemma 26. *[11] Let $v : \mathcal{V} \to \{0, 1, *\}$ be a partial assignment and let $j, k \in \{1, \dots, w\}$ be arbitrary. Let us assume that $D_j[v]$ and $D_k[v]$ are nontrivial. Then $D_j[v]$ and $D_k[v]$ are resolvable if and only if $E_j \cap E_k = \emptyset$, particularly D_j and D_k are resolvable over a variable in $\mathcal{A}$, and moreover*

· *$j = k + 1$ and $v(a_{f(j)}) = *$ or*
· *$j = k - 1$ and $v(a_{f(k)}) = *$.*

Proof. Let us at first assume that $D_j[v]$ and $D_k[v]$ are resolvable, which means that they have a conflict in exactly one variable. It immediately follows that $j \neq k$. Let us at first show, that E_j and E_k must be disjoint. For contradiction let us at first assume, that they share an element of Z, i.e. that $h(j) = h(k)$. In this case also $C^j_{h(j)}$ and $C^k_{h(k)}$ are on the same set of variables $\mathcal{C}_{h(j)} = \mathcal{C}_{h(k)}$. But according to definition of $C^j_{h(j)}$ and $C^k_{h(k)}$ we have that they have conflict in two variables $c^j_{h(j)}$ and $c^k_{h(j)}$. If t assigns value to neither of them, then D_j and D_k would not be resolvable, if t assigns value to one of them then no matter which value it is, we have that one of $D_j[v]$ and $D_k[v]$ would be equal to constant 1. Hence it must be the case that $h(j) \neq h(k)$. Similarly we can show that $g(j) \neq g(k)$ we only consider set Y instead of Z. It also follows that if $D_j[v]$ has one conflict variable with $D_k[v]$, then it must be either $a_{f(j)}$, or $a_{f(k)}$. The proposition follows from definition of $A_{f(j)}$ and $A_{f(k)}$.

Now let us assume, that $E_j \cap E_k = \emptyset$, $f(j) = f(k) + 1$, and $v(a_{f(j)}) = *$ (the case when $f(j) = f(k) - 1$ is similar and is left to the reader). In this case there is only one conflict variable of D_j and D_k and it is $a_{f(j)}$, which is left intact by v and hence it is also a conflict variable of $D_j[v]$ and $D_k[v]$, which means that $D_j[v]$ and $D_k[v]$ are resolvable. □

Lemma 27. *[11] Let $v : \mathcal{V} \to \{0, 1, *\}$ be any partial assignment of variables in F_W and let $j \in \{1, \ldots, w\}$. If $D_j[v]$ is not equal to 1 and it contains exactly one literal with variable in $\mathcal{B}_{g(j)}$ ($\mathcal{C}_{h(j)}$ respectively), then no other nontrivial clause in $F_W(t)$ can contain a variable in $\mathcal{B}_{g(j)}$ ($\mathcal{C}_{h(j)}$ respectively).*

Proof. The proof is similar to the proof of Lemma 26. Let us proceed by contradiction and let us assume that there is a clause D_k, $k \neq j$, such that $D_k(t)$ is a nontrivial clause containing a variable from $\mathcal{B}_{g(j)}$, which means that $g(k) = g(j)$. Let us denote the only variable in $\mathcal{B}_{g(j)}$ which appears in $D_j[v]$ by $b^r_{g(j)}$ for some $r \in \{1, \ldots, w\}$. Since v sets all variables from $\mathcal{B}_{g(j)}$ except $b^r_{g(j)}$, otherwise $D_j[v]$ would contain all variables from $\mathcal{B}_{g(j)}$ unset by v, we have that $b^r_{g(j)}$ is also the only variable from $\mathcal{B}_{g(j)}$ which appears in $D_k[v]$. If $r \neq k$, then literal $b^k_{g(j)}$ is set to 1 by v in one of D_j or D_k meaning that one of these clauses would be trivial, because D_j and D_k have a conflict in $b^k_{g(j)}$. If $r = k$, then the same can be told about $b^j_{g(j)}$, because also in this variable D_j and D_k have a conflict. □

From the Lemma 27 we can conclude an easy corollary.

Corollary 28. *[11] Let $v : \mathcal{V} \to \{0, 1, *\}$ be any partial assignment of variables in F_W such that $F_W(t)$ does not contain the empty clause and let $x \in \mathcal{B} \cup \mathcal{C}$ be a variable, which is not set by v (i.e. $v(x) = *$). Then one of $F_W[v, x = 0]$ and $F_W[v, x = 1]$ does not contain the empty clause.*

Proof. If the empty clause occurs in $F_W[t, x = 0]$, then it is because literal u is present in $F_W[v]$, according to Lemma 27 we have that this literal u is the only occurrence of variable u in $F_W[v]$, hence setting u to 1 cannot produce the empty clause. □

In the rest of this section we will say that *SLUR algorithm gives up on a formula F with variable x and assignment v* if x was the last variable chosen at line 6 before the algorithm *gives up* and v was the variable assignment at the moment of that choice.

The following lemma shows we show that only variables from $\mathcal{A}$ can cause that SLUR algorithm *gives up.*

Lemma 29. *[11] Let us assume that SLUR algorithm* gives up *on F_W with variable x and assignment $v : \mathcal{V} \to \{0, 1, *\}$, then $x \in \mathcal{A}$.*

Proof. If SLUR algorithm *gives up* on F_W with variable x and assignment v, then unit propagation on both $F_W[v, x = 0]$ and $F_W[v, x = 1]$ generate the empty clause. Let us assume for contradiction that $v \in \mathcal{B} \cup \mathcal{C}$.

If the empty clause is directly present in $F_W[v, x = 0]$ or $F_W[v, x = 1]$, it would mean that we have a unit clause in $F_W[v]$, which is not possible, because performing unit propagation each time after assigning a value to a variable ensures, that in SLUR algorithm at the beginning of *while* cycle, formula F contains neither a unit clause nor the empty clause.

It follows that both $F_W[v, x = 0]$ and $F_W[v, x = 1]$ contain new unit clause which causes unit propagation to generate the empty clause, let us assume, that $d = b_i^j$ for some $i \in \{1, \ldots, q\}$ and $j \in \{1, \ldots, w\}$ (the case of variable from $\mathcal{C}$ is the same). Let D_k be the clause for which $g(k) = i$ and $D_k[v]$ is a quadratic clause containing literal with b_i^j. If the other variable in $D_k[v]$ belongs to $\mathcal{B} \cup \mathcal{C}$, then no resolution can occur by setting b_i^j to 0 or 1 according to Lemma 26, and therefore unit propagation cannot generate the empty clause. If the other variable belongs to $\mathcal{A}$, then b_i^j occurs only once in $F_W[v]$ according to Lemma 27 and therefore $D_k[v, b_i^j = 0] \equiv 1$ or $D_k[v, b_i^j = 1] \equiv 1$, which means that in one of these cases unit propagation will not do anything and in particular it will not generate the empty clause. □

Lemma 30. *[11] Let us assume that SLUR algorithm* gives up *on F_W with variable $x \in \mathcal{A}$ and assignment $v : \mathcal{V} \to \{0, 1, *\}$, then $A_1, \ldots, A_q, (a_{q+1} \vee a_1), (\overline{a_1} \vee u), (\overline{u} \vee \overline{a_1})$ are all present in $F_W[v]$.*

Proof. The idea of the proof is as follows. According to Lemma 26 if two clauses in $F_W[v']$ for any partial assignment v' are resolvable, then they are resolvable over a variable in $\mathcal{A}$, thus to generate the empty clause in $F_W[v, x = 0]$ we must follow chain of resolutions over variables from $\mathcal{A}$, the only chain of such resolutions in whole formula is formed by clauses among $A_1, \ldots, A_q, (a_{q+1} \vee a_1), (\overline{a_1} \vee u)$. If all these clauses are present in $F_W[v]$, then setting any variable in $\mathcal{A}$ to any value causes unit propagation to generate the empty clause. On the other hand, we shall show, that if this chain is broken, i.e. some of these clauses do not appear directly in $F_W[v]$ (although they may appear as a proper subclause of a clause in $F_W[v]$), then in one of the assignments unit propagation stops before it generates the empty clause. To this end we have to distinguish several cases according to $x \in \mathcal{A}$.

- If $x = a_1$, then because SLUR algorithm have not yet unit propagated a_1, then $(a_{q+1} \vee a_1)$, $(\overline{a_1} \vee u)$ and $(\overline{u} \vee \overline{a_1})$ must all belong to $F_W[v]$. Because unit propagation generates the empty clause in $F_W[v, a_1 = 0]$, we have that a_1 must appear positively in a quadratic clause in $F_W[v]$, note that $F_W[v]$ does not contain any unit clauses, because we have proceeded with unit propagation on it. Let us now show that also $A_1, \ldots, A_q$ all belong to $F_W[v]$. Let us proceed by contradiction and let us assume that some $A_i \notin F_W[v]$. Let $i \in \{1, \ldots, q\}$ be the smallest index for which $A_i \notin F_W[v]$ and let $j \in \{1, \ldots, q\}$ be the maximum index for which $A_j \notin F_W[v]$. If $i = 1$, then the only unit clause in $F_W[v, a_1 = 0]$ is a_{q+1}. Unit propagation will

follow the chain of quadratic clauses $A_q, A_{q-1}, \ldots, A_{j+1}$, where from A_{j+1} we will get a unit clause a_{j+1} (if $j = q$, then we still have $a_{q+1} = a_{j+1}$). Let D_k be an arbitrary clause such that $D_k[v, a_1 = 0]$ is nontrivial and such that it contains $\overline{a_{j+1}}$. If no such clause exists, then unit propagation stops without generating the empty clause. Similarly if $D_k[v, a_1 = 0]$ contains at least three literals, then unit propagation sets a_{j+1} to 1 and still $D_k[v, a_1 = 0]$ will after this assignment contain at least two literals, which means that unit propagation will stop without generating the empty clause. If $D_k[v, a_1 = 0]$ is quadratic then because A_j is not directly present in $F_W[v]$, we have that the other literal in $D_k[v, a_1 = 0]$ must be on a variable from $\mathcal{B} \cup \mathcal{C}$, but according to Lemma 27 this is the only occurrence of this variable in $F_W[v]$ and hence again unit propagation will not generate the empty clause.
Now let us assume that $1 < i \leq j \leq q$. Now we have two chains of quadratic clauses, one is $A_1, \ldots, A_{i-1}$, and the other is $A_q, A_{q-1}, \ldots, A_{j+1}$. But using the same arguments as in the case of only one chain we can argue that in this case unit propagation would not generate the empty clause in any of these chains. This is true even if $i = j$, in which case both literals in A_i are assigned values, in this case any clause $D_k[v]$ which contains A_i must contain at least three literals, otherwise A_i would be present in $F_W[v]$, if it contains at least four literals, then after assigning values to the variables in A_i, we have two literals remaining in $D_k[v]$ and thus unit propagation stops. If only one literals remains in $D_k[v]$, then it is on a variable from $\mathcal{C} \cup \mathcal{B}$ in which case it is the only occurrence of this variable in $F_W[v]$ according to Lemma 27 and thus unit propagation again does not generate the empty clause.

· $x = a_i$ for some $i \in \{2, \ldots, q\}$. Let us again proceed by contradiction a let $j \in \{1, \ldots, i\}$ be maximum index for which A_j is not present in $F_W[v]$ and let $k \in \{i, \ldots, q\}$ be maximum index for which A_k is not present in $F_W[v]$. If we set a_i to 0, then we get a unit clause, otherwise unit propagation cannot generate the empty clause, therefore $A_i = (a_i \vee \overline{a_{i+1}})$ belongs to $F_W[v]$. Unit propagation then follows the chain of clauses $A_i, A_{i+1}, \ldots, A_{k-1}$, in which case we can argue in the same way as in the case of a_1 that it unit propagation stops when it reaches clauses containing A_k as a subclause. Similarly, if we set a_i to 1, then $A_{i-1} = (a_{i-1} \vee \overline{a_i})$ must belong to $F_W[v]$ and then unit propagation follows the chain of clauses $A_{i-1}, A_{i-2}, \ldots, A_{j-1}$ and again we can argue that it stops when it reaches clauses having A_j as a subclause. It follows that $A_1, \ldots, A_q$ all belong to $F_W[v]$. It means that we have not yet unit propagated a_1 and a_{q+1} and thus also $(a_{q+1} \vee a_1)$, $(\overline{a_1} \vee u)$, and $(\overline{u} \vee \overline{a_1})$ must belong to $F_W[v]$.
· $x = a_{q+1}$. If we set a_{q+1} to 1, then because we have to produce a quadratic clause by this assignment to be able to generate the empty clause using unit propagation, we must have that $A_q = (a_q \vee \overline{a_{q+1}})$ belongs to $F_W[v]$. Thus next step in unit propagation is equivalent to assigning $a_q = 1$ and by the same arguments as in the case $x = a_q$ we can show that it means that all $A_1, \ldots, A_q$ belong to $F_W[v]$. Because we have not yet unit propagated a_{q+1} clause $(a_{q+1} \vee a_1)$ must be present in $F_W[v]$ and thus we have not yet unit propagated a_1 as well, which implies that $(\overline{a_1} \vee u)$ and $(\overline{u} \vee \overline{a_1})$ must also be present in $F_W[v]$.

· If $x = u$. No matter which value we assign to u, next step in unit propagation is to assign $a_1 = 0$, as we have already shown, it implies together with the fact that we have not yet unit propagated neither u nor a_1, that all required clauses belong to $F_W[v]$.

□

In the following lemma we finally show, that F_W is not SLUR if and only if W contains a perfect matching.

Lemma 31. *[11] Instance X, Y, Z and W of* 3DM *contains a perfect matching if and only if F_W is not SLUR.*

Proof. First let us assume, that W contains a perfect matching $M \subseteq W$, and let $M = \{E_{j_1}, \dots, E_{j_q}\}$. Let v be a partial assignment, which assigns variables in $B^{j_1}_{g(j_1)}, \dots, B^{j_q}_{g(j_q)}$ and $C^{j_1}_{h(j_1)}, \dots, C_{h(j_q)}$ and no other variables, moreover this assignment sets the variables in such a way that $B^j_{g(j)}[v] = 0$ and $C^j_{h(j)}[v] = 0$ for every $j \in \{j_1, \dots, j_q\}$. Note, that since triples in M are pairwise disjoint, all variables appear at most once in $B^{j_1}_{g(j_1)}, \dots, B^{j_q}_{g(j_q)}$ and $C^{j_1}_{h(j_1)}, \dots, C_{h(j_q)}$, and thus such assignment v exists and is unique. If SLUR chooses variables assigned by v in any order and if it chooses their values according to v, then we can observe, that no unit resolution will ever occur, because no unit clause will be produced by these assignments. Hence the SLUR algorithm will not fail in the process and in each step it will be able to choose the assignment of a variable according to v. When all variables are set according to v, we thus get formula $F_W[v]$, in which from every clause D_j for $j \in \{j_1, \dots, j_q\}$ remained only A_j. Thus we get, that $F_W[v]$ contains the following subformula:

$$\left(\bigwedge_{j \in \{j_1, \dots, j_q\}} A_j \right) \wedge (a_{q+1} \vee a_1)(\overline{a_1} \vee u)(\overline{u} \vee \overline{a_1}) \tag{3.1}$$

Because M is a matching and thus every $x_i \in X$ appears in exactly one triple of M, we get that CNF (3.1) is equivalent to:

$$\left(\bigwedge_{j=1}^{q} A_j \right) \wedge (a_{q+1} \vee a_1)(\overline{a_1} \vee u)(\overline{u} \vee \overline{a_1}) \tag{3.2}$$

It follows, that $F_W[v]$ is actually equivalent to (3.2), because each clause in $F_W[v]$ is absorbed by one of the clauses in (3.2). Using definition of A_j we get that this is equivalent to:

$$(a_1 \vee \overline{a_2})(a_2 \vee \overline{a_3}) \dots (a_q \vee \overline{a_{q+1}}) \wedge (a_{q+1} \vee a_1)(\overline{a_1} \vee u)(\overline{u} \vee \overline{a_1}) \tag{3.3}$$

Observe, that CNF (3.3) is an unsatisfiable quadratic CNF (it reduces to $a_1 \wedge \overline{a_1}$ after resolutions are made) and therefore it is not SLUR. Hence whole formula F_W is not SLUR.

Now let us assume that SLUR fails on F_W. According to Lemma 29 it must fail on a variable from $\mathcal{A}$, let $v : \mathcal{V} \to \{0, 1, *\}$ be a partial assignment produced by SLUR algorithm before it chooses a variable on which it fails. Then according to Lemma 30 we must have that all clauses $A_1, \ldots, A_q \in F_W[v]$, let $D_{j_1}, \ldots, D_{j_q}$ be some clauses such that $D_{j_1}[v] = A_1, \ldots, D_{j_q}[v] = A_q$. Now let $a, b \in \{1, \ldots, q\}$ such that $a \neq b$. Because $A_a \neq A_b$ we have that $D_{j_a} \neq D_{j_b}$, and thus $j_a \neq j_b$. Because $f(j_a) = a$ and $f(j_b) = b$, we also have that $x_{f(j_a)} \neq x_{f(j_b)}$. By the same arguments as in only if part of the proof of Lemma 26 we can show, that in fact $E_{j_a} \cap E_{j_b} = \emptyset$. In particular let us assume that $h(j_a) = h(j_b)$ (the case with $g(j_a) = g(j_b)$ is the same), in this case both $C^{j_a}_{h(j_a)}$ and $C^{j_b}_{h(j_b)}$ are on the same set of variables $\mathcal{C}_{h(j_a)} = \mathcal{C}_{h(j_b)}$ and each of these variables is assigned a value 0 or 1 by v. By definition there is a variable, in which $C^{j_a}_{h(j_a)}$ and $C^{j_b}_{h(j_b)}$ have a conflict in a variable, that implies that one of $C^{j_a}_{h(j_b)}[v]$ and $C^{j_b}_{h(j_b)}[v]$ is evaluated to 0, which is in contradiction with fact that $D_{j_a}[v] = A_a$ and $D_{j_b}[v] = A_b$. ☐

Now we can use previous to show the result of coNP-completeness of SLUR membership testing.

Theorem 32. *Problem* SLUR-M *is coNP-complete.*

Proof. The result is consequence of Lemma 24 and 31. ☐

3.3 Further properties of SLUR formulae

As we have seen in Section 3.2, there is a very small chance to find out an algorithm recognizing SLUR formulae polynomially. For that reason it is natural to ask for some sufficient conditions. In [11] we showed that every formula which contains all the prime implicates, i.e. contains the canonical formula as a subformula, is SLUR.

Later in [4] we strengthened this result so that it claims that it suffices if the prime implicates can be derived by one resolution step from the formula. The key part of the proof is the following lemma, which is extending Lemma 5.

Lemma 33. *[4] Let $F \in CANON(1)$ and let x be any variable of F. Then both $F[x = 0]$ and $F[x = 1]$ are also in $CANON(1)$.*

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

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

Because $F' = F[x = 0]$, 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 t, too. If $v(x) = 0$, then t 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

$$C \leq C' \vee x. \tag{3.4}$$

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

$$C[x = 0] = C'. \tag{3.5}$$

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

Let us now assume that $C \notin F$. From our assumption that $F \in \mathrm{CANON}(1)$ it follows, that there are $C_1, C_2 \in F$ 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 F' including their conflict variable. We can therefore do the same resolution step as before and get $C = C' = R(C_1, C_2)$.
   (b) On the other hand, if the resolution step uses x as a conflict variable, then we can write $C_1 = A \vee x$, $C_2 = B \vee \overline{x}$, and $C = R(C_1, C_2) = A \vee B$ for some clauses A and 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 3.4. Now since $C_1 \in F$, we get that $C_1[x = 0] = A \in F'$ and because C' is a prime implicate of f', it must be the case that in fact $C' = A \in F'$.
2. It remains to consider the case when C contains x, but $C \notin F$. 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 $C_1, C_2 \in F$. Therefore $C_1[x = 0]$, $C_2[x = 0]$ are two resolvable clauses which belong to F' and we have that $C' = R(C_1[x = 0], C_2[x = 0])$.

□

We will also need the following simple observation about unsatisfiable clauses from the class CANON(1).

Lemma 34. *[4] If F is unsatisfiable and $F \in CANON(1)$, then either F contains an empty clause, or an empty clause is generated during unitprop(F).*

Proof. Let f denote the function represented by F. If F is unsatisfiable, then f has only one prime implicate — an empty clause $\emptyset$. Due to the assumption that $F \in \mathrm{CANON}(1)$, we get that either $\emptyset \in F$ or $\emptyset = R(C_1, C_2)$, where $C_1, C_2 \in F$. 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 $C_2 = x$). It means that an empty clause would be generated during unit propagation. □

Now we prove promised result that $\mathrm{CANON}(1) \subseteq SLUR$ and afterwards present an example showing that the inclusion is strict.

Theorem 35. *[4] If F is in CANON(1) then it is also in the SLUR class.*

Proof. If F is unsatisfiable, then by Lemma 34 SLUR algorithm would correctly recognize it after unit propagation in step 2.

Let us assume, that F is satisfiable. Inductive use of Lemma 33 ensures that in every step of the algorithm the formula considered belongs to CANON(1). This is because every formula originates from F by a partial assignment and unit propagation do nothing other then partial assignments. Particularly, $F\&\overline{x}$ corresponds to $F[x = 0]$ and $F\&x$ corresponds to $F[x = 1]$. If at the beginning of the while cycle formula F is satisfiable, then one of F_1 and F_2 is satisfiable and if one of them is unsatisfiable, it contains an empty clause by Lemma 34. Thus, at the beginning of the next cycle F is again satisfiable and at the end the SLUR algorithm finds satisfying assignment. □

Example 36. It is not hard to create a formula which is in the SLUR class, but it is not in CANON(1). It suffices to consider as simple formula as $x\&(\overline{x}\vee y)\&(\overline{y}\vee\overline{x})$. The SLUR algorithm clearly discovers that this formula is unsatisfiable, but it is not CANON(1) because its only prime implicate, the empty clause, lays in depth two.

In [4] we have given also an example of a formula which showed that the result of Theorem 35 cannot be strengthen to class of CANON(2). The formula proving this is as follows.

$$F = (x \vee y \vee a)\&(x \vee \overline{y} \vee b)\&(\overline{x} \vee y \vee c)\&(\overline{x} \vee \overline{y} \vee d) \tag{3.6}$$

All the other implicates which can be derived by resolution from F are the following:

$$(x \vee a \vee b), (y \vee a \vee c), (\overline{y} \vee b \vee d), (\overline{x} \vee c \vee d), (a \vee b \vee c \vee d). \tag{3.7}$$

The last clause has resolution depth two, thus $F \in$ CANON(2). All the remaining clauses have resolution depth one. In order to show that F is not SLUR let us consider the following run of the SLUR algorithm. Let the algorithm first choose variables a, b, c and d and sets them all to 0. It leads to a complete quadratic CNF, which is unsatisfiable and thus the SLUR algorithm must *give up* later in the process. This implies that not all CNF formulae from CANON(2) are SLUR.

CNF F from line (3.6) above has another interesting property. It is the only prime and irredundant CNF representing the given function f. And it is also the only prime representation which is not SLUR. It suffices to add any other implicate from list 3.7 to F to make it SLUR.

If e.g. we add $(x \vee a \vee b)$ to F, then the SLUR algorithm recognizes 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 is assigned a value before a, b, c or d, or if one of a, b, c or d is assigned value 1, then SLUR algorithm gets a satisfiable quadratic formula, which is SLUR. The cases of the next three implicates in 3.7 are symmetric. Adding the implicate $(a \vee b \vee c \vee d)$ makes the formula belong to CANON(1) and therefore SLUR, because it is the only implicate with resolution depth 2.

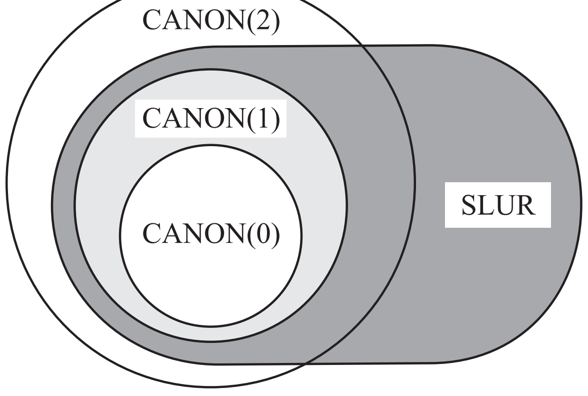


Figure 3.1: Mutual inclusions of CANON(i) hierarchy and the SLUR class

This is also an example of a function, that does not have a prime and irredundant SLUR representation.

Yet further insight into the area of sufficient condition arise as a consequence of the proof of equivalence of SLUR class and class of unit refutation formula discussed in [21, 22]. As it concerns hierarchies described in the following chapter, we will discuss this matter later on.

4. Hierarchies over the SLUR class

In this chapter we will consider some ways of extending the SLUR class into hierarchies. We will present some of their properties and show mutual relations.

4.1 Hierarchy SLUR(i)

In this section we shall show the most straightforward idea how the SLUR class can be generalized into a hierarchy of classes of CNFs. This approach was presented in [37, 11]. For each fixed $i \geq 1$ we define class SLUR(i) by parametrized version of the original SLUR algorithm (Algorithm 3 on page 18) as follows.

Instead of selecting a single variable as on line 6 of Algorithm 3 (the original SLUR algorithm), the parametrized version SLUR(i) (Algorithm 4) non-deterministically selects i variables, and instead of considering the two possible values for the selected variable on lines 7 and 8 in Algorithm 3 it considers all possible 2^i assignments for the selected i-tuple. If all assignments produce the empty clause in the first iteration (after selecting the first i-tuple of variables) algorithm SLUR(i) returns *unsatisfiable.* If all assignments produce the empty clause in any of the subsequent iterations SLUR(i) *gives up.* If at least one of the assignments does not produce the empty clause SLUR(i) non-deterministically chooses one of such assignments and continues in the same manner. The class SLUR(i) is then defined as the class of CNFs F on which SLUR(i) (i.e. Algorithm 4) never *gives up* regardless of the choices made. Note that the SLUR class is a strict subset of SLUR(1) since there is a CNF on which the SLUR algorithm *gives up* after selecting the first variable but the contradiction is discovered by *unitprop* on the first layer. For instance complete quadratic formula $(x \vee y)$ & $(x \vee \overline{y})$ & $(\overline{x} \vee y)$ & $(\overline{x} \vee \overline{y})$ is not in the SLUR class because it does not contain any unit clause and the initial *unitprop* of the SLUR algorithm (step 1 of Algorithm 3) does not discover the contradiction. Whereas, if we assign a value to any of the variables, *unitprop* can derive the empty clause and because both branches on the first layer lead to a contradiction, the formula is SLUR(1).

Definition 37. Formula F is a member of SLUR(i) if Algorithm 4 does not *give up* on F after any series of non-deterministic choices on lines 10 and 26.

It is obvious from the definition, that SLUR(i) provides for every fixed i a polynomial time SAT algorithm with respect to the length of the input CNF F (of course, the time complexity grows exponentially with 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.

Since there are 2^i possible combinations of positive and negative literals over the set of i literals, the for-cycle on line 12 can be implemented in time $O(n \cdot 2^i)$ provided that we implement *unitprop* in linear time [12]. That gives us an upper bound of $O(n^2 \cdot 2^i)$ for the whole algorithm.

In the rest of this section we shall show two more results. First, we prove,

Algorithm 4 SLUR(i)

```
Input: A CNF formula $F$.
Output: An assignment satisfying $F$, unsatisfiable or give up.
 1: $F$, $v$ := unitprop($F$)
 2: if $F$ contains an empty clause then
 3:    return unsatisfiable
 4: end if
 5: $j := 0$
 6: while $F$ is not empty do
 7:    $j := j + 1$
 8:    $Q$ := empty set
 9:    $l_Q := 0$
10:    $A$ := set of arbitrary $i$ variables from $F$ (if it contains less variables, then
       all of them)
11:    $B := \{l_1 \ \& \ \dots \ \& \ l_{|A|} : l_1, \dots, l_{|A|}$ are literals over all variables from set $A\}$
       $\{$Every element of $B$ is a formula of unit clauses over variables of $A\}$
12:    for all $S \in B$ do
13:       $F_S, v_S$ := unitprop($F$ & $S$)
14:       if $F_S$ does not contain an empty clause then
15:          add $(F_S, v_S)$ into $Q$
16:          $l_Q := l_Q + 1$
17:       end if
18:    end for
19:    if $l_Q = 0$ then
20:       if $j = 1$ then
21:          return unsatisfiable
22:       else
23:          return give up
24:       end if
25:    else
26:       $F, v_{\text{chosen}}$ := arbitrary element of $Q$
27:       $v := v \cup v_{\text{chosen}}$
28:    end if
29: end while
30: return v
```

that the SLUR(i) hierarchy does not collapse, and then we extend the coNP-completeness recognition result for SLUR(1) to SLUR(i) for an arbitrary i.

Lemma 38. *[37, 11] For every i there is a formula F_{i+1} such that $F_{i+1} \in$ SLUR($i+1$) $\setminus$ SLUR(i).*

Proof. Let us construct F_{i+1} as an unsatisfiable CNF on $i+2$ variables $V = \{x_1, \dots, x_{i+2}\}$. It contains each possible combination of positive and negative literals and can be written as:

$$F_{i+1} = \bigwedge_{P \subseteq V} \left(\bigvee_{v \in P} v \vee \bigvee_{v \in V \setminus P} \overline{v} \right).$$

If we assign values to any i-tuple of variables some of the clauses disappear and the rest of the formula is the complete unsatisfiable formula over two variables (e.g. if we assign arbitrary zero-one values to $x_1, \ldots, x_i$, we get $(x_{i+1} \vee x_{i+2})$ & $(x_{i+1} \vee \overline{x_{i+2}})$ & $(\overline{x_{i+1}} \vee x_{i+2})$ & $(\overline{x_{i+1}} \vee \overline{x_{i+2}})$). *Unitprop* will not derive the empty clause on such CNF and the algorithm SLUR(i) will have to *give up* in the next step.

On the other hand, if we assign values to any $(i+1)$-tuple we will get a CNF of the x & $\overline{x}$ form. *Unitprop* will derive the empty clause on such a formula and so the algorithm SLUR($i+1$) returns *unsatisfiable.* □

It is also easy to observe that any CNF formula is an element of SLUR(i) for some i. It holds because if we chose i equal to the number of variables in the formula or greater, algorithm obviously never *gives up.*

It is not a big surprise that recognition of SLUR(i) formulae is coNP-complete again. In the following lemma we present a way how to show that using the analogous result for SLUR class.

Lemma 39. *[11] For each i the membership problem for the class SLUR(i) is coNP-complete.*

Proof. Similarly as in the SLUR case, an order of selected variables and the values assigned to them which forces SLUR(i) to *give up* on F, serves as a polynomially verifiable certificate that F is not in SLUR(i). Therefore, the membership problem for SLUR(i) is in coNP.

To prove coNP-hardness we modify the proof used for the SLUR case. We take i copies of formula F_W defined in Section 3.2, each of them on a new set of variables, and add one more clause containing disjunction of another i new variables. That is, given an instance W of 3DM we construct a CNF

$$F = {F_W}^{(1)} \ \& \ {F_W}^{(2)} \ \& \ \ldots \ \& \ {F_W}^{(i)} \ \& \ (n_1 \vee \ldots \vee n_i),$$

where $F_W^{(1)}, \ldots, F_W^{(i)}$ are copies of F_W on pairwise disjoint sets of variables. It now suffices to prove that the input instance of 3DM contains a perfect matching if and only if F does not belong to the class SLUR(i).

Let us first assume that the input instance of 3DM contains a perfect matching. Using Lemma 31 we get that the original SLUR algorithm *gives up* on F_W for some order of variable assignments. Now we construct an order of variable assignments which will force SLUR(i) to *give up* on F. First the algorithm picks the dummy variables $\{n_1, \ldots n_i\}$ and assigns some values that lead to the satisfying assignment of the last clause of F (this step prevents the algorithm to return *unsatisfiable* instead of *giving up* in the case when the original SLUR algorithm *gives up* after assigning just one variable). Now SLUR(i) on formula F will follow the order of variable assignments by using the one which forces the SLUR algorithm to *give up* on F_W. Every time the SLUR algorithm pick a variable and assigns a value to it, SLUR(i) does the same in all i copies of F_W which sit inside of F. This order clearly leads SLUR(i) to *give up* on F, proving that F does not belong to the class SLUR(i).

Now let us assume that F does not belong to the class SLUR(i). That means that SLUR(i) *gives up* on F for some order of variable assignments. Since F consists of $i+1$ sub-CNFs on disjoint sets of variables, it follows that SLUR(i) derives the empty clause by unit propagation from one of these subCNFs for all possible assignments of the last selected i-tuple of variables. Clearly, this cannot happen for the last clause in F, so it must happen for one of the copies of F_W. But now restricting the variable assignment only to the variables from this particular copy of F_W, we get an order of variable assignments which makes the original SLUR algorithm *give up* on F_W. Thus F_W is not SLUR and using Lemma 31 we get that the input instance of 3DM contains a perfect matching. □

4.2 Hierarchy SLUR*(i)

Further step in building up hierarchies is to realise that we can perform unit propagation after each variable assignment [4]. This way we get a stronger hierarchy, as can be seen from the following example.

Example 40. [4] Let us consider the following formula

$$\begin{aligned} F =&(x \vee \overline{y}) \ \& \ (\overline{x} \vee y) \ \& \ (x \vee y \vee a \vee b) \ \& \ (x \vee y \vee \overline{a} \vee b) \\ &(x \vee y \vee a \vee \overline{b}) \ \& \ (x \vee y \vee \overline{a} \vee \overline{b}) \end{aligned} \tag{4.1}$$

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 $x = y = 0$, the SLUR(2) algorithm *gives up* as it is left with a complete 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 were 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 a hierarchy consisting of classes SLUR*(i). The SLUR*(i) algorithm also chooses i variables at each step, but it performs unit propagation after each substitution into every single variable rather then one unit propagation after all of them. Formally, the SLUR*(i) class is defined using Algorithm 6, which uses procedure *test* (Algorithm 5) as a sub-procedure.

Definition 41. [4] Formula F is a member of SLUR*(i) class if Algorithm 6 does not return *give up* for any of non-deterministic choices made during its run.

Note, that all non-determinism is now stored in step 8 of procedure *test* (Algorithm 5). 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 [14]), 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 $i \geq 1$ the inclusion

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

is strict, in fact exactly the same argumentation as for the previous SLUR(i) hierarchy can be used. Modification of the proof to SLUR*(i) hierarchy is contained in the following lemma.

Algorithm 5 test

Input: A pair of a CNF formula F and a number of variables to be assigned k.

Output: A partial assignment of k variables which does not enable us to conclude a contradiction from F by unit propagation or *unsatisfiable*, if there is no such assignment.

```
 1: $F$, $v$ := unitprop($F$)
 2: if $F$ contains an empty clause then
 3:     return unsatisfiable
 4: end if
 5: if $k = 0$ then
 6:     return an empty assignment
 7: end if
 8: $l$ := a literal (positive or negative) over an unassigned variable from $F$
 9: $t_1$ := test($F$ & $l$, $k - 1$)
10: if 'test' on the previous line did not return unsatisfiable then
11:     return $t \cup t_1$
12: end if
13: $t_2$ := test($F$ & $\bar{l}$, $k - 1$)
14: if 'test' on the previous line did not return unsatisfiable then
15:     return $t \cup t_2$
16: end if
17: return unsatisfiable
```

Algorithm 6 SLUR*(i)

Input: A CNF formula without an empty clause.

Output: A partial assignment satisfying F, *unsatisfiable* or *give up.*

```
 1: $F$, $v$ := unitprop($F$)
 2: if $F$ contains an empty clause then
 3:     return unsatisfiable
 4: end if
 5: while $F$ contains a clause do
 6:     $t'$:=test($F$, $i$)
 7:     if previous test returns unsatisfiable then
 8:         if it is the first run of the while cycle then
 9:             return unsatisfiable
10:         else
11:             return give up
12:         end if
13:     end if
14:     return $t$:=$t \cup t'$
15: end while
16: return $t$
```

Lemma 42. *[4] For each i there is a formula F_{i+1} such that $F_{i+1} \in SLUR^*(i+1) \setminus SLUR^*(i)$.*

Proof. Let F_{i+1} be the complete CNF formula on $i+2$ variables $V = \{x_1, \ldots, x_{i+2}\}$, i.e. formula containing all 2^{i+2} possible clauses.

$$F_{i+1} = \bigwedge_{P \subseteq V} \left(\bigvee_{v \in P} v \vee \bigvee_{v \in V \setminus P} \overline{v} \right).$$

Now we can see that assigning values to any i-tuple of variables some of the clauses disappear and the rest of the formula is quadratic and unsatisfiable. E.g. if we assign arbitrary zero-one values to $x_1, \ldots, x_i$, we get $(x_{i+1} \vee x_{i+2})$ & $(x_{i+1} \vee \overline{x_{i+2}})$ & $(\overline{x_{i+1}} \vee x_{i+2})$ & $(\overline{x_{i+1}} \vee \overline{x_{i+2}})$. Unit propagation after each variable assignment does not help us, because all clauses are too long. The last *unitprop* will not derive the empty clause on such CNF and the algorithm SLUR*(i, F_{i+1}) will have to *give up* in the next step.

On the other hand, if we assign values to any $(i+1)$-tuple we will get a CNF of the x & $\overline{x}$ form. *Unitprop* will derive the empty clause on such a formula and so the algorithm SLUR*($i+1$,F_{i+1}) returns *unsatisfiable.* □

It can be immediately seen, that every CNF F on n variables belongs to SLUR*(n). 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 formula (4.1) defined in Example 40 at the beginning of this section shows that the inclusion SLUR(2) $\subseteq$ SLUR*(2) is in fact strict, this example can 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 43. *[4] For every $i > 1$ we have $[SLUR(i) \setminus SLUR(i-1)] \cap SLUR^*(2) \neq \emptyset$.*

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

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

This formula is logically equivalent to

$$(y_1 \leftrightarrow y_2 \leftrightarrow \ldots \leftrightarrow y_i) \ \& \ \Big(y_1 \vee \ldots \vee y_i \vee \big((a \vee b) \ \& \ (a \vee \bar{b}) \ \& \ (\bar{a} \vee b) \ \& \ (\bar{a} \vee \bar{b}) \big) \Big)$$

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 6 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 F belongs to SLUR*(2). □

The following is now an easy corollary.

Corollary 44. *[4] For every $i > 1$ we have that SLUR$(i) \subsetneq$ SLUR*(i).*

Let us now return to the classes CANON(i). Let $F = (x \vee y \vee a)$ & $(x \vee \overline{y} \vee b)$ & $(\overline{x} \vee y \vee c)$ & $(\overline{x} \vee \overline{y} \vee d)$, i.e. the CNF formula (3.6) defined on page 28. We have seen that this CNF belongs to CANON(2), but it is not SLUR. Now we can even observe that F 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 F does not even belong to SLUR*(2). By concatenating copies of F by disjoint union, we could in fact get an example of a formula showing the following lemma (we omit formal proof).

Lemma 45. *[4] For every $i \geq 1$ we have that $[SLUR^*(i) \setminus SLUR^*(i-1)] \cap CANON(2) \neq \emptyset$.*

This means, that CANON(2) is not a subclass of any level of SLUR*(i) hierarchy.

4.3 Hierarchy SLUR$_i$

In [21, 22] the previous two hierarchies were generalized yet further. In fact, they showed that the SLUR class coincides with the class of unit-refutation complete formulae and built up a hierarchy based on generalized variant of unit resolution. This hierarchy is called SLUR$_i$ and it generalizes even CANON(i) hierarchy, which was not included in the previous ones.

We will present some results of [21, 22] here. In order to do that it is natural to start with an algebraic definition of SLUR class presented there.

4.3.1 Algebraic definition of the SLUR class

The algebraic approach to the SLUR class definition presented in [21, 22] is based on relation '$\xrightarrow{\text{SLUR}}$' which corresponds to one step of SLUR algorithm consisting of assigning a value and performing unit propagation.

Definition 46. [21, 22] We say that formula F and formula F' that does not contain an empty clause are in relation $F \xrightarrow{\text{SLUR}} F'$ if there is a literal l in F such that F' is obtained from F by applying the partial assignment to that variable in l that sets l to 1 and performing unit propagation afterwards. The transitive and reflexive closure of relation '$\xrightarrow{\text{SLUR}}$' is then denoted by $F \xrightarrow{\text{SLUR}}_* F'$.

The actual SLUR class can be equivalently defined in the following way. Firstly, for a formula F a set of formulae reachable by '$\xrightarrow{\text{SLUR}}_*$' relation which has no successor, called $slur(F)$, is defined. Then SLUR formulae corresponds to a formulae F from which an empty clause can be derived by unit propagation or $slur(F)$ contains just the empty formula.

Definition 47. [21, 22] Let F be a formula. We define $slur(F) = \{F' : F \xrightarrow{\text{SLUR}}_* F' \ \& \ \neg\exists F'' : F' \xrightarrow{\text{SLUR}} F''\}$.

The SLUR class is then defined this way. SLUR $= \{F : [\mathit{unitprop}(F)$ does not contain an empty clause$] \Rightarrow [\mathit{slur}(F)$ contains only the empty formula$]\}$.

The equality between this SLUR definition to the original one is straightforward because the new one is nothing other than reformulating the algorithmic notion in more algebraic manner. The SLUR*(i) hierarchy can be defined in a similar way by using relation '$\xrightarrow{\text{SLUR}_i^*}$' that performs i valuations and decisions instead of just one. For further details see [21, 22].

4.3.2 Generalised unit propagation

In [21, 22] the authors used their previously discovered generalization of unit propagation [25] and developed a hierarchy over the SLUR class. This generalized propagation is defined as follows.

Definition 48. [21, 22, 25] Let $\mathcal{CLS}$ be a class of all formulae. Let $r_i : \mathcal{CLS} \to \mathcal{CLS}, i \in \mathbb{N}$ be mappings defined as follows:

· $r_0(F) = \begin{cases} \{\bot\} & \text{if } \bot \in F \\ F & \text{otherwise} \end{cases}$

· $r_{i+1}(F) = \begin{cases} r_{i+1}(F[l=1]) & \text{if } \exists \text{ literal } l : r_i(F[l=0]) = \{\bot\} \\ F & \text{otherwise} \end{cases}$

Then we call r_i *generalised unit clause propagation of level* i.

In the previous definition r_1 corresponds exactly to unit propagation. The fact that r_i is well-defined (i.e. it is defined for all formulae and the recursive definition is not inconsistent), as well as other properties of the generalized unit propagation can be found in [21, 22, 25]. In the rest of this section we shall need the following complexity result about generalized unit clause propagation.

Lemma 49. *[21, 22, 25] Let F be a formula over n variable which contains l literal occurrences. Then $r_i(F)$ can be computed in $O(l \cdot n^{2(i-1)})$ time and linear space.*

In [22, 25, 26] was also mentioned that relations r_i can serve as approximations of the relation $F \models C$ (i.e. relation 'C is implicate of F'). These mappings represent weaker variants of general resolution, which can be used to algorithmically test relation $F \models C$. In this sense notation $F \models_i C$ was used for these weaker variants (see definition bellow).

Definition 50. [22, 25, 26] Let F be a CNF formula and $C = (l_1 \vee \ldots \vee l_k)$ a clause. We write $F \models_i C$ if $r_i(F \;\&\; \overline{l_1} \;\&\; \ldots \;\&\; \overline{l_k}) = \{\bot\}$.

Note that for $i = 1$ it exactly coincides with the requirement we have in definition of unit refutation complete formula (Definition 16 on page 15).

The author of [26] also proved that relation r_i coincides with the so called *i-times nested input resolution*. It is defined using the *Horton-Strahler number*, which is a number defined for an arbitrary binary tree in the following way.

Definition 51. [1, 21, 22] Horton-Strahler number of a rooted binary tree T is defined as follows

· $hs(T) = 0$ for T with only one node,

- $\cdot$ $hs(T) = max(hs(T_1), hs(T_2))$ if for T with the left sub-tree T_1 and the right sub-tree T_2 holds $hs(T_1) \neq hs(T_2)$,
- $\cdot$ $hs(T) = max(hs(T_1), hs(T_2)) + 1$ if for T with the left sub-tree T_1 and the right sub-tree T_2 holds $hs(T_1) = hs(T_2)$.

Definition 52. [21, 22, 25, 26] We say that a clause C can be derived from a formula F in CNF by *i-times nested input resolution* if there exists $C' \subseteq C$ which can be derived by a tree resolution from F with resolution tree T satisfying $hs(T) \leq i$. We denote this fact by $F \vdash_i C$.

Lemma 53. *[22, 25, 26] For a formula F in CNF and a clause C it holds that $F \models_i C$ if and only if $F \vdash_i C$.*

4.3.3 SLUR$_i$ definition and summary of the results

Hierarchy SLUR$_i$ was defined in exactly the same way as the SLUR class in its algebraic definition but instead of using unit propagation r_1 generalised unit propagation r_i is used.

Definition 54. [21, 22] We say that two formulae F and F' are in relation $F \xrightarrow{\text{SLUR}:i} F'$ if there is a literal l such that $F' = r_i(F[l = 1])$ and F' is not a formula with an empty clause. Transitive and reflexive closure of this relation is denoted '$\xrightarrow{\text{SLUR}:i}_*$'.

Let $slur_i(F) = \{F' : F \xrightarrow{\text{SLUR}:i}_* F' \ \& \ \neg\exists F'' : F' \xrightarrow{\text{SLUR}:i}_* F''\}$, then the SLUR$_i$ hierarchy is defined as follows: SLUR$_i$ = $\{F$: [$r_i(F)$ does not contain an empty clause] $\Rightarrow$ [$slur_i(F)$ contains only the empty formula]$\}$.

The equality of SLUR and SLUR$_1$ is obvious, because we use unit propagation r_1 in this case. Thanks to the Lemma 49 we have that the satisfiability problem for SLUR*(i) formulae can be solved in $O(l \cdot n^{2i-1})$ time, where l is the number of literal occurrences in the input formula and n the number of variables.

In [21, 22] the authors also showed that this hierarchy is actually the same as a hierarchy they built over unit refutation complete formulae. The definition of this hierarchy uses the concept of hardness and is defined below.

Definition 55. [21, 22] The *hardness of a formula F* in CNF, denoted by $hd(F)$, is the minimal i such that for all clauses C satisfying $F \models C$ we have $F \models_i C$ (or equivalently $F \vdash_i C$ thanks to Lemma 53).

Definition 56. [21, 22] The class of *unit refutation complete formulae of level i* is $UC_i = \{F : F \text{ is a CNF}, hd(F) \leq k\}$.

Determining the hardness of a formula F, and consequently the membership problem of the class UC_i is coNP-complete. However the following equivalence, which follows directly from Lemma 53, is known.

Theorem 57. *[22] A CNF formula F is in UC_i if and only if for every prime implicate C we have that $F \vdash_i C$.*

Lemma 58. *[21, 22] Let F be a formula in CNF and let v be a partial assignment such that $r_i(F[v]) \neq \{\perp\}$ then $F \xrightarrow{SLUR:i}_* r_i(F[v])$ holds.*

Proof. It follows from the fact that assignments of v can be performed by '$\xrightarrow{\text{SLUR}:i}$' transitions. □

Theorem 59. *[21, 22] For every i $SLUR_i = UC_i$.*

Proof. The equivalence corresponds to the fact that for any formula F it is true that $F \in SLUR_i \Leftrightarrow hd(F) \leq i$. If F is unsatisfiable the equivalence follows directly from the definitions.

Let now $F \in SLUR_i$ be a satisfiable formula and let us consider a partial assignment v such that $F[v]$ is unsatisfiable. We have to prove that $r_i(F[v]) = \{\bot\}$. Let us assume for contradiction that $r_i(F[v]) \neq \{\bot\}$ holds. It follows that $F \xrightarrow{\text{SLUR}:i}_* r_i(F[v])$ by Lemma 58. Generally it holds that if F is satisfiable, $F \in SLUR_i$ and $F \xrightarrow{\text{SLUR}:i}_* F'$ then F' is satisfiable. Thus $r_i(F[v])$ is satisfiable, which is a contradiction with the assumption of unsatisfiability of $F[v]$. We can conclude that $hd(F) \leq i$.

For the other direction let us assume that $hd(F) \leq i$. We show that $F \in SLUR_i$, i.e. $slur_i(F)$ contains only the empty formula (which is always satisfied). Assume for contradiction that there is $F' \in slur_i(F)$ other than the empty formula. According to the condition $\neg\exists F'' : F' \xrightarrow{\text{SLUR}:i}_* F''$ in definition of $slur_i(F)$ (see Definition 54) we have that F' is unsatisfiable but $r_i(F') \neq \{\bot\}$. However by Lemma 3.11 in [25] hardness cannot be increased by applying a partial assignment, i.e. $hd(F') \leq i$ and therefore $r_i(F') = \{\bot\}$. □

4.4 Mutual relations of the hierarchies in inclusion

In this section we present known results [4, 21, 22] about mutual relations of presented hierarchies under inclusion. The first one is rather clear.

Lemma 60. *[4] SLUR$\subsetneq$SLUR(i)$\subseteq$SLUR*(i) for every $i \geq 1$.*
SLUR(i)$\subsetneq$SLUR(i) for $i \geq 2$.*

Proof. The first inclusion is obvious from the definitions of SLUR class and SLUR(i) extension. Its strictness is a consequence of modification in the SLUR algorithm we proposed (not *giving up* after the first round if all the branches lead to contradiction).

The mutual relation between SLUR(i) and SLUR*(i) for $i \geq 2$ was already discussed in Corollary 44 on page 35. Note that for $i = 1$ the classes SLUR(i) and SLUR*(i) are exactly the same as the algorithms perform equally in this case. □

In Lemma 35 we proved that CANON(1) is contained in the SLUR class. This obviously holds for every class that extends SLUR class, i.e. all the hierarchies presented. On the other hand, there is a formula from CANON (2) which is neither SLUR(i) nor SLUR*(i).

Lemma 61. *[4] CANON(2) is not contained in SLUR*(i).*

Proof. The proof follows simply by concatenating i copies of the formula (3.6) mentioned below Example 36 on page 28. □

In [21, 22] it was proven that hierarchy SLUR$_i$ generalizes all SLUR, SLUR(i), SLUR*(i) and CANON(i) hierarchies as well as the class of generalized unit refutation complete formulae UC$_i$, and as a consequence of this result we see that SLUR contains the class of propagation complete formulae, as well. The direct proof of that fact can be found in [3].

Theorem 62. *[22] For every i it holds:*

1. $CANON(i) \subseteq UC_i$,
2. $UC_1 \not\subseteq CANON(i)$,
3. $SLUR^*(i) \subsetneq SLUR_{i+1}$,
4. $SLUR_2 \not\subseteq SLUR^*(i)$,
5. *for every* $i \geq 2$ *it is* $SLUR^*(i) \not\subseteq SLUR_i$ *and* $SLUR_i \not\subseteq SLUR^*(i)$.

Proof.

1. The fact that $CANON(i) \subseteq UC_i$ is a consequence of Theorem 57 and the fact that Horton-Strahler number is at most the depth (i.e. weight) of the resolution tree.
2. In order to prove $UC_1 \not\subseteq CANON(i)$ let us consider the following formula: x_{2^i} & $(x_0 \vee \overline{x_1})$ & $(x_1 \vee \overline{x_2})$ & $\ldots$ & $(x_{2^i-2} \vee \overline{x_{2^i-1}})$ & $(x_{2^i-1} \vee \overline{x_{2^i}})$. It is a SLUR (i.e. UC_1) formula because it is completely solvable by unit propagation, however it is not in CANON(i) since the derivation with the smallest depth has to use a binary tree of height at least i to deal with all 2^i binary clauses and one more resolution in order to derive prime implicate x_0.

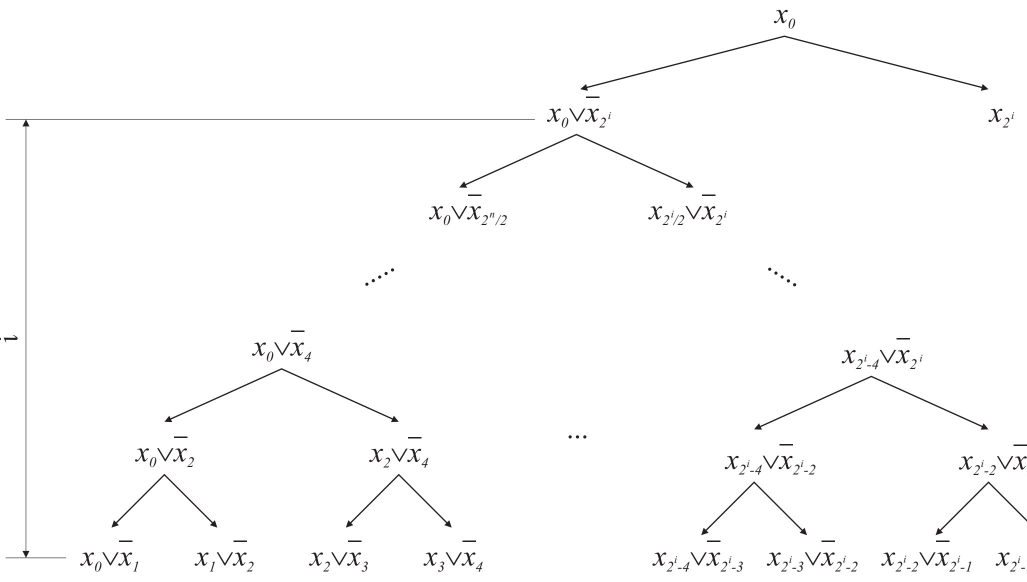

x_0
$x_0 \vee \overline{x}_{2^i}$
x_{2^i}
$x_0 \vee \overline{x}_{2^n/2}$
$x_{2^i/2} \vee \overline{x}_{2^i}$
i
$x_0 \vee \overline{x}_4$
$x_{2^i-4} \vee \overline{x}_{2^i}$
$x_0 \vee \overline{x}_2$
$x_2 \vee \overline{x}_4$
$x_{2^i-4} \vee \overline{x}_{2^i-2}$
$x_{2^i-2} \vee \overline{x}_{2^i}$
$x_0 \vee \overline{x}_1$
$x_1 \vee \overline{x}_2$
$x_2 \vee \overline{x}_3$
$x_3 \vee \overline{x}_4$
$x_{2^i-4} \vee \overline{x}_{2^i-3}$
$x_{2^i-3} \vee \overline{x}_{2^i-2}$
$x_{2^i-2} \vee \overline{x}_{2^i-1}$
$x_{2^i-1} \vee \overline{x}_{2^i}$

Figure 4.1: Example of a UC_1 formula that is not CANON(i).

3. $SLUR^*(i) \subseteq SLUR_{i+1}$ is given by the fact that hardness of a formula does not increase after partial assignment (see Lemma 3.11 in [25]) and by an easy fact that $slur^*(k)(F) = \{F\}$ for F other than the empty formula implies $r_{i+1}(F) = \{\perp\}$. The strictness follows from the fact that the hardness of a formula is always bounded by the number of variables (Lemma 3.18 in [25]).
4. $SLUR_2 \not\subseteq SLUR^*(i)$ follows from Lemma 61, the first item of this theorem and equality $UC_i = SLUR_i$.

5. The last item follows from the previous one in the direction $SLUR_i \not\subseteq SLUR^*(i)$. The other direction is given by the observation made in Lemma 42 that full unsatisfiable clause on i variables F_i is element of $SLUR(i+1)$ but it is not in $SLUR^*(i)$ because $hd(F_{i+1}) = i + 1$ (by Lemma 3.18 in [25]).

□

All the inclusions proved in Theorem 62 are depicted in Figure 4.2.

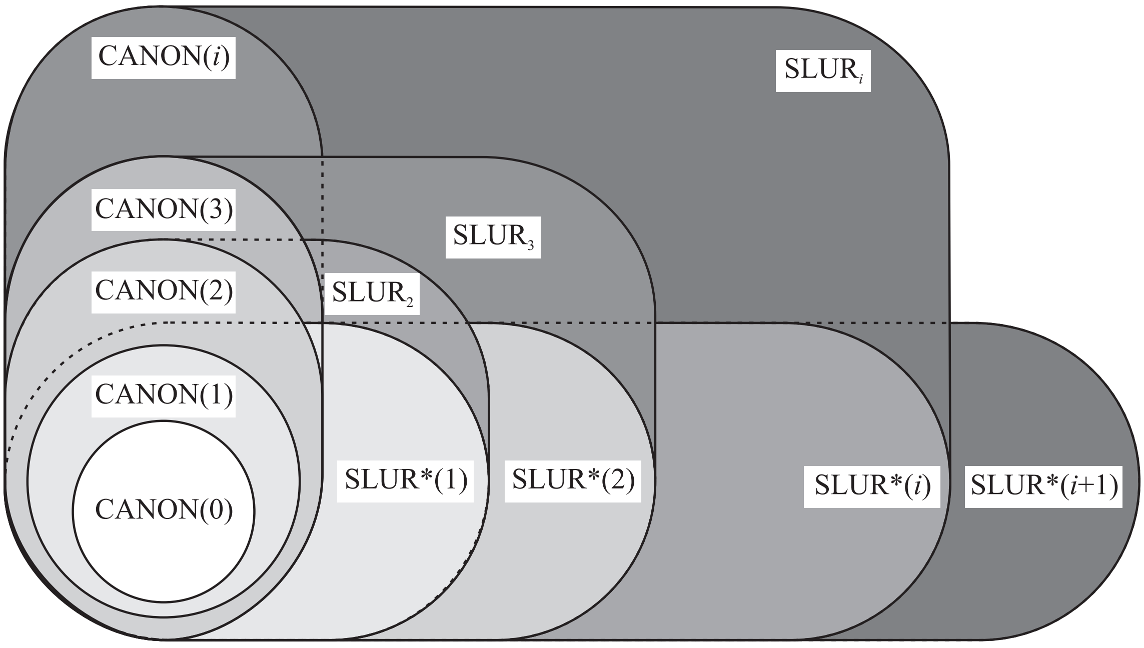


Figure 4.2: Mutual inclusions of CANON(i), SLUR*(i) and SLUR$_i$ hierarchies.

Conclusion

The main topic of this dissertation is the study of the SLUR class of Boolean formulae. We showed that the problem of recognition of SLUR formulae is coNP-complete. On the other hand, we proved that every formulae which allows deriving all prime implicates in at most one resolution step is SLUR.

Despite the fact, that it is very improbable to find a polynomial time algorithm for the membership problem, the SLUR class is still interesting as a target class for compiling the knowledge into, because the algorithm for testing SLUR satisfiability does not need to know whether the input formula is from the SLUR class or not. If it is, it guarantees that it will answer, however it may still answer correctly in the other case (particularly if the formula is satisfiable and the algorithm makes right choices).

The same holds about the hierarchies presented in Chapter 4. These hierarchies extend the idea of the original SLUR algorithm such that they apply more time consuming operations in each round of the main cycle. The most basic SLUR(i) hierarchy just picks i variables instead of just one. Its extension SLUR*(i) applies unit propagation after each substitution to one of the chosen variables, instead of just one unit propagation after all i variables are chosen and substituted into. The last hierarchy SLUR$_i$ uses a more complex variant of unit propagation. All of these hierarchies slow down the satisfiability testing exponentially in the parameter i and the membership remains coNP-complete for every level of every hierarchy.

Bibliography

[1] Ansótegui, Carlos, Bonet, María L., Levy, Jordi, Manyà, Felip. Measuring the hardness of SAT instances. In Proceedings of the 23th AAAI Conference on Artificial Intelligence (AAAI-08) (eds. Fox D., Gomes C.). 2008, pp. 222-228.

[2] Aspvall, Bengt. Recognizing Disguised NR(1) Instances of the Satisfiability Problem. *Journal of Algorithms.* Volume 1, Issue 1, March 1980, pp. 97–103.

[3] Babka, Martin, Balyo, Tomáš, Čepek, Ondřej, Gurský, Štefan, Kučera, Petr, Vlček, Václav. Complexity Issues Related to Propagation Completeness. Submitted to Artifficial Intelligence.

[4] Balyo, Tomáš, Gurský, Štefan, Kučera, Petr and Vlček, Václav. On hierarchies over the SLUR class [online]. In *International Symposium on Artificial Intelligence and Mathematics (ISAIM 2012)*, January 2012. Available at `http://www.cs.uic.edu/pub/Isaim2012/WebPreferences/ISAIM2012_Boolean_Balyo_etal.pdf`.

[5] Biere, Armin, Heule, Marijn, van Maaren, Hans, Walsh, Toby (Eds.). *Handbook of Satisfiability.* IOS Press, 2009. ISBN 978-1-58603-929-5.

[6] Bordeaux, Lucas, Marques-Silva, João . Knowledge Compilation with Empowerment In *SOFSEM 2012: Theory and Practice of Computer Science* (eds. Bielikova M., Friedrich G., Gottlob G., Katzenbeisser S., Turán, G.) LNCS 7147. Springer Berlin Heidelberg, 2012, pp. 612-624. ISBN 978-3-642-27659-0 (print), ISBN 978-3-642-27660-6 (online). DOI: 10.1007/978-3-642-27660-6_50.

[7] Chandru, Vijay, Hooker, John N. Extended Horn sets in propositional logic. Journal of the ACM. Volume 38 Issue 1, January 1991, pp. 205-221. DOI: 10.1145/102782.102789.

[8] Conforti, Michele, Cornuéjols, Gérard, Kapoor, Ajai, Vušković, Kristina, Rao, M. R. Balanced Matrices. In *Mathematical Programming: State of the Art.* (eds. Birge J. R., Murty K. G.). Braun-Brumfield, United States. Produced in association with the 15th Int'l Symposium on Mathematical Programming, University of Michigan, 1994.

[9] Cook, Stephen A. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing* STOC '71. ACM, New York, NY, USA, 1971, pp. 151-158.

[10] Čepek, Ondřej. Private communication.

[11] Čepek, Ondřej, Kučera, Petr, Vlček, Václav. Properties of SLUR Formulae. In *SOFSEM 2012: Theory and Practice of Computer Science* (eds. Bielikova M., Friedrich G., Gottlob G., Katzenbeisser S., Turán, G.) LNCS 7147. Springer Berlin Heidelberg, 2012, pp. 177-189. ISBN 978-3-642-27659-0 (print), ISBN 978-3-642-27660-6 (online). DOI: 10.1007/978-3-642-27660-6_15.

[12] Dalal, Mukesh, Etherington, David W. A hierarchy of tractable satisfiability problems. *Information Processing Letters*. Volume 44, Issue 4, 10 December 1992, pp. 173–180. ISSN 00200190. DOI: 10.1016/0020-0190(92)90081-6.

[13] Darwiche, Adnan, Marquis, Pierre. A Knowledge Compilation Map. *Journal of Artificial Intelligence Research*. Volume 17, 2002, pp. 229-264. DOI: 10.1613/jair.989.

[14] Davis, Martin, Logemann, George, Loveland, Donald. A machine program for theorem-proving. *Communications of the ACM*. New York: Association for Computing Machinery, Volume 5, Issue 7, July 1962, pp. 394-397. ISSN 00010782. DOI: 10.1145/368273.368557.

[15] Davis, Martin, Putnam, Hilary. A Computing Procedure for Quantification Theory. *Journal of the ACM*. Volume 7, Issue 3, July 1960, pp. 201-215. ISSN 00045411. DOI: 10.1145/321033.321034.

[16] del Val, Alvaro. Tractable databases: How to make propositional unit resolution complete through compilation. In *Proceedings of the 4th International Conference on Principles of Knowledge Representation and Reasoning (KR'94)*. pp. 551-561.

[17] Dowling, William F., Gallier, Jean H. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *The Journal of Logic Programming*. Volume 1, Issue 3, October 1984, pp. 267–284. DOI: 10.1016/0743-1066(84)90014-1.

[18] Franco, John, van Gelder, Allen. A perspective on certain polynomial-time solvable classes of satisfiability. *Discrete Applied Mathematics*. Volume 125, 2-3, 2003, p. 177-214. DOI: 10.1016/S0166-218X(01)00358-4.

[19] Garey, Michael R., Johnson, David S. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979. ISBN 0-7167-1045-5.

[20] Genesereth, Michael R., Nilson, Nils J. *Logical foundations of artificial intelligence: a guide to the theory of NP-completeness*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1987. ISBN 0-934613-31-1.

[21] Gwyne, Matthew, Kullmann, Oliver. Generalising and unifying SLUR and unit-refutation completeness. In *SOFSEM 2013: Theory and Practice of Computer Science*. (eds. van Emde Boas P., Groen F. C. A. , Italiano G. F., Nawrocki J., Sack H.), LNCS 7741, Springer, 2013, pages 220–232. ISBN 978-3-642-35842-5 (print), 978-3-642-35843-2 (online). DOI: 10.1007/978-3-642-35843-2_20.

[22] Gwyne, Matthew, Kullmann, Oliver. *Generalising unit-refutation completeness and SLUR via nested input resolution*. Journal of Automated Reasoning, 2013. To appear.

[23] Haken, Armin. The Intractability of Resolution. *Theoretical Computer Science.* Vol. 39, January 1985, pp. 297-308. ISSN 03043975. DOI: 10.1016/0304-3975(85)90144-6.

[24] Karp, Richard M. Reducibility among combinatorial problems. Complexity of Computer Computations (eds. Miller R. E., Thatcher J. W.). Plenum Press, 1972, pp. 85-103.

[25] Kullmann, Oliver. Investigating a general hierarchy of polynomially decidable classes of CNF's based on short tree-like resolution proofs. Technical Report TR99-041, *Electronic Colloquium on Computational Complexity (ECCC)*, October 1999. Available at `http://eccc.hpi-web.de/report/1999/041/`.

[26] Kullmann, Oliver. Upper and lower bounds on the complexity of generalised resolution and generalised constrained satisfaction problem *Annals of Mathematics and Artificial Intelligence*, 2004, Volume 40, Issue 3-4 , pp. 303-352. ISSN 1012-2443 (print), 1573-7470 (online). DOI: 10.1023/B:AMAI.0000012871.08577.0b.

[27] Lewis, Harry R. Renaming a Set of Clauses as a Horn Set. *Journal of the ACM.* Volume 25 Issue 1, January 1978, pp. 134-135.

[28] Marques-Silva, João P., Sakallah, Karem A. GRASP: A new search algorithm for satisfiability. In *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, ICCAD '96. IEEE Computer Society, San Jose, California, USA, November 1996, pp. 220-227. ISBN 0-8186-7597-7.

[29] Quine, Willard van O. The problem of simplifying truth functions. *The American Mathematical Monthly.* Vol. 59, No. 8 (Oct., 1952), pp. 521-531. Mathematical Association of America.

[30] Quine, Willard van O. *A Way to Simplify Truth Functions.* In *The American Mathematical Monthly.* Vol. 62, No. 9 (Nov., 1955), pp. 627-631. Mathematical Association of America.

[31] Schlipf, John S., Annexstein, Fred S., Franco, John V., Swaminathan, R.P. On finding solutions for extended Horn formulas. *Information Processing Letters.* 1995, Volume 54, Issue 3, p. 133-137. DOI: 10.1016/0020-0190(95)00019-9.

[32] Scutellà, Maria Grazia. A note on Dowling and Gallier's top-down algorithm for propositional Horn satisfiability. *The Journal of Logic Programming.* Volume 8, Issue 3, May 1990, pp. 265–273. DOI: 10.1016/0743-1066(90)90026-2.

[33] Somenzi, Fabio. Binary Decision Diagrams. Calculational System Design, Volume 173 of NATO Science Series F: Computer and Systems Sciences. 1999.

[34] Tseitin, Grigorii S. On the complexity of derivation in propositional calculus. In *Studies in Constructive Mathematics and Mathematical Logic* (ed. Slisenko A. O.), Part 2. Consultants Bureau, New York, 1970, pp. 115–125. Reprinted in [20], Vol. 2, pp. 466–483.

[35] Urquhart, Alasdair. The Depth of Resolution Proofs. *Studia Logica*, 2011, Volume 99 Issue 1-3, pp. 349-364. ISSN 0039-3215 (print), 1572-8730 (online). DOI: 10.1007/s11225-011-9356-9.

[36] Vlček, Václav. *Classes of Boolean Formulae with Effectively Solvable SAT.* In *WDS'10 Proceedings of Contributed Papers: Part I — Mathematics and Computer Sciences* (eds. Šafránková J., Pavlů J.). Matfyzpress, Prague, pp. 42–47, 2010. ISBN 978-80-7378-139-2. Available at `http://www.mff.cuni.cz/veda/konference/wds/proc/pdf10/WDS10_107_i1_Vlcek.pdf`.

[37] Vlček, Václav. *Třídy Booleovských formulí s efektivně řešitelným SATem.* Prague, 2009. Master thesis [in Czech]. Charles University in Prague. Faculty of Mathematics and Physics. Supervised by Ondřej Čepek.

[38] Wegener, Ingo. *The Complexity of Boolean Functions.* John Wiley & Sons, Inc. New York, NY, USA, 1987. ISBN 0-471-91555-6.