Solving the SAT Problem by Hyper-Unit Propagation*

Gábor Kusper
gkusper@risc.uni-linz.ac.at

Research Institute for Symbolic Computation (RISC-Linz)
Johannes Kepler University, Linz, Austria
http://www.risc.uni-linz.ac.at/

January 28, 2002

Abstract

The SAT problem is the problem of finding a model for a formula in conjunctive normal form. We developed **two algorithms based on hyper-unit propagation**, which **solve** the **SAT** problem. Hyper-unit propagation is unit propagation simultaneously by literals, as unit clauses, of an assignment. The first method, called **Unicorn-SAT** algorithm, **solves the resolution-free SAT problem in linear time**. A formula is resolution-free if and only if there are no two clauses, which differ only in one variable. For such a restricted formula we can find a model in linear time by hyper-unit propagation. We obtain a sub-model, i.e., a part of the model, by negation of a resolution-mate of a minimal clause, which is a clause with the smallest number of literals in the formula. We obtain a resolution-mate of a clause by negating one literal in it. By hyper-unit propagation by a sub-model we obtain a formula, which has fewer variables and clauses and remains resolution-free. Therefore, we can obtain a model by joining the sub-models while we perform hyper-unit propagation by a sub-model recursively until the formula becomes empty. The second method, called **General-Unicorn-SAT** algorithm, **solves the general SAT problem**. The algorithm is the same as Unicorn-SAT but in the general case this method may result in an unsatisfiable formula, i.e., the Unicorn-SAT is not complete for the genral SAT problem. To become complete we use backtrack. If the obtained intermediate formula is unsatisfiable we backtrack and we add the resolvent of the last chosen minimal clause and its resolution-mate, which is the negation of the sub-model generated from this minimal clause. This resolvent will be the minimal clause of the formula. If this resolvent is the empty clause then we do backtrack again. If it is not possible then the input problem is unsatisfiable. If we reach the empty formula then the union of intermediate sub-models is a model for the input problem. With this method, called General Unicorn-SAT, we find a model for a formula if it is satisfiable or we detect if it is unsatisfiable.

* Sponsored by Upper Austrian Government (scholarship), FWF SFB F013 (P1302) and Austro-Hungarian Action Foundation.

1 Introduction

Propositional Satisfiability is the problem of determining, for a formula of the propositional calculus, if there is an assignment of truth values to its variables for which that formula evaluates the true. By SAT we mean the problem of propositional satisfiability for formulae in conjunctive normal form (CNF).

SAT is the first, and one of the simplest, of the many problems which have been shown to be **NP**-complete [1]. It is dual of propositional theorem proving, and many practical **NP**-hard problems may be transformed efficiently to SAT. Thus, a good SAT algorithm would likely have considerable utility. It seems improbable that a polynomial time algorithm will be found for the general SAT problem but we know there are restricted SAT problems that are solvable in polynomial time. So a "good" SAT algorithm should check first the input SAT instance whether it is an instance of such a restricted SAT problem. In this paper we introduce the resolution-free SAT problem, which is solvable in polynomial time.

The following restricted SAT problems are solvable in polynomial time:

1. The restriction of SAT to instances where all clauses have length k is denoted by k-SAT. Of special interest are 2-SAT and 3-SAT: 3 is the smallest value of k for which k-SAT is **NP**-complete, while 2-SAT is solvable in linear time [2, 3].
2. Horn-SAT is the restriction to instances where each clause has at most one unnegated variable. Horn SAT is solvable in linear time [4, 5], as are a number of generalizations such as renameable Horn SAT [6], extended Horn SAT [7] and q-Horn SAT [8, 9].
3. The hierarchy of tractable satisfiability problems [10], which is based on Horn SAT and 2-SAT, is solvable in polynomial time. An instance on the k level of the hierarchy is solvable in $O(n^{k+1})$ time.
4. Nested SAT, in which there is a linear ordering on the variables and no two clauses overlap with respect to the interval defined by the variables they contain [11].
5. SAT in which no variable appears more than twice. All such problems are satisfiable if they contain no unit clauses [12].
6. r,r-SAT, where r,s-SAT is the class of problems in which every clause has exactly r literals and every variable has at most s occurrences. All r,r-SAT problems are satisfiable in polynomial time [12].

The resolution-free SAT problem is also a restriction of SAT to instances where no resolution (see definition of resolution, which is not the usual one, since we consider only resolution resulting in a non-tautologous resolvent) can be performed on any two clauses of the formula. In other words, any two clauses either overlap (they do not differ in any variable) or they differ in at least in two variables. This could be a real problem if we want to prove a theorem by resolution but we do not reach the empty clause (it means it does not hold) after doing all possible resolutions then we may have a resolution-free formula. If it is the case then the Unicorn-SAT algorithm provides a model for this formula in linear time, and this model contains essential information why the theorem does not hold.

However we can generalize Unicorn-SAT such that it solves the general SAT problem. We call the generalized algorithm General Unicorn-SAT. It is complete for the general SAT problem, i.e., it can decide whether an SAT instance is satisfiable or not. If it is satisfiable then returns a model for it.

Intuitively Unicorn-SAT works as follows. We perform hyper-unit propagation by a sub-model until our formula becomes empty. We obtain the model by joining the sub-models. To be more precise: We obtain a sub-model, i.e., a part of the model, by negation of a resolution-mate of a minimal clause, which is a clause with the smallest number of literals in the formula. We obtain a resolution-mate of a clause by negating one literal in it. By hyper-unit propagation by a sub-model we obtain a formula, which has fewer variables and clauses and remains resolution-free. Therefore, we can obtain a model by joining the sub-models while we perform hyper-unit propagation by a sub-model recursively until the formula becomes empty.

The General Unicorn-SAT algorithm uses the same method as Unicorn-SAT, but since it works on a general SAT problem it may run into unsuccessful branches. In this case it does a backtrack- and a resolution step. To be more precise, the General Unicorn-SAT algorithm assumes that it works on a resolution-free SAT problem. If it turns out that it is not resolution-free, i.e., resolution can take place, then it makes a resolution step, which simplifies the problem. By this trick we lose efficiency but the algorithm becomes complete.

This paper contains description of two algorithms. The first one is described by Section 3 - 6. The second is the generalization of the first one. It is described by Section 7 - 10. In Section 2 we give common definitions.

In Section 6 we introduce the Unicorn-SAT algorithm and we show it is correct and solves the resolution-free SAT problem in linear time. In Section 3 we show important properties of negation of a resolution-mate. In Section 4 we show important properties of hyper-unit propagation. In Section 5 we prove lemmas that are needed to prove that the Unicorn-SAT is correct. In Section 6 we show an example how the algorithm works.

In Section 9 we introduce the General Unicorn-SAT algorithm and we show it is correct and solves the general SAT problem in non-polynomial time. In Section 7 we give definitions for notions used in General Unicorn-SAT. In Section 8 we prove theorems that are needed to prove that the General Unicorn-SAT is complete. In Section 10 we show some examples how the algorithm works. Section 11 is the conclusion.

Appendix A contains the recursive version of the General Unicorn-SAT algorithm.

2 Definitions

Let V be a set of Boolean variables. The negation of a variable v is denoted by $\bar{v}$. Given a set U, we denote $\overline{U} = \{\bar{u} \mid u \in U\}$. *Literals* are the members of the set $W = V \cup \overline{V}$. *Positive literals* are the members of the set V. *Negative literals* are their negations. If w denotes a negative literal $\bar{v}$, then $\bar{w}$ denotes the positive literal v. A *clause* is a finite set of literals that does not contain simultaneously any literal together with its negation. The empty clause, denoted by ∅, is interpreted as *False*. A *formula* (*formula in CNF*) is a finite set of clauses. The empty formula, denoted by ∅, is interpreted as *True*. The *length of a clause* C is its cardinality, denoted by |C|. If |C| is k we say that C is a *k-clause*. Special cases are *unit clauses* which are *1-clauses*, and *clear clauses* which are *n-clauses*

where n = |V|. The *length of a formula* S is sum of length of its clauses, denoted by |S|. Clauses of a formula with minimal length are called *minimal clauses* of this formula. If we say that a *literal* v *occurs* in a clause or in a formula, we mean that this clause or this formula contains the literal v. However, if we say that a *variable* v *occurs* in a clause or in a formula, we mean that this clause or this formula contains the literal v, or it contains the literal $\bar{v}$. We denote by V{S} the *set of variables occurring in* the formula S, and by |V{S}| *number of variables occurring in* S. We say that two clauses *differ in some variables* if these variables occur in both clauses but as different literals. The clause C *subsumes* the clause B if and only if every literal in C occur in B. For example {v} subsumes {v, w}. We say that two clauses *overlap* if and only if they differ in no variable (they overlap because there is at least one clause they subsume commonly). We say that *resolution can be performed on* two clauses if they differ only in one variable. Note that this is not the usual notion of resolution because we allow resolution only if it results in a non-tautologous resolvent. For example resolution cannot be performed on {v, w} and $\{\bar{v}, \bar{w}\}$ but can on {v, w} and $\{\bar{v}, z\}$. If resolution can be performed on two clauses, say A and B, then the *resolvent*, denoted by Res(A,B), of this two clauses is the union of them excluded the variable they differ in. A formula is *resolution-free* if and only if no resolution can be performed on any two clauses of the formula. An *assignment* is a finite subset of W that does not contain any literal together with its negation. Informally speaking, if an assignment I contains a literal v, it means that v has the value *True* in I. Note that both, a clause and an assignment, are finite set of literals. We obtain *a resolution-mate of* a clause by negating one literal of it. *Negation of resolution-mate*, also called *lucky-negation*, of a clause C, denoted by L(C), is the negation of a resolution-mate of C. By definition L(∅) = ∅. A *sub-model* is a negation of resolution-mate of a minimal clause. Note that we may use L(C) also as an assignment and a clause.
Now we define the crucial notion of *hyper-unit propagation*.

Definition 2.1 (Hyper-Unit Propagation)

S[I] (read: "hyper-unit propagation by assignment I on formula S") :=

$\{C \setminus \bar{I} \mid C \in S \wedge C \cap I = \varnothing\}$,

where S is a formula and I is an assignment.

C[I] (read: "hyper-unit propagation by assignment I on clause C") :=

$C \setminus \bar{I}$, if $C \cap I = \varnothing$, undefined otherwise,

where C is a clause and I is an assignment.

Note that sometimes we use sub-model propagation as a synonym of hyper-unit propagation.

Informally S[I] means for the formula S and the sub-model I = $\{v_1,\ldots,v_n\}$, we

- remove from S all clauses containing the literals v_i, and
- we delete all occurrences of the literals $\bar{v}_i$ from the other clauses, i = 1..n.

With other words hyper-unit propagation on S by I is equivalent to unit propagation [13] on S by each literal of I considering each literal as a unit clause. An assignment M is a *model* for a formula S if S[M] is the empty formula. A formula is *satisfiable* if there exists a model assignment for it.

3 Properties of Negation of Resolution-Mate

In this section let C be a non-empty clause.

Lemma 3.1 (Trivial Properties of Negation of Resolution-Mate)

L(C) is a clause, is an assignment, contains exactly one literal occurring in C, contains the negation of all other literals occurring in C, contains all variables of C.

At this point the reader should realize that in this paper clause and assignment are defined by the same definition. Thus, we can use L(C) either as a clause or an assignment.

Lemma 3.2

Resolution can be performed on C and L($\overline{C}$).

Proof:

From Lemma 3.1 we know that C and L(C) differ in every variable of C except in one. So C and L($\overline{C}$) differ only in this literal.

Hence, resolution can be performed on C and L($\overline{C}$). ❑

Lemma 3.3 (The Lucky Property)

Let S be a resolution-free formula that does not contain the empty clause. Let C be an element of S. Then L($\overline{C}$) is not element of S.

Proof:

If L($\overline{C}$) would be an element of S then S would be non resolution-free since we know from Lemma 3.2 that resolution can be performed on C and L($\overline{C}$).

Hence, L($\overline{C}$) is not element of S. ❑

Negation of resolution-mate has a "lucky" property namely it provides a clause outside the resolution-free formula.

Example 3.4

Let C := $\{a, \overline{b}, \overline{c}\}$. Since C has 3 literals it has 3 possible resolution-mate D, E and F:

1. D:=$\{\overline{a}, \overline{b}, \overline{c}\}$, Res(C,D) = $\{\overline{b}, \overline{c}\}$;
2. E:=$\{a, b, \overline{c}\}$, Res(C,E) = $\{a, \overline{c}\}$;
3. F:=$\{a, \overline{b}, c\}$, Res(C,F) = $\{a, \overline{b}\}$.

Since C has 3 resolution-mate L(C) has 3 possible value: either $\{a, b, c\}$, which is negation of D, or $\{\overline{a}, \overline{b}, c\}$, which is negation of E, or $\{\overline{a}, b, \overline{c}\}$, which is negation of F.

4 Properties of Hyper-Unit Propagation

In this section we will see that formula remains the same for certain properties after hyper-unit propagation by the negation of a resolution-mate of a minimal clause of the formula.

Lemma 4.1 (Hyper-Unit Propagation Preserves $\emptyset \notin$ S if)

Let S be a resolution-free formula that does not contain the empty clause. Let C be a minimal clause of S. Then S[L(C)] does not contain the empty clause.

Proof:

It is suffices to show that there is no such a clause in S such that hyper-unit propagation by L(C) deletes every literal from it, i.e., L($\overline{C}$) is not element of S. (We do not need

consider clauses which are subset of L($\overline{C}$) since C is a minimal clause from S). We know from Lemma 3.3 that L($\overline{C}$) is not element of S.
Hence, S[L(C)] does not contain the empty clause. ❑

We see that negation of a resolution-mate allows to simplify (perform hyper-unit propagation) the formula without the danger that the formula becomes unsatisfiable (it will not contain the empty clause).

Lemma 4.2 (Hyper-Unit Propagation Preserves Resolution-Free-ness)
Let S be a resolution-free formula and let I be an assignment. Then S[I] is resolution-free.
Proof:
Let B' and C' be elements of S[I]. Then we know, by definition of hyper-unit propagation, that B' = B[I] and C' = C[I] for some elements B and C of S. It can be shown that B and $\overline{I}$ overlap and C and $\overline{I}$ overlap, i.e., B and C differ in no variable among variables of I. From this and from the definition of hyper-unit propagation we can obtain that B[I] and C[I] differ in the same variables like B and C, i.e., no resolution can be performed on B[I] and C[I]. Hence, S[I] is resolution-free. ❑

Now we see that a resolution-free formula that does not contain the empty clause has the same properties after hyper-unit propagation by negation of a resolution-mate of some minimal clause of the formula.

Lemma 4.3
Let S be a formula, A and B be assignments such that $A \cap B = \emptyset$. Let $C = A \cup B$. Then S[C] = S[A][B].
Proof:
Since A and B have no common variable $A \cup B$ is an assignment. Hence, $S[A \cup B]$ is defined and, by definition of hyper-unit propagation and union, $S[A \cup B] = S[A][B]$.
Hence, S[C] = S[A][B]. ❑

This lemma allows us to join the sub-models to obtain the final model. It is stated by the following corollary.

Corollary 4.3.1
Let X be a formula, I be an assignment. Let S be a formula such that X[I] = S. Let B be an element of S. Then $X[I \cup B] = S[B]$.
Proof:
It is an easy consequence of Lemma 4.3 but we give a proof for it. I and B have no common variable since B is element of S = X[I]. Therefore $X[I \cup B] = X[I][B]$. Hence, $X[I \cup B] = S[B]$. ❑

Lemma 4.4 (Hyper-Unit Propagation Simplifies the Problem)
Let S be a non-empty clause that does not contain the empty clause. Let B be an element of S. Then $|S[L(B)]| < |S|$ and $|V\{S[L(B)]\}| < |V\{S\}|$.
Proof:
S and L(B) have at least one common variable.
Hence, $|S[L(B)]| < |S|$ and $|V\{S[L(B)]\}| < |V\{S\}|$. ❑

5 The Unicorn-SAT Algorithm

Now we introduce the Unicorn-SAT algorithm and we show that it solves the resolution-free SAT problem in linear time.

Let RfS(S) be the predicate which is true if S is resolution-free formula, false otherwise. Let MinC(S) be the set of minimal clauses of S. Let M(S, I) be the predicate which is true if I is a model for S, false otherwise.

We use "{}" to mark formulae that are true at the respective points of algorithm, in order to prove the correctness of the algorithm in the Hoare calculus.

Algorithm 5.1 (Unicorn-SAT):

```
Unicorn-SAT(X, Z)
input: formula X that is resolution-free and does not
contain the empty clause,
output: clause Z, a model of X.
START
{∅∉X ∧ RfS(X)}                              precondition
{∅∉X ∧ RfS(X) ∧ X[∅]=X}                     by a trivial
                                            property of hyper-
                                            unit propagation
(S,I):=(X,∅);
{∅∉S ∧ RfS(S) ∧ X[I]=S}                     by assignment axiom
{∅∉S ∧ RfS(S) ∧ X[I]=S}                     loop invariant
while (S≠∅) do                              // main loop
 {∅∉S ∧ RfS(S) ∧ X[I]=S ∧ S≠∅}              by while rule
 {∅∉S ∧ RfS(S) ∧ X[I]=S}
 let B ∈ MinC(S);
 {∅∉S ∧ RfS(S) ∧ X[I]=S ∧ B∈MinC(S)}        by assignment axiom
 {∅∉S[L(B)]   ∧   RfS(S[L(B)])   ∧          by Lemmas 4.1 - 4.3
X[I∪L(B)]=S[L(B)]}
 S:=S[L(B)];
 {∅∉S ∧ RfS(S) ∧ X[I∪L(B)]=S}               by assignment axiom
 {∅∉S ∧ RfS(S) ∧ X[I∪L(B)]=S}
 I:=I∪L(B);
 {∅∉S ∧ RfS(S) ∧ X[I]=S}                    by assignment axiom
 {∅∉S ∧ RfS(S) ∧ X[I]=S}
od
{∅∉S ∧ RfS(S) ∧ X[I]=S ∧ S=∅}               by while rule
{X[I]=∅}
Z:=I;
{X[Z]=∅}                                    by assignment axiom
{M(X,Z)}                                    postcondition
HALT.
```

Now we see that from the precondition (the formula is resolution-free and does not contain the empty clause) the postcondition follows (the computed assignment is a model for the formula) if we follow the steps of the algorithm. The only question is whether the algorithm stops always or sometime runs for forever. We know from Lemma 4.4 that it stops always.

Theorem 5.2 (Correctness of Unicorn-SAT)

Let S be a resolution-free formula that does not contain the empty clause. Then after execution of Unicorn-SAT(S, I), I is a model for S.

Proof:
From Algorithm 5.1 and from Lemma 4.4 we can obtain that Unicorn-SAT stops for every resolution-free formula that does not contain the empty clause and gives back a model for it. ❑

Now we can prove that Unicorn-SAT is a linear time algorithm.

Theorem 5.3 (Complexity of Unicorn-SAT)
Let S be a resolution-free formula that does not contain the empty clause. Let *n* be the number of variables occurring in S and *m* the number of clauses in S. Then the time complexity of Unicorn-SAT(S, I) is O(*nm*), i.e., it is linear in length of S.

Proof:
The most expensive computation in an iteration is the hyper-unit propagation. Let us assume we need *k* iterations of the main loop to find a model for S. Since we eliminate at least 1 variable in each iteration we know that $k \leq n$. From definition of hyper-unit propagation we know that at the i^{th} iteration the number of variables of S decreases by t_i, where t_i is the number of variable of the sub-model at this iteration. Therefore, $\sum_{i=1..k} t_i = n$.
Hence, the *k* hyper-unit propagation can be simulated by *n* unit propagation. Unit propagation is an O(*m*) time method [10, 14]. Hence, time complexity of Unicorn-SAT(S, I) is O(*nm*). ❑

6 Example for Unicorn-SAT

We show how the Unicorn-SAT algorithm works for an example. In this example we use the convention that we obtain L(C) by negating all literals of C except the first one according to some variable ordering. We assume that variables are sorted by alphabetic order.

Example 6.1
Let X := $\{\{a, b, c\}, \{b, c, d, \overline{e}\}, \{a, \overline{b}, \overline{c}, d\}, \{c, \overline{d}, e\}\}$. X is resolution-free formula and does not contain the empty clause, so Unicorn-SAT can be applied to it. Step of Unicorn-SAT and values of variables:

After initialization (S, I) = $(\{\{a, b, c\}, \{b, c, d, \overline{e}\}, \{a, \overline{b}, \overline{c}, d\}, \{c, \overline{d}, e\}\}, \emptyset)$.

After first iteration (S, I, B) = $(\{\{d, \overline{e}\}, \{\overline{d}, e\}\}, \{a, \overline{b}, \overline{c}\}, \{a, b, c\})$.

After second iteration (S, I, B) = $(\emptyset, \{a, \overline{b}, \overline{c}, d, e\}, \{d, \overline{e}\})$.

The model Z for X is $\{a, \overline{b}, \overline{c}, d, e\}$.

7 Definitions for General Unicorn-SAT

We need some more definition to be able to generalize the Unicorn-SAT algorithm.

Definition 7.1 (Set of All n-Clauses)
CC (read: “set of all n-clauses”) := {C | C is a clause ∧ |C| = n}, where n is the number of propositional variables.

Note that n is a constant, the number of the propositional variables, i.e., an n-clause contains a literal for each propositional variable.

Definition 7.2 (Subsumed)

A $\supseteq\in$ S (read: “S subsumes A” or “A is subsumed by S”) $:\Leftrightarrow \exists B\; B \in S \wedge B \subseteq A$, where S is a formula and A is a clause.

Note that in Section 2 we defined notion of subsume for clauses. This definition is for formulae and clauses.

The notation $\supseteq\in$ comes from the following idea. The right-hand side of the definition can be reformulated as $\exists B\; A \supseteq B \wedge B \in S$. If we see this as a string and delete the "not interesting" parts we get $A \supseteq\in S$.

Definition 7.3 (Set of n-Clauses of a Formula)

$\{\{S\}\}$ (read: “the set of n-clauses of S”) $:= \{C \mid C \in CC \wedge C \supseteq\in S\}$, where S is a formula.

Note that $\{\{\ \{\varnothing\}\ \}\} = CC$.

Definition 7.4 (Formula Equivalence)

$S \equiv T$ (read: “S and T are equivalent” or “S and T subsumes the same set of n-clauses” or “S and T have the same set of models”)

$:\Leftrightarrow \{\{S\}\} = \{\{T\}\}$, where S and T are formulae.

The formula $S \equiv \{\varnothing\}$ means that S is unsatisfiable. It is so because S equivalent to a formula which contains the empty clause.

8 Properties of Formula Equivalence

In this section we will see properties of notion defined in Section 7. The main theorem in this section is the Expansion Rule, which state that if S[C] is unsatisfiable then we can "expand" the formula by adding the negation of the clause C to the formula. We can do this because the (original) formula and the “expanded” one have the same set of models.
In this section we will use notations more heavily.

Lemma 8.1 (Trivial Properties)

(a) $\varnothing \in S \Rightarrow S \equiv \{\varnothing\}$,

(b) $\{\varnothing\} \equiv CC$,

(c) $\{\varnothing\}[B] \equiv \{\varnothing\}$,

(d) $S \equiv T \Leftrightarrow T \equiv S$,

where S and T are formulae and B is an assignment.

Proof:

We do not give proofs for these trivial properties.

These properties are trivial. Point (a) states that if a formula contains the empty clause then it is unsatisfiable. Point (b) states that an unsatisfiable formula subsumes every n-clause. Point (c) states that hyper-unit propagation preserves unsatisfiability. Point (d) states that formula equivalence is commutative.

Theorem 8.2 (Clear Clause Rule or **n-Clause Rule)**

(a) $\overline{C} \supseteq\in S \Leftrightarrow S[C] \equiv \{\varnothing\}$,

(b) $\neg(\overline{C} \supseteq\in S) \Leftrightarrow S[C] = \varnothing$,

where S is a formula and $C \in CC$.

Proof:

Assume $C \in CC$.

(a) We show $\overline{C} \supseteq\in S \Leftrightarrow S[C] \equiv \{\varnothing\}$.

($\Rightarrow$) Assume $\overline{C} \supseteq\in$ S we show $S[C] \equiv \{\emptyset\}$. From $\overline{C} \supseteq\in$ S we know that there exists a $B \in S$ such that $B \subseteq \overline{C}$. From this and from $C \in CC$, by definition of hyper-unit propagation, we obtain that $\emptyset \in S[C]$. Hence, $\overline{C} \supseteq\in S \Rightarrow S[C] \equiv \{\emptyset\}$.
($\Leftarrow$) Assume $S[C] \equiv \{\emptyset\}$ we show $\overline{C} \supseteq\in$ S. From $S[C] \equiv \{\emptyset\}$ we know that $C \in \{\{S[C]\}\}$. From this we know that there exist a $B \in S$ such that $B \cap C = \emptyset$. Since $C \in CC$, $B \subseteq \overline{C}$. From $B \in S$ and $B \subseteq \overline{C}$ we obtain $\overline{C} \supseteq\in$ S. Hence, $\overline{C} \supseteq\in S \Leftarrow S[C] \equiv \{\emptyset\}$.
Hence, $\overline{C} \supseteq\in S \Leftrightarrow S[C] \equiv \{\emptyset\}$.
(b) We show $\neg(\overline{C} \supseteq\in S) \Leftrightarrow S[C] = \emptyset$.
($\Rightarrow$) Assume $\neg(\overline{C} \supseteq\in S)$ we show $S[C] = \emptyset$. To show this it suffices to show that for all $B \in S$ we have $B \cap C \neq \emptyset$. Let $B \in S$ arbitrary but fixed. We show that $B \cap C \neq \emptyset$. From the assumption $\neg(\overline{C} \supseteq\in S)$ we know that $\neg(B \subseteq \overline{C})$. From this, since $C \in CC$, we know that $B \subseteq C$. Therefore, $B \cap C \neq \emptyset$. Hence, $\neg(\overline{C} \supseteq\in S) \Rightarrow S[C] = \emptyset$.
($\Leftarrow$) Assume $S[C] = \emptyset$ we show $\neg(\overline{C} \supseteq\in S)$. Let $B \in S$ arbitrary but fixed. Now it suffices to show that $\neg(B \subseteq \overline{C})$. From the assumption $S[C] = \emptyset$ we know that $B \cap C \neq \emptyset$. Now we can use the same argument as in the previous case. Hence, $\neg(\overline{C} \supseteq\in S) \Leftarrow S[C] = \emptyset$.
Hence, $\neg(\overline{C} \supseteq\in S) \Leftrightarrow S[C] = \emptyset$. ❑

Clear Clause Rule (we prefer this name) states that an n-clause, say C, is a model for a formula, say S, if its negation, $\overline{C}$, is not subsumed by the formula, S. It is clear that if the negation of the n-clause, $\overline{C}$, is not subsumed by the formula, S, then hyper-unit propagation by it, S[C], results in a formula which does not contain the empty clause, i.e., may be satisfiable. Since C is n-clause and S[C] does not contain the empty clause, S[C] is empty, i.e., C is a model for S.

This lemma is a preliminary result, which will help us to proof the Expansion Rule.

Theorem 8.3 (Hyper-Unit Propagation Preserves Equivalence)
$S \equiv T \Rightarrow S[C] \equiv T[C]$, where S and T are formula.

Proof:
Assume $\{\{S\}\} = \{\{T\}\}$ we show $\{\{S[C]\}\} = \{\{T[C]\}\}$.
($\subseteq$) Let A be an arbitrary but fixed n-clause. Then we assume $A \in \{\{S[C]\}\}$ and show $A \in \{\{T[C]\}\}$. To show this, by definition of subsumed, we have to find a clause E such that $E \in T[C]$ and a subset of A. From $A \in \{\{S[C]\}\}$ we know that there is a clause $B \in S$ such that $B \cap C = \emptyset$ and $B \setminus \overline{C}$ is a subset of A. Then we know that $A \setminus C \cup \overline{C}$ is an n-clause and a superset of B. Then from the assumption $\{\{S\}\} \subseteq \{\{T\}\}$ we know that there exists a clause $D \in T$ such that D is a subset of $A \setminus C \cup \overline{C}$. From this we know that $D \cap C = \emptyset$. Therefore, $D \setminus \overline{C} \in T[C]$ and $D \setminus \overline{C}$ is a subset of A. Hence, $D \setminus \overline{C}$ is a suitable choice for E. Hence, $\{\{S[C]\}\} \subseteq \{\{T[C]\}\}$.
($\supseteq$) Analogously.
Hence, $S[C] \equiv T[C]$. ❑

Lemma 8.4

$S[A] \equiv \{\emptyset\} \wedge A \subseteq C \Rightarrow S[C] \equiv \{\emptyset\}$, where S is a formula and A and C are assignments.

Proof:

Let $B = C \setminus A$. Then $A \cap B = \emptyset$ and $C = A \cup B$. From this, by Lemma 4.3, we know that $S[C] = S[A][B]$. Since $S[A]$ is unsatisfiable, by Lemma 8.1.c, $S[A][B]$ is unsatisfiable.

Hence, $S[A] \equiv \{\emptyset\} \Rightarrow A \subseteq C \wedge S[C] \equiv \{\emptyset\}$. ❑

Theorem 8.5 (Expansion Rule)

$S[C] \equiv \{\emptyset\} \Leftrightarrow (S \cup \{\overline{C}\}) \equiv S$, where S is a formula and C is an assignment.

Proof:

($\Rightarrow$) If S[C] is unsatisfiable then, by Lemma 8.4 and Theorem 8.2, S subsumes all n-clauses being superset of $\overline{C}$. Therefore $S \cup \{\overline{C}\}$ and S have the same set of models.

($\Leftarrow$) If $S \cup \{\overline{C}\}$ and S have the same set of models then, by Lemma 8.3, $(S \cup \{\overline{C}\})[C]$ and S[C] have the same set of models. We know that $(S \cup \{\overline{C}\})[C]$, by definition of hyper-unit propagation, is unsatisfiable. Therefor, S[C] is unsatisfiable.

Hence, $S[C] \equiv \{\emptyset\} \Leftrightarrow (S \cup \{\overline{C}\}) \equiv S$. ❑

Expansion Rule is our main statement. It states that if an assignment, say C, makes a formula, say S, unsatisfiable then we can "expand" the formula by its negation, $\overline{C}$. We can do so because the resulting formula, $S \cup \{\overline{C}\}$, is equivalent to the original one, S. It is so because the negation of every n-clause subsumed by the assignment is subsumed by the original formula, see Clear Clause Rule. Thus, intuitively speaking, if we add $\overline{C}$ to S then we add no new "information" to it.

Now we should find a clever way how to use this rule. We will show that a clever way is to do hyper-unit propagation by a negation of a resolution-mate, because if the hyper-unit propagation results in an unsatisfiable formula then we can expand our original formula by a resolution-mate, i.e., a resolution can take place. Of course there may be other, even better, ways of using Expansion Rule.

Lemma 8.6

$C \in S \wedge L(\overline{C}) \in S \Rightarrow (S \cup \{Res(C, L(\overline{C}))\}) \equiv S$, where S is a formula.

Proof:

It is a basic property of resolution. Lemma 3.2 makes sure that resolution can be performed on C and $L(\overline{C})$. ❑

Lemma 8.7 (The Crucial Step in General-Unicorn-SAT)

$C \in S \wedge S[L(C)] \equiv \{\emptyset\} \Rightarrow (S \cup \{Res(C, L(\overline{C}))\}) \equiv S$, where S is a formula.

Proof:

It is an easy consequence of Theorem 8.5 and Lemma 8.6. ❑

Intuitively General Unicorn-SAT is the following algorithm. We do recursively hyper-unit propagation by a sub-model (the negation of a resolution-mate of a minimal clause) until either we reach the empty formula or a formula, which contains the empty clause. In the first case we are done we found a model, which the union of intermediate sub-models. In the second case we have to backtrack. It is the point where all the pre-request of the crucial step is met. The empty set is not in S (otherwise backtrack would take place on this level), C is a (minimal) clause from S and the empty clause is in S[L(C)] (we just do backtrack because of this fact). Now we know that we can expand S by $L(\overline{C})$ and do resolution (this is the crucial step) because on C and $L(\overline{C})$ we can always perform resolution. Now the resolvent will be the minimal clause of the formula because C was minimal. If the resolvent is empty we do backtrack, otherwise we do hyper-unit propagation by the sub-model generated from the new minimal clause.

9 The General Unicorn SAT Algorithm

Now we introduce the General Unicorn-SAT algorithm and we show that it solves general SAT problem in non-polynomial time.

Let MinC(S) be the set of minimal clauses of S. Let M(S, I) be the predicate which is true if I is a model for S, false otherwise.

We use "{}" to mark formulae that are true at the respective points of algorithm, in order to prove the correctness of the algorithm in the Hoare calculus.

In Hoare calculus formulae we will use the following invariants and an auxiliary formula.

$\mathrm{Aux(i)} :\Leftrightarrow S_i\neq\emptyset \wedge B_i\in \mathrm{MinC}(S_i)$.

$\mathrm{Inv1(i)} :\Leftrightarrow i\geq 0 \wedge S\neq\emptyset \wedge S_0=S \wedge B_0=\emptyset \wedge S_0\neq\emptyset \wedge$
$\quad$ for all $0\leq j\leq i$ $(S_{j+1}\equiv S[L(B_0)\cup...\cup L(B_j)]) \wedge$
$\quad$ for all $1\leq j\leq i$ $(\mathrm{Aux(j)} \wedge B_j\neq\emptyset)$.

$\mathrm{Inv2(i)} :\Leftrightarrow \mathrm{Inv1(i)} \wedge \mathrm{Aux(i+1)}$.

Algorithm 9.1 (General Unicorn-SAT):

```
General-Unicorn-SAT(S, M)
input: formula S that is not empty,
output: clause M, a model of S if exists, ∅ otherwise.
START
{S≠∅}                                             precondition
{S≠∅}
i:=0, B_0:=∅, S_0:=S, S_1:=S;
{S≠∅ ∧ i=0 ∧ B_0=0 ∧ S_0=S ∧ S_1=S}               by assignment axiom
{Inv1(i)}                                         1. loop invariant
while (Si+1 ≠ ∅) do
   {Inv1(i) ∧ S_i+1≠∅}                            by while rule
   {Inv1(i) ∧ S_i+1≠∅}
   i:=i+1, let B_i ∈ MinC(S_i);
   {i>0 ∧ Inv1(i-1) ∧ Aux(i)}                     by assignment axiom
   {i>0 ∧ Inv2(i-1)}                              2. loop invariant
   while (B_i = ∅) do
     {i>0 ∧ Inv2(i-1) ∧ B_i=∅}                    by while rule
     {i>0 ∧ Inv2(i-1) ∧ B_i=∅}
     i:=i-1;
     {i≥0 ∧ Inv2(i) ∧ B_i+1=∅}                    by assignment axiom
     {i≥0 ∧ Inv2(i) ∧ B_i+1=∅}
     if i=0 then
          {i=0 ∧ Inv2(0) ∧ B_1=∅}                 by if then rule
          {S≡{∅}}                                 by Lemma 8.1.a
          M:=∅;
          {S≡{∅} ∧ M=∅}                           by assignment axiom
          {(S[M]=∅) ∨ (S≡{∅} ∧ M=∅)}              postcondition
```

```
        HALT.
    fi
    {i>0 ∧ Inv2(i)}                          by if else rule and by logical
                                             consequence rule
    {i>0 ∧ Inv2(i-1) ∧ B_i≠∅}                by logical consequence rule
    B_i:=Res(B_i,L(B̄_i));
    S_i:=S_i ∪ B_i;
    {i>0 ∧ Inv2(i-1)}                        by assignment axiom and by
                                             Lemma 8.7
    {i>0 ∧ Inv2(i-1)}
  od
  {i>0 ∧ Inv2(i-1) ∧ B_i≠∅}                  by while rule
  {i>0 ∧ Inv1(i-1) ∧ Aux(i) ∧ B_i≠∅}
  S_i+1:=S_i[L(B_i)];
  {i>0 ∧ Inv1(i)}                            by assignment axiom and by
                                             Lemma 4.3
  {i>0 ∧ Inv1(i)}
od
{Inv1(i) ∧ S_i+1=∅}                          by while rule
{Inv1(i) ∧ S_i+1=∅}
M:=L(B_0)∪...∪L(B_i);
{S[M]=∅}                                     by assignment axiom and by
                                             Lemma 4.3
{(S[M]=∅) ∨ (S≡{∅} ∧ M=∅)}                   postcondition
HALT.
```

Now we see that from the precondition (the formula is not empty) the postcondition (the computed assignment is a model for the formula if it is satisfiable, otherwise the empty clause) follows if we follow the steps of the algorithm. The only question is whether the algorithm stops always or sometime runs for forever. We know from Lemma 4.4 and from a basic property of resolution, namely it simplifies the problem, that it stops always.

Theorem 9.2 (General Unicorn-SAT is Complete)

Let S be a non-empty formula. Then after execution of General Unicorn-SAT(S, M), M is a model for S if it is satisfiable, otherwise it is the empty clause.

Proof:

From Algorithm 9.1, from Lemma 4.4 and from the fact that resolution simplifies the problem we can obtain that General Unicorn-SAT stops for every formula and gives back a model for it if the formula satisfiable otherwise gives back the empty clause. ❑

Now we can prove that General Unicorn-SAT is a non-polynomial time algorithm.

Theorem 9.3 (Complexity of General Unicorn-SAT)

Let S be a non-empty formula. Let n be the number of variables occurring in S and m the number of clauses in S. Then the time complexity of General Unicorn-SAT(S, M) is $O(m2^n)$.

Proof:

The complexity of General Unicorn-SAT depends on how many literals are in the formula. Therefore, the worst case is if the input formula is CC. We assume that there is an ordering on the variables of the input formula and $V = \{v_1, ..., v_n\}$ is the set of variables. Let us consider the set CCk := {C | C is a clause $\wedge$ every variables occur in C from {v_1, ..., v_k}}. Note that CC = CCn. We show, by induction on k, that General Unicorn-SAT perform 2^n-1 hyper-unit propagations to determine that CC is unsatisfiable. For k = 1 it is straightforward, we need 1 hyper-unit propagation to show that CC1= is unsatisfiable. Assume that for CCi, i ≤ k, we need 2^i-1 hyper-unit propagations to show that this is unsatisfiable. We show that for CC$k+1$ we need 2^{k+1}-1 hyper-unit propagations to show that this is unsatisfiable. We assume that negation of a resolution-mate of a clause always negates the first literal of the clause. Therefore, in the crucial step, see Lemma 3.7, the resolution deletes the first literal of the sub-model (a negation of a resolution-mate of a minimal clause). We have to count how many hyper-unit propagations are needed to delete all the literals from the sub-model. To delete the i-th literal, such that we assume that the $1..i$-1-th literals are already deleted, General Unicorn-SAT has to detect that CC$k+1$-i is unsatisfiable which needs, by the induction hypothesis, 2^{k+1-i}-1 hyper-unit propagations. Therefore, to delete every literal from the sub-model, which has $k+1$ literals, we need

$$\left(\sum_{i=1..k} 2^{k+1-i} - 1\right) + k + 1 \qquad \textbf{(1)}$$

hyper-unit propagations. The formula (1) equals to 2^{k+1}-1. Hence, we need 2^n-1 hyper-unit propagations to determine that CC is unsatisfiable.
Since we perform 2^n-1 hyper-unit propagations and a hyper-unit propagation means in the worst case n unit propagations and since the unit propagation is an O(m) time method [2, 4], the time complexity of General Unicorn-SAT is O($mn2^n$-1). ❑
Example 10.3 shows how does the General Unicorn-SAT algorithm work on CC2.

It seems a very disappointing result but we should note that we did not use any simplification steps like unit propagation if S contains a unit. General Unicorn-SAT is compatible with the most simplification strategy and we should use them to get better complexity results in the future.

10 Example for General Unicorn-SAT

Let us first recall the General Unicorn-SAT without Hoare calculus formulae in order to be more readable.

Algorithm 10.1 (General Unicorn-SAT):

```
START
i:=0, B0:=∅, S0:=S, S1:=S;
while (Si+1 ≠ ∅) do
   i:=i+1;
   let Bi ∈ MinC(Si);
   while (Bi = ∅) do
     i:=i-1;
     if i = 0 then
          M:=∅;
          HALT.
     fi
     Bi:=Res(Bi,L(B̄i));
```

```
    Si:=Si ∪ Bi;
  od
  Si+1:=Si[L(Bi)];
od
M:=L(B0) ∪ ... ∪ L(Bi);
HALT.
```

The essence of the invariant is the following. The model for the formula is the union of intermediate sub-models, i.e., $M:=L(B_0) \cup ... \cup L(B_i)$, and we can generate the last formula by hyper-unit propagation by the model, i.e., $S_{i+1} = S[M]$. We try to focus of these by giving examples.

Example 10.2

Let S = {{a}, {b}}. It is easy to see that S is satisfiable M = {a, b} is a model for it. We give a table, which shows value of variables used in General Unicorn-SAT. The variables have these values before the second while except S_i = {} which means we found a model. By level we mean that if we would use the recursive algorithm on which recursion level would this step occur.

Let $M:=L(B_0) \cup ... \cup L(B_i)$, $S_{i+1} = S[M]$.

Level	I	S_I	B_i	$L(B_i)$	M
1	0	{{a}, {b}}	{}	{}	{}
1	1	{{a}, {b}}	{a}	{a}	{a}

Note that here we could also choose {b} for B_1 since both {a} and {b} are minimal clauses. Note also that a negation of a resolution-mate of a unit clause is the unit clause itself.

2	2	{{b}}	{b}	{b}	{a, b}
3	3	{}			

M is a model for S.

Example 10.3

For denoting literal negation we will use the notation ~v instead of the notation $\bar{v}$ in this example.

Let S = {{a, b}, {a, ~b}, {~a, b}, {~a, ~b}}. It is easy to see that S is unsatisfiable, actually S = CC. For more instruction see Example 10.2.

Let $M:=L(B_0) \cup ... \cup L(B_i)$, $S_{i+1} = S[M]$.

Level	I	S_I	B_i	$L(B_i)$	M
1	0	{{a, b}, {a, ~b}, {~a, b}, {~a, ~b}}	{}	{}	{}
1	1	{{a, b}, {a, ~b}, {~a, b}, {~a, ~b}}	{a, b}	{a, ~b}	{a, ~b}

Note that here we could choose any clause from S_1 since all of them are minimal clauses. We recall the first part of definition of negation of a resolution-mate. Negation of a resolution-mate of a clause C, denoted by L(C), is the negation of of a resolution-mate of C. In this example we use the convention that we obtain L(C) by negating all literals of C except the first one (according to some variable ordering). In this example we use the ordering a < b. Therefore, $L(B_1)$ is {a, ~b}.

2	2	{{}}	{}	{}	{a, ~b}

We recall the second part of the definition of negation of a resolution-mate. By definition $L(\varnothing) = \varnothing$. Therefor $L(B_2)$ is {}. Now we fulfil the condition of the second while loop. So we enter it and do that many backtrack step as needed.

Note that $\varnothing \in S_2$ means that, by Theorem 8.5 (expansion rule), that we can add the negation of $L(B_1)$, which is {~a, b}, to S_1. Therefore S_1 is equivalent to {{~a, b}, {a, b}, ...}, where ... means the remaining part of S_1. So we can perform resolution and we get the new S_1. See next line.

1	1	{{b}, {a, b}, {a, ~b}, {~a, b}, {~a, ~b}}	{b}	{b}	{b}
2	2	{{a}, {~a}}	{a}	{a}	{a, b}
3	3	{{}}	{}	{}	{}

Now we enter the second while loop and we track back three time, i becomes 0. It means S is unsatisfiable.

1	0	{{a, b}, {a, ~b}, {~a, b}, {~a, ~b}}	{}	{}	{}

Since S is unsatisfiable M is the empty clause.

11 Conclusion and Future Work

Resolution-free SAT problem is a new type of linear time SAT problems. On resolution-free SAT problems we should try to build up a hierarchy of polynomial time problems like tractable satisfiability problems [10] which is based on Horn SAT and 2-SAT and an instance of the hierarchy on the k level is solvable in $O(n^{k+1})$ time.

The General Unicorn-SAT algorithm is complete for the general SAT problem like the Davis and Putnam method [13]. We should rigorously study which one is better for which set of SAT instances. For example for the resolution-free SAT problem General Unicorn-SAT is better since it solves it in linear time. It seems that the search tree of General Unicorn-SAT has less (or equal) nodes than the search tree of Davis and Putnam method. But General Unicorn-SAT uses more unit propagation (if we consider hyper-unit propagation as unit propagation after each other) than Davis and Putnam method. We should study rigorously the time complexity of hyper-unit propagation.

Appendix A

This is the recursive version of the General Unicorn-SAT algorithm.

```
General-Unicorn-SAT(X1, X2)
input: formula X1 that does not contain the empty clause, X2
proposed sub-model,
output: clause Z, a model of X1[X2].
START
(S, I) := (X1[X2], X2);
if S = ∅ then return I;
if ∅ ∈ S then return ∅;
let B ∈ MinC(S);
while ((C := General-Unicorn-SAT(S, L(B))) = ∅) do
     B := Res(B,L(B̄));
     if B = ∅ then return ∅;
od
I := I ∪ C;
Z := I;
return Z;
HALT.
```

References

1. Stephen Cook. The complexity of theorem proving procedures. In *Proc. 3rd Ann. ACM Symp. on Theory of Computing*, pages 151--158, New York, 1971.
2. S. Even, A. Itai and A. Shamir. On the complexity of timetable and multi-commodity flow problems. *SIAM Journal on Computing*, 5(4), 1976.
3. Bengt Aspvall, Michael F. Plass and Robert Endre Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulae. *Information Processing Letters*, 8(3):121--123, March 1979.
4. W.F. Dowling and J.H. Gallier. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *Journal of Logic Programming*, 1(3):267--284, 1984.
5. Maria Grazia Scutella. A note on Dowling and Gallier's top-down algorithm for propositional Horn satisfiability. *Journal of Logic Programming*, 8(3):265--273, May 1990.
6. Bengt Aspvall. Recognizing disguised NR(1) instances of the satisfiability problem. *Journal of Algorithms*, 1:97--103, 1980.
7. Vijaya Chandru and John Hooker. Extended Horn sets in propositional logic. Journal of the ACM, 38(1):205--221, 1991.
8. E. Boros, P. L. Hammer, and X. Sun. Recognition of q-Horn formulae in linear time. *Discrete Applied Mathematics*, 55:1--13, 1994.
9. E. Boros, Y. Crama, P. L. Hammer, and M. Saks. A complexity index for satisfiability problems. *SIAM Journal on Computing*, 23:45--49, 1994.
10. Mukesh Dalal and David W. Etherington. A hierarchy of tractable satisfiability problems. *Information Processing Letters*, 44:173--180, 1992.
11. Donald E. Knuth. Nested satisfiability. *Acta Informatica*, 28:1--6, 1990.
12. Craig A. Tovey. A simplified NP-complete satisfiability problem. *Discrete Applied Mathematics*, 8:85--89, 1984.
13. M. Davis and H. Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7:201--215, 1960.

14. H. Zhang and M. E. Stickel. An efficient algorithm for unit propagation. *Fourth International Symposium on Artificial Intelligence and Mathematics*, 166--169, 1996.