Finding Models for Blocked 3-SAT Problems in Linear Time by Systematical Refinement of a Sub-model

Gábor Kusper

Eszterházy Károly College
Department of Information Technology
1. sqr. Eszterházy , Eger 3300 Hungary
gkusper@sztech.ektf.hu
http://sztech.ektf.hu/~gkusper

Abstract. We report a polynomial time SAT problem instance, the Blocked SAT problem. A blocked clause set, an instance of the Blocked SAT problem, contains only blocked clauses. A close is blocked (for resolution) if it has a literal on which no resolution is possible in the clause set. We know from work of O. Kullmann that a blocked clause can be added or deleted from a clause set without changing its satisfiability. Hence, any blocked clause set is satisfiable, but it is not clear how to find a satisfying assignment for it. We introduce the Blocked SAT Solver algorithm, which provides a model for Blocked SAT problems in linear time, if we know at least one blocked literal per clause. To collect these information polynomial time is needed in general. We show that in case of 3-SAT we can collect these information in linear time. This means that the Blocked 3-SAT problem is a linear time problem. We also discuss how to use blocked clauses if the whole clause set is not blocked.

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 [Coo71]. 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 that there are restricted SAT problems that can be solved in polynomial time. So a "good" SAT algorithm should check the input SAT instance first whether it is an instance of such a restricted SAT problem. In this work we introduce the Blocked SAT problem, which is solvable in polynomial time. We list some polynomial time solvable restricted SAT problems:

C. Freksa, M. Kohlhase, and K. Schill (Eds.): KI 2006, LNAI 4314, pp. 128–142, 2007.
© Springer-Verlag Berlin Heidelberg 2007

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 [EIS76, APT79].

2. *Horn SAT* is the restriction to instances where each clause has at most one un-negated variable. Horn SAT is solvable in linear time [DG84, Scu90], as are a number of generalizations such as *renamable Horn SAT* [Lew78, Asp80], *extended Horn SAT* [CH91] and *q-Horn SAT* [BHS94, BCH+94].

3. The hierarchy of *tractable* satisfiability problems [DE92], 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 $O(nk + 1)$ 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 [Knu90].

5. SAT in which no variable appears more than twice. All such problems are satisfiable if they contain no unit clauses [Tov84].

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 [Tov84].

7. A formula is *SLUR* (Single Lookahead Unit Resolution) *solvable* if, for all possible sequences of selected variables, algorithm SLUR does not give up. Algorithm SLUR is a nondeterministic algorithm based on unit propagation. It eventually gives up the search if it starts with, or creates, an unsatisfiable formula with no unit clauses. The class of SLUR solvable formulae was developed as a generalization including Horn SAT, renamable Horn SAT, extended Horn SAT, and the class of CC-balanced formulae [SAF+95].

8. *Resolution-Free* SAT Problem, where every resolution results in a tautologous clause, is solvable in linear time [Kus05].

The Blocked SAT problem is also a restriction of SAT to instances where each clause of the clause set is blocked, i.e., we have at least one blocked literal in each clause. It is a generalization of the Resolution-Free SAT problem, where all literals are blocked.

A clause is blocked in a clause set if it has a literal on which no resolution is possible in that clause set, i.e., it is blocked for resolution. The notion of blocked clause was introduced by O. Kullmann in [Kul99a, Kul99b]. He studied blocked clauses because we wanted to add new short clauses to the input clause set to improve the worst case time complexity of his algorithm. He proved that a blocked clause can be added or deleted from a clause set without changing its satisfiability.

Based on that idea it is easy to show that any blocked clause set is satisfiable, as if we remove all blocked clauses then we obtain the trivially satisfiable clause set. But this process do not show how to find a model for blocked clause sets.

The Blocked SAT Solver algorithm provides a model for Blocked SAT problems in linear time, if we know at least one blocked literal per clauses. It uses heavily the notion of sub-model [Kus02, Kus05].

A sub-model is a partial assignment created, by definition, by the negation of a resolution-mate. A resolution-mate is obtained from a clause, the generator clause, by negating one of its literals, the generator literal.

The algorithm exploits the fact that if a sub-model is generated form a blocked clause (generator clause) using one of its blocked literal as the generator one (generator literal), then it is a model for those clauses in the set which contain either positively or negatively the generator literal. Moreover, it satisfies from the rest the ones which differ from it. (The clause A differs from the clause B if it has a literal a, i.e., $a \in A$, which occurs in B negatively, i.e., $\overline{a} \in B$.) Which means it satisfies almost the whole clause set! Only those clauses are not satisfied which do not contain the generator literal neither positively nor negatively and do not differ from the generator clause. Here comes the trick. If we have such a clause (no-occurrence clause), then the union of it and the generator clause is a clause. This union will be our new generator clause. It contains an unused blocked liter. This literal will our next generator literal (keeping also the older ones). We generate a new sub-model from the generator clause and generator literals which will satisfy all the clauses which was satisfied by the old one and which satisfies also many more clauses. This process is called as the refinement of the sub-model. After finitely many refinement steps the sub-model will satisfy the whole clause set.

It is not necessary that from the beginning all clauses are blocked, but only that the (confluent) reduction process of eliminating blocked clauses finally eliminates all clauses.

We also discuss how to collect the information whether a literal is blocked or not. We show that in case of 3-SAT we can collect these information in linear time by using cubic memory space.

We introduce the data structure called NLC, Number of Literal Combinations. We create NLC by reading each clause only once. For every subset of every clause we increase the corresponding counter in NLC by one. Afterwards we read again the clause set and for every literal in every clause we calculate the number of possible resolution partners minus the number of blocking clauses. For example if the clause is $\{a, b, c\}$ and the literal is a than this number is

$$NLC[\overline{a}] - NLC[\overline{a}, \overline{b}] - NLC[\overline{a}, \overline{c}] + NLC[\overline{a}, \overline{b}, \overline{c}].$$

If this number is zero then this literal is blocked in the input clause set.

In case of 3-SAT we need cubic memory space to store NLC. We read and write it 7 times (3 times for one length subsets of the clause, 3 times for two length subsets of the clause and 1 time for the clause itself) per clauses in the first loop. In the second loop we read it 4 times per literal (see the example above). Both steps are linear in the number of literals. This means, since the Blocked SAT Solver is a linear time method if we know at least one blocked literal per clauses, that the Blocked 3-SAT problem is a linear time problem.

An input clause set is rarely blocked but during the work of a general SAT solver algorithm we may encounter a blocked clause set. Any general SAT solver uses some simplification steps: resolution, unit-propagation, removing subsumed

clauses, etc. The fewer clauses (or literals) are in a clause set the more likely that it is blocked. This means that soon or later a general SAT solver encounters a blocked clause set. In this case it is worth switching to the Blocked SAT Solver algorithm since it is polynomial. The simplification steps may update the NLC data structure which makes it easier to decide whether an immediate clause set is blocked or not.

We also discuss how to use blocked clauses if the whole clause set is not blocked. We introduce two lemmas, the Blocked Clear Clause Rule and the Independent Blocked Clause Rule, to describe cases where the answer is true.

2 Definitions

We use the well known set based representation of SAT. The abbreviation "iff" means "if and only if".

Let V be a finite *set of Boolean variables*. The *negation of a variable* v is denoted by $\overline{v}$. Given a set U, we denote $\overline{U} := \{\overline{u} \mid u \in U\}$ and call the *negation of the set* U.

Literals are 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 $\overline{v}$, then $\overline{w}$ denotes the positive literal v.

Clauses and *assignments* are finite sets of literals that do not contain any literal together with its negation simultaneously. A clause is interpreted as a disjunction of its literals. An assignment is interpreted as a conjunction of its literals. A *clause set* is a finite set of clauses. A clause set is interpreted as a conjunction of its clauses.

We use the following constants: n is the number of variable, and m is the number of clauses in the input clause set.

If C is a clause and $|C| = k$, then we say that C is a *k-clause*. Special cases are *unit clauses* or *units* which are 1-clauses, and *clear* or *total clauses* which are n-clauses. Note that any unit clause is a clause and an assignment at the same time. The clause set CC is the set of all clear clauses.

$$CC := \{C \mid \mathit{Clause}(C) \wedge |C| = n\}.$$

A clause C is *subsumed by the clause set* S, denoted by $C \supseteq\in S$, iff

$$C \supseteq\in S :\iff \mathit{Clause}(C) \wedge \mathit{ClauseSet}(S) \wedge \exists_{B \in S} B \subseteq C.$$

A clause C is *entailed by the clause set* S, denoted by $C \supseteq\in_{CC} S$, iff

$$C \supseteq\in_{CC} S :\iff \mathit{Clause}(C) \wedge \mathit{ClauseSet}(S) \wedge \forall_{\substack{D \in CC \\ C \subseteq D}} \exists_{B \in S} B \subseteq D.$$

A clause C is *independent in clause set* S iff it is not entailed by S.

If A and B are clauses then we define the *clause difference* of them, denoted by $\mathit{diff}(A, B)$, as

$$\mathit{diff}(A, B) := A \cap \overline{B}.$$

If $diff(A,B) \neq \emptyset$ then we say that A *differs from* B.

Resolution can be performed on two clauses iff they differ only in one variable. If resolution can be performed then the *resolvent*, denoted by $Res(A,B)$, is

$$Res(A,B) := (A \cup B) \setminus (diff(A,B) \cup diff(B,A)).$$

If S is a clause set and A is an assignment, then we can do *hyper-unit propagation*, for short *HUP*, by A on S, denoted by $HUP(S,A)$, as follows:

$$HUP(S,A) := \{C \setminus \{\overline{A}\} \mid C \in S \wedge C \cap A = \emptyset\}.$$

A *literal* $c \in C$ is *blocked* in the clause C and in the clause set S, denoted by $Blck(c,C,S)$, iff

$$Blck(c,C,S) : \Longleftrightarrow \underset{\substack{B\in S\\ \overline{c}\in B}}{\forall} \underset{\substack{b\in B\\ b\neq\overline{c}}}{\exists} \overline{b} \in C,$$

A *clause* C is *blocked* in the clause set S, denoted by $Blck(C,S)$, iff a blocked literal exists in it. A *clause set* S is *blocked*, denoted by $Blck(S)$, iff all clauses in it are blocked.

If C is a clause and A is an assignment then we say that the sub-model generated by C and A, denoted by $sm(C,A)$, is

$$sm(C,A) = \begin{cases} \overline{(C \setminus A)} \cup A & ,if C \neq \emptyset; \\ \emptyset & ,otherwise. \end{cases}$$

A partial assignment I is a *model for a clause set* S, denoted by $M(S,I)$, iff $HUP(S,I)$ is the empty clause set.

3 The Blocked SAT Problem

In this section we introduce the Blocked SAT problem which is a restriction of SAT to instances where each clause of the clause set is blocked.

A clause is blocked in a clause set if it has a literal on which no resolution is possible in that clause set. The notion of blocked clause was introduced in [Kul99a, Kul99b]. A blocked clause can be added or deleted from a clause set without changing its satisfiability. From this it is easy to show that any blocked clause set is satisfiable, as if we remove all blocked clauses then we obtain the trivially satisfiable clause set. But this process do not show how to find a model for blocked clause sets.

Now we recall the formal definition of the blocked clause set:

$$Blck(S) : \Longleftrightarrow \underset{A\in S}{\forall} \underset{a\in A}{\exists} \underset{\substack{B\in S\\ \overline{a}\in B}}{\forall} \underset{\substack{b\in B\\ b\neq\overline{a}}}{\exists} \overline{b} \in A.$$

Since the definition has 2 quantifiers on clauses and 2 quantifiers on literals, in these clauses we need $O(n^2m^2)$ time to decide whether the clause set is blocked.

We give three examples for blocked clause sets. Notation is explained below.

1st	2nd	3rd
(+)+ x	(-)+ x x	+ +(-)
x +(+)	x(+)+ x	+(-)+
(-)- x	x -(-)x	(+)- -
x -(-)	x x -(+)	(-)+ +
		-(+)-
		- -(+)
+ - +	- - + +	+ + +
- + -	x + - x	- - -

In the example we used the literal matrix representation of clause sets where rows corresponds to clauses and columns to variable. The symbol + means positive literal occurrence, - means negative literal occurrence and x means no literal occurrence. Under each example we list all models for it. Blocked literals are in brackets.

The Blocked SAT problem is a very restrictive subset of the general SAT problem, but it can be solved in polynomial time. During the work of a general SAT solver immediate blocked clause sets are frequent.

4 The Blocked SAT Solver Algorithm

In this section we introduce the blocked SAT Solver algorithm which solves the Blocked SAT problem in polynomial time. To be more precise we present a linear time version which uses $O(4m)$ basic set operations. But this version assumes that we already know about at least one blocked literal per clause. However, in general, polynomial ($O(n^2m^2)$) time is needed to collect that information.

Algorithm 1 (Blocked SAT Solver)
BlockedSATSolver(S, Z)
input: clause set S that is non-empty and blocked,
output: assignment Z, a model for S.

```
1  START
2    (A, B) = (∅, ∅);
3      //  A is the generator clause, B is the set of generator literals.
4    for each C ∈ S do
5        //  If C and A differ then sm(A, B) satisfies C, see Lemma 1.
6      if (diff(A, C) = ∅) then
7          //  Otherwise, C is a no-occurrence clause.
8          //  We have to consider it.
9        A := A ∪ C;
10         //  This is the new generator clause.
11       if (B ∩ C = ∅) then
```

```
12              // In this case we have to refine B
13              // to gain C ∩ sm(A, B) ≠ ∅.
14            Let c ∈ C be a blocked literal in C, S;
15            B := B ∪ {c};
16              // This is the new set of generator literals.
17          fi
18        fi
19      od
20      Z := sm(A, B);
21    HALT
```

The main idea of the algorithm is the following. If the input clause set is blocked then if we generate a sub-model from a blocked clause (generator clause, in the algorithm variable A) and from a blocked literal (generator literal(s), in the algorithm variable B) then we obtain a model for those clauses from the clause set which contain the generator literal or the negation of it. But there might be a clause (no-occurrence clause, in the algorithm variable C) which does not contain the generator literal either positively or negatively. This means that the union of the no-occurrence clause and the generator clause is a clause. This clause will be our new generator clause. Since the no-occurrence clause is blocked too, we can select a blocked literal from it and add to the generator literals. We do this only if it is necessary, i.e., if the sub-model does not satisfies the no-occurrence clause. The new sub-model is generated from the new generator clause and from the new generator literals. This step is the refinement step. One can show that this new sub-model is still a model for the clauses which are satisfied by the old one (see the 2nd auxiliary lemma). Hence, we obtain a model for the input clause set by performing refinement steps while we read its clauses one after the other.

It is not necessary that from the beginning all clauses are blocked, but only that the (confluent) reduction process of eliminating blocked clauses finally eliminates all clauses.

We see that this algorithm uses each clause only once and in an iteration it does 4 basic set operations in the worst case.

First we give an example on how the Blocked SAT Solver algorithm works. The variables S, C, A, c, B are the variables from the algorithm. Blocked literals are in brackets in columns S and C. The abbreviation "unch." means unchanged.

S	C	A	$\{c\}$	B	$sm(A,B)$
+(-)x x (+)x + x - x(-)x x x -(+)	+(-)x x	+-xx	x-xx	x-xx	--xx
unch.	(+)x + x	+-+x	+xxx	+-xx	+--x
unch.	- x(-)x	unch.		unch.	+--x
unch.	x x -(+)	unch.		unch.	+--x

The input clause set, S, is non-empty and blocked. So the precondition is fulfilled. We take the first clause and one of its blocked literal. We generate a sub-model from them. This will be the base of the refinement. We read the next clause, which is a no-occurrence clause. We refine the sub-model. We read the next clause, but it is satisfied, so we do not refine the sub-model. The same is true for the lest clause. The last sub-model as we expected is indeed a model for the input clause set.

The following lemmas are needed to show that the Blocked SAT Solver algorithm always finds a model for a blocked clause set. The first one states that if we have a blocked clause set S and its subset G which contains only clauses that do not differ form each other then there is an algorithm which constructs B that has $sm(\bigcup G, B)$ is a model for the clauses which differ from $\bigcup G$.

Lemma 1 (Auxiliary Lemma 1 for Blocked SAT Solver)
Assume S is a non-empty, blocked clause set. Assume $G \subseteq S$, G is non-empty and for all $E, F \in G$ we have $diff(E, F) = \emptyset$. Assume $A = \bigcup G$. Assume B is a clause constructed by the following piece of pseudo-code:

```
1   (B, i) := (∅, 1);
2   for each E ∈ G do
3     if (B ∩ E = ∅) then
4       Let a_i ∈ E be a blocked literal in E, S;
5       (B, i) := (B ∪ {a_i}, i + 1);
6     fi
7   od
```

Assume $C \in S$ and $diff(A, C) \neq \emptyset$. Then $C \cap sm(A, B) \neq \emptyset$.

Proof by Contradiction: Assume $C \cap sm(A, B) = \emptyset$. We show that this assumption leads to a contradiction. From this assumption we know, by definition of sub-model, that $diff(A, C) \subseteq B$. Let $k = |B|$. Note that we know, from the construction of B, that $B = \{a_1..a_k\}$. Let $i \in \{1..k\}$ be the largest index such that $a_i \in diff(A, C)$ and $a_i \in B$. (Such $i \in \{1..k\}$ exists, because $diff(A, C) \neq \emptyset$ and $diff(A, C) \subseteq B$.) Then we know, from the construction of B, that there is a clause $E \in G$ which has $(B \setminus \{a_i..a_k\}) \cap E = \emptyset$ and $a_i \in E$ and $a_i \in E$ is blocked in E, S. Since $a_i \in diff(A, C)$ we know, by definition of clause difference, that $\overline{a_i} \in C$. From this and from that $a_i \in E$ is blocked in E, S and from $C \in S$ we know, by definition of blocked literal, that for some $\overline{c} \in C$ we have $\overline{c} \neq \overline{a_i}$ and $c \in E$. From this and from $A = \bigcup G$ we know that $c \in A$. Hence, $c \in diff(A, C)$.

From $c \in E$ and from $(B \setminus \{a_i..a_k\}) \cap E = \emptyset$ we know that $c \notin \{a_1..a_{i-1}\}$. From $\overline{c} \neq \overline{a_i}$, i.e., from $c \neq a_i$ and from $c \in diff(A, C)$ and from the fact that $i \in \{1..k\}$ is the largest index such that $a_i \in diff(A, C)$ and $a_i \in B$ we know that $c \notin \{a_i..a_k\}$. Hence, $c \notin B$.

But this is a contradiction, because $c \in diff(A, C)$ and $diff(A, C)$ is a subset of B. Hence, $C \cap sm(A, B) \neq \emptyset$.

The main idea of this proof is that in the worst-case A and C differ only in blocked literals from B, i.e., the sub-model does not satisfy C. But the construction of B makes sure that this cannot happen.

This lemma makes sure that $sm(A, B)$ satisfies C provided that it is not a no-occurrence clause.

If we look at the Blocked SAT Solver algorithm we see that it does the same as the piece of the pseudo-code in this auxiliary lemma. The only difference is that in the lemma we already have the set of clauses which does not differ from each other (this is the clause set G), while in the algorithm we collect these clauses during the computation. In both cases A is the union of these clauses.

Remember, we enlarge A and B only if we encounter a no-occurrence clause C in order to refine the sub-model, which means that the new sub-model will satisfies C.

The second auxiliary lemma states that this new sub-model still satisfies those clauses which were satisfied by the old one.

Lemma 2 (Auxiliary Lemma 2 for Blocked SAT Solver). *Let S be a non-empty, blocked clause set. Let A be a clause. Let $B \subseteq A$. Let $C \in S$ such that $C \subseteq A$ and $B \cap C = \emptyset$. Let $c \in C$ be a blocked literal in C, S. Let $D \in S$ such that $D \cap sm(A, B) \neq \emptyset$. Then $D \cap sm(A, B \cup \{c\}) \neq \emptyset$.*

> Proof: If $\overline{c} \notin D$ then $D \cap sm(A, B \cup \{c\}) \neq \emptyset$ follows from $D \cap sm(A, B) \neq \emptyset$. Assume $\overline{c} \in D$. Then since $c \in C$ is blocked in C, S, by definition of blocked literal, we know that for some $d \in D$ we have $d \neq \overline{c}$ and $\overline{d} \in C$. Since $C \subseteq A$ we know that $\overline{d} \in A$ and from $B \cap C = \emptyset$ we know that $\overline{d} \notin B$. Therefore, by definition of sub-model, $\overline{d} \in sm(A, B)$. From $\overline{d} \neq c$ we know, by definition of sub-model, that $\overline{d} \in sm(A, B \cup \{c\})$. Hence, $D \cap sm(A, B \cup \{c\}) \neq \emptyset$.

Now we show that the Blocked SAT Solver algorithm solves the Blocked SAT problem. 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. The Hoare calculus makes sure that if we have a so-called "while program" and we can arrive from the precondition to the postcondition using the rules of the calculus, then for each input which fulfills the precondition the computed output fulfills the postcondition if the computation terminates. For more details about Hoare calculus please consult [LS87].

For better readability we use the infix version of hyper-unit propagation, which is denoted by an asterisk (*). This means that instead of $HUP(S, A)$ we use $S * A$. This variant of Blocked SAT Solver uses a new variable T in order to be able to give the invariant of the algorithm. We know that Block SAT Solver considers each clause in the input claus set. In T we collect the visited clauses. The essence of the invariant is that $sm(A, B)$ is a model of the visited clauses.

In Hoare calculus we will use the following invariant and auxiliary formulae:

$$Inv :\Leftrightarrow Blck(S) \wedge S \neq \emptyset \wedge T * sm(A, B) = \emptyset.$$

$$If1 :\Leftrightarrow Inv \wedge C \in (S \setminus T).$$

$$If2 :\Leftrightarrow If1 \wedge C \subseteq A.$$

$$IT2 :\Leftrightarrow If2 \wedge B \cap C = \emptyset.$$

Algorithm 2 (Blocked SAT Solver)
BlockedSATSolver(S, Z)
input: clause set S that is non-empty and blocked,
output: assignment Z, a model for S.

1 **START**
2 // $\{Blck(S) \wedge S \neq \emptyset\}$, *precondition*
3 // $\{Blck(S) \wedge S \neq \emptyset \wedge \emptyset * sm(\emptyset, \emptyset) = \emptyset\}$
4 $(A, B, T) := (\emptyset, \emptyset, \emptyset)$;
5 // $\{Blck(S) \wedge S \neq \emptyset \wedge T * sm(A, B) = \emptyset\}$, *by assignment axiom*
6 // $\{\{Inv\}$, *loop invariant*
7 **while** $(T \neq S)$ **do**
8 // $\{\{Inv \wedge T \neq S\}$, *by while rule*
9 // $\{\{Inv \wedge T \neq S\}$
10 **Let** $C \in S \setminus T$;
11 // $\{\{Inv \wedge C \in S \setminus T\}$
12 // $\{\{If1\}$
13 **if** $(diff(A, C) = \emptyset)$ **then**
14 // $\{\{If1 \wedge diff(A, C) = \emptyset\}$, *by if-then rule*
15 // $\{\{If1 \wedge T * sm(A \cup C, B) = \emptyset\}$
16 $A := A \cup C$;
17 // $\{\{If1 \wedge C \subseteq A \wedge T * sm(A, B) = \emptyset\}$, *by assignment axiom*
18 // $\{\{If2\}$, *it is the same*
19 **if** $(B \cap C = \emptyset)$ **then**
20 // $\{\{If2 \wedge B \cap C = \emptyset\}$, *by if-then rule*
21 // $\{\{IT2\}$, *it is the same*
22 **Let** $c \in C$ be a blocked literal in C, S;
23 // $\{\{IT2 \wedge Blck(c, C, S)\}$
24 // $\{\{IT2 \wedge T * sm(A, B \cup \{c\}) = \emptyset\}$, *by Lemma 2*
25 $B := B \cup \{c\}$;
26 // $\{\{If2 \wedge B \cap C \neq \emptyset \wedge T * sm(A, B) = \emptyset\}$, *by assign. axiom*
27 // $\{\{If2 \wedge B \cap C \neq \emptyset\}$
28 **fi**
29 // $\{\{If2 \wedge B \cap C \neq \emptyset\}$, *by if-else rule*
30 // $\{\{If2 \wedge C \cap sm(A, B) \neq \emptyset\}$

31 **else**
32 // $\{\{If1 \wedge diff(A,C) \neq \emptyset\}$, *by if-else rule*
33 // $\{\{If1 \wedge C \cap sm(A,B) \neq \emptyset\}$, *by Lemma 1*
34 **fi**
35 // $\{\{If1 \wedge C \cap sm(A,B) \neq \emptyset\}$
36 // $\{\{Inv \wedge C \in S \setminus T \wedge T \cup \{C\} * sm(A,B) = \emptyset\}$, *logical consequence*
37 $T := T \cup \{C\}$;
38 // $\{\{Inv \wedge T * sm(A,B) = \emptyset\}$, *by assignment axiom*
39 // $\{\{Inv\}$, *it is the same*
40 **od**
41 // $\{\{Inv \wedge T * sm(A,B) = \emptyset \wedge T = S\}$, *by while rule*
42 // $\{\{S * sm(A,B) = \emptyset\}$, *logical consequence*
43 $Z := sm(A,B)$;
44 // $\{\{S * Z = \emptyset\}$, *by assignment axiom*
45 // $\{\{M(S,Z)\}$, *postcondition*
46 **HALT**

Now we see that from the precondition (the clause set is non-empty and blocked) the postcondition (the computed assignment is a model for the clause set) follows if we follow the steps of the algorithm.

The only question is whether the algorithm always terminates or sometimes runs in an infinite loop. But it terminates always, because it does nothing else but uses each clause from the clause set one after the other and clause sets are finite sets.

Theorem 3 (Correctness of the Blocked SAT Solver)
Let S be a non-empty and blocked clause set. Then after execution of Blocked SAT Solver(S, I), I is a model for S.

> Proof: From Algorithm 2, and from the fact that it terminates we obtain that Blocked SAT Solver terminates for every non-empty and blocked clause set and gives back a model for it.

Before we discuss the worst case time complexity of the algorithm we show how to decide in linear time whether a 3-SAT problem is blocked or not.

We use a special data structure called NLC, Number of Literal Combinations. For every subset of every clause we increase the corresponding counter in NLC by one. This means that NLC contains 3 arrays, a one dimensional (1D) with $2n$ entries for counting 1 length subsets, a 2D one with $2n * 2n$ entries for counting 2 length subsets and a 3D one with $2n * 2n * 2n$ entries for counting the clauses.

We use literals as indices in square brackets. If we give one literal in the square brackets then we access the 1D array, by 2 literals the 2D one and by 3 literals the 3D one.

Since $NLC[x, y]$ is the same as $NLC[y, x]$, we assume that the indices are written in alphabetical order. Note that this means that the 2D and 3D arrays contain only zeros below the diagonal.

We initialize NLC by filling in it by zeros. Afterwards, we use it in two loops.

1. In the first loop we read the clauses of the input clause set one after the other. For all subsets of all clauses we increase the corresponding entry in NLC by one. For example if the input clause is $\{a, b, c\}$ then we increase these entries:

$$NLC[a], NLC[b], NLC[c], NLC[a, b], NLC[a, c], NLC[b, c], NLC[a, b, c].$$

This means that we do 7 read and write steps (to increase one entry, we have to read its value first, add one, and write it back) for all clauses, i.e., the first loop is an $O(2 * 7m)$ time method.

Before we go to the second loop we have to investigate the 2D and 3D arrays further. We see that they contain a lots of zeros. In the 3D one we have at most only m non zero entries. In the 2D one we have at most only $3m$ non zero entries. Hence, it is worth using hash tables instead of arrays. We assume that we can initialize these hash tables in $O(nm)$ time. Therefore, the initialization of NLC takes $O(nm + 2n)$ write steps.

2. In the second loop we read again the clauses of the input clause set one after the other. For each literal in each clause we compute the number of possible resolution partners minus the number of blocking clauses. Let us assume that the actual clause is $A = \{a, b, c\}$ and the actual literal is a. Then the possible resolution partners are those clauses that contain $\overline{a}$. We know the number of those closes, it is stored in $NLC[\overline{a}]$. We recall the definition of blocked literal:

$$Blck(a, A, S) : \Longleftrightarrow \underset{\substack{B \in S \\ \overline{a} \in B}}{\forall} \underset{\substack{\overline{x} \in B \\ \overline{x} \neq \overline{a}}}{\exists} x \in A,$$

This means that in those clauses which are possible resolution partners, there is an other literal $\overline{x}$ such that x occurs in A. In this case x is either b or c. So we obtain the number of blocking clauses as $NLC[\overline{a}, \overline{b}] + NLC[\overline{a}, \overline{c}]$. But if we do so, then we count the clauses, which contains $\overline{a}$ and $\overline{b}$ at the same time, twice. This means that we have to subtract $NLC[\overline{a}, \overline{b}, \overline{c}]$ from the number of blocking clauses. Hence, the number of possible resolution partners minus the number of blocking clauses is:

$$NLC[\overline{a}] - NLC[\overline{a}, \overline{b}] - NLC[\overline{a}, \overline{c}] + NLC[\overline{a}, \overline{b}, \overline{c}].$$

If this number is zero then it means that all possible resolution partners are at the same time blocking clauses, i.e., the literal a in the clause $\{a, b, c\}$ is blocked in the input clause set.

In the second loop we read NLC 4 times for each literal, i.e., it is a $O(4nm)$ time method in the worst case.

The overall worst case time complexity of the usage of NLC is $O(nm + 2n + 2 * 7m + 4nm) = O(5nm + 14m + 2n) = O(5nm)$.

We can also use NLC for the general SAT problem. Then we need $O(2^k)$ memory space and $O(2 * (2^k - 1)m)$ time for the first loop, and $O(2^{k-1}nm)$ time

for the second loop in the worst case, where k is the length of largest clause in the input clause set.

Now we can prove that Blocked SAT Solver is a linear time algorithm if at least one blocked literal per clause is known or the input clause set is a 3-SAT problem, otherwise it is a quadratic time algorithm.

Theorem 4 (Complexity of the Blocked SAT Solver) *Let S be a non-empty and blocked clause set. The worst-case time complexity of Blocked SAT Solver(S, I) is $O(4nm)$, if at least one blocked liter per clause is known; $O(9nm)$ if S is a 3-SAT problem; $O(n^2m^2)$, otherwise.*

Proof: If at least one blocked literal per clause is known then in the worst-case we need $O(4m)$ basic set operations, because there is only one for loop on clauses of the set and in one iteration in the worst-case we perform 4 basic set operation. Since any basic set operation can be performed in $O(n)$ time, the worst-case time complexity of Blocked SAT Solver is $O(4nm)$, i.e., linear in the number of literals.

Otherwise, if the S is a 3-SAT problem then by using NLC data structure we need $O(5nm)$ time in the worst case to provide the necessary information for the Blocked SAT Solver algorithm. This means that we need $O(9nm)$ time altogether which is linear in the number of literals. Otherwise, in the worst-case we need $O(n^2m^2)$ time, see the previous section, to provide the necessary information for the Blocked SAT Solver algorithm. Since $O(n^2m^2)$ dominates $O(4nm)$ the worst-case time complexity of Blocked SAT Solver is $O(n^2m^2)$.

5 Blocked Clause Rules

If the clause set is not blocked but it contains blocked clauses then it is a question whether we can use the blocked clauses to simplify the clause set or not? We also discuss this question briefly.

The Blocked Clear Clause Rule states that if a clause set contains only clear clauses (i.e., full clauses) and one of them is blocked, then the sub-model generated from this blocked one is a model. This is a very rare case but it serves as a basis for the next rule.

Lemma 3 (Blocked Clear Clause Rule). *Let S be a clause set. Let $C \in S$ be a blocked and clear clause. Let $a \in C$ be a blocked literal C, S. Then $sm(C, \{\ a\})$ is a model for S, provided that*

(a) either S is a subset of CC,

(b) or C is not subsumed by $S \setminus \{C\}$.

Proof:

(a) To show this, by definition of model, it suffices to show that for an arbitrary but fixed $B \in S$ we have that $B \cap sm(C, \{\ a\})$ is not empty. Since S is a subset of CC we know that B is a clear clause. Hence, there are two cases, either $a \in B$ or $\overline{a} \in B$.

In case $a \in B$ we have, by definition of sub-model, that $a \in sm(C, \{a\})$. Hence, $B \cap sm(C, \{\ a\})$ is not empty.

In case $\overline{a} \in B$, since $a \in C$ is blocked in C, S we know, by definition of blocked literal, that for some $b \in B$ we have $b \neq \overline{a}$ and $\overline{b} \in C$. From this, by definition of sub-model, we know that $b \in sm(C, \{\ a\})$. Hence, $B \cap sm(C, \{\ a\})$ is not empty.

Hence, if S is a subset of CC, then $sm(C, \{\ a\})$ is a model for S.

(b) To show this, by definition of model, it suffices to show that for an arbitrary but fixed $B \in S$ we have that $B \cap sm(C, \{\ a\})$ is not empty. Since C is not subsumed by $S \setminus \{C\}$ we know, by definition of subsumption, that $B \nsubseteq C$. From this, since C is a clear clause we know that for some $b \in B$ we have $\overline{b} \in C$. There are two cases, either $\overline{b} = a$ or $\overline{b} \neq a$.

In the first case we have $\overline{b} = a$, i.e., $\overline{a} \in B$. From this since $a \in C$ is blocked in C, S we know, by definition of blocked literal, that for some $d \in B$ we have that $d \neq \overline{a}$ and $\overline{d} \in C$. From this, by definition of sub-model, we know that $d \in sm(C, \{\ a\})$. Hence, $B \cap sm(C, \{\ a\})$ is not empty.

In the second case we have $\overline{b} \neq a$. From this and from $\overline{b} \in C$ we know, by definition of sub-model, that $b \in sm(C, \{\ a\})$. Hence, $B \cap sm(C, \{\ a\})$ is not empty.

Hence, If C is not subsumed by $S \setminus \{C\}$, then $sm(C, \{\ a\})$ is a model for S.

The Independent Blocked Clause Rule states that if a clause set contains an independent blocked clause, then it is satisfiable and a sub-model generated from this clause is a part of a model. This situation occurs quite often, but checking independent-ness is expensive.

Lemma 4 (Independent Blocked Clause Rule). *Let S be a clause set. Let $A \in S$ be blocked in S and independent in $S \setminus \{A\}$. Let $a \in A$ be a blocked literal in A, S. Then there is a model M for S such that $sm(A, \{\ a\}) \subseteq M$, i.e., $HUP(S, sm(A, \{\ a\}))$ is satisfiable.*

Proof: We know that A is independent in $S \setminus \{A\}$. Hence, by definition of independent, we know that there is a clear clause C that is subsumed by A and not subsumed by any other clause in S. Since $A \subseteq C$ we know that $sm(A, \{\ a\}) \subseteq sm(C, \{\ a\})$. Hence, it suffices to show that $sm(C, \{\ a\})$ is a model for S. To show this, by definition of model, it suffices to show that for an arbitrary but fixed $B \in S$ we have that $B \cap sm(C, \{\ a\})$ is not empty. The remaining part of the proof is the same as the proof of the (b) variant of the Blocked Clear Clause Rule.

Hence, $B \cap sm(C, \{\ a\})$ is not empty. Hence, there is a model M for S such that $sm(A, \{\ a\}) \subseteq M$.

This proof is traced back to the proof of Blocked Clear Clause Rule. We can do this because we know that there is a clear clause which is blocked and not

entailed by $S \setminus \{A\}$. Note that for clear clauses the notion of subsumed and entailed are the same.

The proof shows that if we perform an independent clause check and we find a clear clause which is subsumed by only one clause, then we know the whole model ($sm(C, \{\ a\})$) and not only a part of the model ($sm(A, \{\ a\})$).

References

[APT79] B. Aspvall, M. F. Plass, and R. E. Tarjan. A Linear-Time Algorithm for Testing the Truth of Certain Quantified Boolean Formulas. Information Processing Letters, 8(3):121–132, 1979.

[Asp80] B. Aspvall. Recognizing Disguised NR(1) Instances of the Satisfiability Problem. J. of Algorithms, 1:97–103, 1980.

[BHS94] E. Boros, P. L. Hammer, and X. Sun. Recognition of q-Horn Formulae in Linear Time. Discrete Applied Mathematics, 55:1–13, 1994.

[BCH+94] E. Boros, Y. Crama, P. L. Hammer, and M. Saks. A Complexity Index for Satisfiability Problems. SIAM J. on Computing, 23:45–49, 1994.

[CH91] V. Chandru and J. Hooker. Extended Horn Sets in Propositional Logic. J. of the ACM, 38(1):205–221, 1991.

[Coo71] S. A. Cook. The Complexity of Theorem-Proving Procedures. Proceedings of the 3rd ACM Symposium on Theory of Computing, 151–158, 1971.

[DE92] M. Dalal and D. W. Etherington. A Hierarchy of Tractable Satisfiability Problems. Information Processing Letters, 44:173–180, 1992.

[DG84] W. F. Dowling and J. H. Gallier. Linear-Time Algorithms for Testing the Satisfiability of Propositional Horn Formulae. J. of Logic Programming, 1(3):267–284, 1984.

[EIS76] S. Even, A. Itai, and A. Shamir. On the Complexity of Timetable and Multi-Commodity Flow Problems. SIAM J. on Computing, 5(4):691–703, 1976.

[Knu90] D. E. Knuth. Nested Satisfiability. Acta Informatica, 28:1–6, 1990.

[Kul99a] O. Kullmann. New methods for 3-SAT decision and worst-case analysis. Theoretical Computer Science, 223(1-2):1–72, 1999.

[Kul99b] O. Kullmann. On a Generalization of Extended Resolution. Discrete Applied Mathematics, 96-97(1-3):149–176, 1999.

[Kus02] G. Kusper. Solving the SAT Problem by Hyper-Unit Propagation. RISC Technical Report 02-02, 1–18, University Linz, Austria, 2002.

[Kus05] G. Kusper. Solving the Resolution-Free SAT Problem by Hyper-Unit Propagation in Linear Time. Annals of Mathematics and Artificial Intelligence, 43(1-4):129–136, 2005.

[Lew78] H. R. Lewis. Renaming a set of clauses as a Horn set. J. of the Association for Computing Machinery, 25:134–135, 1978.

[LS87] J. Loeckx and K. Sieber. The Foundations of Program Verification. Wiley-Teubner, second edition, 1987.

[SAF+95] J. S. Schlipf, F. Annexstein, J. Franco, and R. P. Swaminathan. On finding solutions for extended Horn formulas. Information Processing Letters, 54:133–137, 1995.

[Scu90] M. G. Scutella. A Note on Dowling and Gallier's Top-Down Algorithm for Propositional Horn Satisfiability. J. of Logic Programming, 8(3):265–273, 1990.

[Tov84] C. A. Tovey. A Simplified NP-complete Satisfiability Problem. Discrete Applied Mathematics, 8:85–89, 1984.