UnitWalk: A new SAT solver that uses local search guided by unit clause elimination *

Edward A. Hirsch†
Steklov Institute of Mathematics at St.Petersburg, 27 Fontanka, 191011 St.Petersburg, Russia, http://logic.pdmi.ras.ru/~hirsch

Arist Kojevnikov
St.Petersburg State University, Department of Mathematics and Mechanics, St.Petersburg, Russia, http://logic.pdmi.ras.ru/~arist

Abstract. In this paper we present a new randomized algorithm for SAT, i.e., the satisfiability problem for Boolean formulas in conjunctive normal form. Despite its simplicity, this algorithm performs well on many common benchmarks ranging from graph coloring problems to microprocessor verification. Our algorithm is inspired by two randomized algorithms having the best current worst-case upper bounds ([27, 28] and [30, 31]). We combine the main ideas of these algorithms in one algorithm. The two approaches we use are local search (which is used in many SAT algorithms, e.g., in GSAT [34] and WalkSAT [33]) and unit clause elimination (which is rarely used in local search algorithms). In this paper we do not prove any theoretical bounds. However, we present encouraging results of computational experiments comparing several implementations of our algorithm with other SAT solvers. We also prove that our algorithm is probabilistically approximately complete (PAC).

Keywords: Boolean satisfiability, local search, empirical evaluation

AMS Mathematics Subject Classification: 68W20; 03B05

1. Introduction

SAT (the problem of satisfiability of a Boolean formula in conjunctive normal form (*CNF*)) is one of the most well-studied $\mathcal{NP}$-complete problems. It also has many applications since individual instances of many combinatorial problems, like graph coloring, planning or circuit design problems, can be encoded into formulas in CNF in a natural way. However, under the hypothesis of $\mathcal{P} \neq \mathcal{NP}$, designing a polynomial-time algorithm for SAT is a hopeless task. During the past decade, this obstacle has been attacked in two main directions: proving "weakly exponential" worst-case upper bounds on the running time of various SAT algorithms and designing more practical "heuristic" algorithms.

* The research described in this publication was made possible in part by Award No. RM1-2409-ST-02 of the U.S. Civilian Research & Development Foundation for the Independent States of the Former Soviet Union (CRDF).

† Supported in part by RFBR grant No. 02-01-00089 and by grant No. 1 of the 6th RAS contest-expertise of young scientists projects (1999).

Weakly exponential worst-case upper bounds. Interesting "weakly exponential" upper bounds on the worst-case running time of SAT algorithms are known mostly for k-SAT (i.e., when the length of a clause is bounded by k), and are typically bounds of the form $p(n)c^n$, where $c < 2$ is a constant, n is the number of variables in the input formula, and p is a polynomial (note that SAT can be easily solved in time $p(n)2^n$, but it is non-trivial that it can be solved faster). The first such bounds were proved in [3, 22, 24]. Currently, the best known deterministic algorithm for k-SAT has the bound $p(n)(2 - 2/(k + 1))^n$ [4]. Randomized algorithms achieve even better bounds [30, 27]. If a formula has exactly one satisfying assignment, it can be found even faster: in the time $p(n)1.308^n$ for a formula in 3-CNF [27]. Concerning other results in this reach field, we refer the readers to the survey [5].

"Heuristic" algorithms. Unfortunately, the worst-case upper bounds currently known for SAT algorithms are still too large for practical purposes. Another direction of research is the design of "heuristic" algorithms. These algorithms, though they are very hard for theoretical study, show good performance in practice both on randomly generated instances and on structured instances encoding various practical problems like planning or circuit design. Theoretical knowledge about these algorithms is very limited and mostly concentrates on exponential worst-case lower bounds [11], average-case bounds [9, 19], or properties related to their completeness [2, 13, 14]. Some of these algorithms are surveyed in [10, 15].

In their turn, "heuristic" algorithms may be complete or incomplete. A *complete* algorithm gives the correct answer with certainty. If an *incomplete* algorithm finds a satisfying assignment, it is guaranteed to be correct; however, if it fails to find a satisfying assignment, this means that either the input formula is unsatisfiable, or it is satisfiable, but appeared "too hard" for this algorithm. Note that since we have no a priori bound on the probability of error, we cannot just repeat such an algorithm sufficiently many times reducing the probability of error to any predefined constant ($O(2^n)$ iterations would do the job for almost all known algorithms, but it is unrealistic in practice). There are several very successful complete algorithms (e.g., satz [21], SATO [39, 40], zChaff [25]; see comprehensive experimental results at SAT-Ex web-site[1] [37]). However, solvers based on incomplete algorithms could be faster just because they belong to a wider class of computations. For example, they are typically better on randomly generated instances.

[1] `http://www.lri.fr/~simon/satex/satex.php3`

In this paper we suggest one more incomplete heuristic algorithm for SAT; therefore, throughout this paper we will speak only about satisfiable formulas. Most incomplete "heuristic" algorithms use local search paradigm (this line of research started from experiments of Gu [8] and Selman, Levesque and Mitchell [34] and theoretical work by Koutsoupias and Papadimitriou [19, 26]). Such an algorithm chooses a random initial assignment and then modifies it step by step until it finds a satisfying assignment. If the random walk is long enough, it is restarted from another initial assignment. There is a number of experimentally studied SAT local search algorithms, for example, GSAT [34], GWSAT (aka GSAT+w) [33], HSAT [6], HWSAT [7], SDF [32], IDB [29], WalkSAT [33] and various algorithms within WalkSAT framework such as WalkSAT/TABU, Novelty, R-Novelty [23], Novelty+ and R-Novelty+ [14].

Some of these algorithms *need* restarts because there are initial assignments such that the probability that the random walk (if allowed to run infinitely long) hits a satisfying assignment is strictly less than one (in fact, it can be zero for most of these algorithms). Such algorithms are called *essentially incomplete* [14] (e.g., GSAT, WalkSAT/TABU, Novelty, R-Novelty [13, 14]). Other algorithms are *probabilistically approximately complete* (*PAC*) [14], i.e., they succeed with probability one without restarts for *every* initial assignment (e.g., Novelty+, R-Novelty+, or GWSAT with strictly positive noise parameter [13, 14]).

Unit clause elimination is a common technique in complete SAT algorithms. However, it seemed hard to use it in local search algorithms. Recently, there has been such an attempt [29]. Our local search algorithm also uses unit clause elimination, though in a different way.

Procedure of Paturi, Pudlák, Saks, and Zane and our algorithm. In this paper, we suggest a new incomplete SAT algorithm UnitWalk. Similarly to the experimentally best incomplete algorithms from WalkSAT family [23] and the theoretically best algorithms by Schöning [30] and Schuler, Schöning and Watanabe [31], our algorithm uses local search. However, the heuristic for flipping a variable is motivated by another theoretically best algorithm by Paturi, Pudlák, Saks, and Zane [27].

The core of the algorithm in [27] is the following procedure [28]. Take a random assignment for the input formula F and a random permutation of variables of F. Consider all variables in the order determined by this permutation. For each variable v, do the following. If the value of v is forced by a unit clause, then set the forced value to the variable; otherwise, take the value from the random assignment. In any case, make the corresponding substitution in the formula, i.e.,

replace F by $F[v \leftarrow \texttt{True}]$ or $F[v \leftarrow \texttt{False}]$ respectively by removing all clauses containing the literal (either v or $\neg v$) having the value `True` and removing the opposite literal from the remaining clauses.

Paturi, Pudlák, and Zane [28] show that this procedure finds a satisfying assignment for a formula in 3-CNF with probability at least $O(2^{-2n/3})$ (in other words, with high probability, only $2n/3$ values in the initial assignment are essential). Therefore, repeating this procedure $O(2^{2n/3})$ times gives a constant probability of error (for any predefined constant). This algorithm can be derandomized by enumerating $2^{2n/3}$ assignments and a polynomial number of permutations ([28], see also [5] for a simpler construction).

Paturi, Pudlák, Saks and Zane [27] give an extension of this algorithm. The new algorithm includes a preprocessing step: it adds some resolvents to the input formula. This allows to achieve a higher probability of success in one iteration and hence the improved running time $p(n)1.363^n$ (or even $p(n)1.308^n$ if there is only one satisfying assignment).

In [28, 27], the procedure described above is used for obtaining a satisfying assignment in one iteration: if a satisfying assignment is not found, the procedure is restarted with a new random assignment and a new random permutation. In our algorithm, we use this procedure to obtain an assignment that is closer to a satisfying assignment than the initial assignment (i.e., the obtained unsatisfying assignment is not dropped, but is used instead of a random assignment for the next iteration of the procedure). In fact, we use a slightly modified procedure based on the version described in [5]. Namely, we process unit clauses as soon as they appear (irrespectively of the chosen permutation of variables). We also make a random choice if more than one unit clause appears. If there are two unit clauses of the opposite signs, we do *not* flip the corresponding variable. Finally, if no variable is flipped after considering all variables, we flip a randomly chosen variable (in fact, this is a very rare situation). See Section 3 for more details about our algorithm and its relation to other heuristic algorithms such as WalkSAT/TABU [23].

We prove that our algorithm is probabilistically approximately complete. We also implemented a SAT solver based on our algorithm; we provide the experimental results of running it on various widely known benchmarks. The C source code of various versions of UnitWalk solver is available from `http://logic.pdmi.ras.ru/~arist/UnitWalk/`.

Organization of the paper. The paper is organized as follows. In Section 2 we briefly describe notation related to Boolean satisfiability and SAT algorithms. Our basic algorithm is described in Section 3, where

it is also shown that the algorithm is probabilistically approximately complete, and its relation to other local search algorithms is discussed. Section 4 contains the details of our solver that took part in SAT Competition 2002 [36]: we discuss its implementation and improvements using other known algorithms. In Section 5 we present comprehensive experimental data describing the execution of our algorithm on various benchmarks including some of the benchmarks used in SAT Competition 2002. Finally, in Section 6 we summarize our results and point directions for further research.

2. Preliminaries

We consider algorithms for the problem of satisfiability of a Boolean formula in conjunctive normal form (*CNF*). A formula in CNF is the conjunction of clauses, a clause is the disjunction of literals, and a literal is a Boolean variable or its negation. A *satisfying assignment* S for a formula F is a truth assignment for the variables appearing in F such that every clause of F has the value `True` under S. If such an assignment exists, then F is called *satisfiable*. The satisfiability problem (*SAT*) can be formulated as follows: given a formula in CNF, find a satisfying assignment[2] for it, or answer "Unsatisfiable" if there are no such assignments.

For any formula F, variable v and truth value t (which may be `True` or `False`), we form the formula $F[v \leftarrow t]$ as follows. If $t =$ `True`, we remove all clauses containing positive occurrences of v from F, and remove the literal $\neg v$ from the remaining clauses of F. If $t =$ `False`, we remove all clauses containing negative occurrences of v from F, and remove the literal v from the remaining clauses of F. In other words, we substitute t for v in F and simplify the clauses that contained the variable v.

We denote the value of a variable v in an assignment A by $A[v]$. The *Hamming distance* between two assignments A and B is the number of variables having different values in A and B.

We study *incomplete randomized* algorithms for SAT. Such an algorithm either finds a (correct) satisfying assignment for the input formula, or gives the answer "Not found". In the latter case the formula may be either unsatisfiable or satisfiable. If there is an *a priori* worst-case bound on the probability of error (an error is the event of giving

[2] SAT is classically formulated as a decision problem: given a Boolean formula F in CNF, output "Satisfiable" if it has a satisfying assignment, otherwise output "Unsatisfiable". Clearly, it is polynomial-time equivalent to the version that we study.

the answer "Not found" for a satisfiable formula), then by repeating the algorithm sufficiently many times, the probability of error can be reduced to any predefined constant, and after that the answer "Not found" can be treated as "Unsatisfiable".

Most incomplete algorithms for SAT are *local search* algorithms. A local search algorithm chooses an initial assignment at random. At each step, it changes (at most) one value in it, trying to get closer to a satisfying assignment. If an algorithm changes the value of a variable, we say that it *flips* this *value*, or even *flips* this *variable*. This procedure (as well as the whole algorithm) is called *probabilistically approximately complete* (*PAC*) [14] if for every satisfiable formula and *every* initial assignment the procedure finds a satisfying assignment with probability one. Even if a local search algorithm has the PAC property, it may be more efficient to choose another initial assignment and restart the random walk if a satisfying assignment is not found after a certain number of steps. After sufficiently many unsuccessful restarts, the output "Not found" is given.

3. The basic algorithm

3.1. Description

As a typical local search algorithm, our algorithm generates an initial assignment at random and then modifies it step by step. The main difference from other local search algorithms is that during this walk our algorithm modifies also the input formula.

The random walk is divided into *periods*. During one period at least one (usually, much more) flip is made. A period starts with choosing a random permutation of variables. Then algorithm takes the input formula and modifies it step by step, sometimes also modifying the current assignment. At each step, the algorithm substitutes the value of one variable in the current formula, i.e., replaces a formula G by the formula $G[v \leftarrow t]$ for a variable v and a truth value t. If there are unit clauses, then v is taken from one of them; if the value of v does not satisfy the unit clause and satisfies no other unit clause, it is flipped before the substitution. If there are no unit clauses, the algorithm substitutes the value to the next variable in the chosen permutation (taking the value from the current assignment).

If a period finishes (i.e., all variables are processed), but no variable was flipped during it, the algorithm chooses a variable at random and flips it (in fact, this is a very rare situation). After period finishes, the algorithm chooses a new random permutation, replaces the current

Input: A formula F in CNF containing n variables $x_1, \ldots, x_n$.

Output: A satisfying assignment for F, or "Not found".

Method:

For $t := 1$ to `MAX_TRIES`(F) do

$A :=$ random truth assignment for n variables;
For $p := 1$ to `MAX_PERIODS`(F) do
$\pi :=$ random permutation of $1..n$;
$G := F$;
$f := 0$;
For $i := 1$ to n do
While G contains a unit clause, repeat
- Pick a unit clause $\{x_j\}$ or $\{\neg x_j\}$ from G at random;
- If this clause is not satisfied by A, and G does not contain the opposite unit clause, then flip $A[j]$ and set $f := 1$;
- $G := G[x_j \leftarrow A[j]]$;

If variable $x_{\pi[i]}$ still appears in G, then $G := G[x_{\pi[i]} \leftarrow A[\pi[i]]]$.
If G contains no clauses (i.e., $G \equiv$ `True`), then output A and exit;
If $f = 0$, choose j at random from $1..n$ and flip $A[j]$.

Output "Not found".

Figure 1. Algorithm UnitWalk

formula (which is trivial now) by the input formula, and starts a new period. The number of periods is limited to `MAX_PERIODS`(F) which may be a function of certain syntactic characteristics of the input formula, e.g., of the number of variables. After the last period finishes, the random walk is restarted from another random initial assignment. If a satisfying assignment is not found after taking `MAX_TRIES`(F) initial assignments, the algorithm outputs the answer "Not found". One of the practical choices for `MAX_PERIODS` and `MAX_TRIES` is to set `MAX_TRIES`(F) to 1 and `MAX_PERIODS`(F) to $+\infty$ (clearly, in this case the algorithm will never stop if given an unsatisfiable formula).

Note that instead of choosing a random permutation in the beginning of each period, we could generate it "on the fly", i.e., choose a variable at random every time it is needed (from the set of all variables that had not yet been substituted by their values). However, we formulate our algorithm using permutations to stress its relation to the procedure of Paturi, Pudlák, Saks and Zane [28, 27].

A more formal description of the algorithm is given in Fig. 1.

3.2. Probabilistic approximate completeness

The following theorem shows that our algorithm is probabilistically approximately complete, i.e., if we set `MAX_PERIODS`(F) to $+\infty$ and `MAX_TRIES`(F) to 1, then for every satisfiable formula and *every* initial assignment, UnitWalk finds a satisfying assignment with probability one.

THEOREM 1. *Algorithm UnitWalk is probabilistically approximately complete.*

Proof. We now prove that for any A and for any satisfying assignment S for the formula F during one period either the Hamming distance[3] between A and S decreases with probability bounded from below or the algorithm outputs a satisfying assignment.

Consider any satisfying assignment S and an assignment A at distance d of the assignment S. We now construct a permutation π such that if it is chosen in the beginning of the period starting with A (note that every permutation is chosen with probability $\frac{1}{n!}$), then the assignment obtained in the end of the period will be closer to S than A.

Let $\pi[1] = i_1, \ldots, \pi[n-d] = i_{n-d}$, where $x_{i_1}, \ldots, x_{i_{n-d}}$ are the variables on which A agrees with S. Clearly, the values of these variables will not be changed during the period. The remaining variables have different values in A and S. Thus, if at least one of them is flipped during the period, then we are done. Note that these are the only variables whose flippings can be forced by unit clauses. However, if no variable is flipped during the period and satisfying assignment is not found, then after the period finishes, UnitWalk chooses a variable at random and flips it. With probability at least $1/n$, it chooses a variable whose values in A and S are different.

Therefore, every period decreases the Hamming distance between A and S with probability at least $\frac{1}{n \cdot n!}$. The probability of outputting a satisfying assignment in at most n periods is thus at least $b(n) = \frac{1}{(n \cdot n!)^n}$ (irrespectively of the initial assignment). Hence, the overall probability of outputting a satisfying assignment is at least $b(n) + (1 - b(n))b(n) + (1 - b(n))^2 b(n) + \ldots = 1$. □

[3] The *Hamming distance* between two assignments A and B is the number of variables having different values in A and B.

3.3. Relation to other local search algorithms

Our algorithm is similar to other local search algorithm because at each step it modifies the value of at most one variable. However, it has an important difference, namely, the use of unit clause elimination. Probably, the closest algorithm to UnitWalk is WalkSAT/TABU [23]. We now reformulate our algorithm in terms of tabu lists (instead of formula modification) and compare it to WalkSAT/TABU.

Similarly to other algorithms in the WalkSAT family [23], WalkSAT/TABU chooses a variable for flipping from an unsatisfied clause (our algorithm behaves in the same way). It maintains a *tabu list* of variables whose flippings are disabled. This list consists of variables *flipped* during the last t steps (t is a parameter). For our algorithm, the tabu list contains all variables whose values were *substituted* into the formula during the current period (the first important difference is that sometimes we put a variable on the tabu list *without flipping* it). In both algorithms, clauses consisting of tabu variables only are not considered. If all unsatisfied clauses consist of tabu variables, WalkSAT/TABU ignores the tabu list; our algorithm empties this list. One more difference is the choice of an unsatisfied clause: WalkSAT/TABU chooses it at random from the set of all unsatisfied clauses containing *at least* one non-tabu variable; our algorithm chooses it at random from the set of all unsatisfied clauses containing *exactly* one non-tabu variable.

The following arguments show that these differences are essential.

1. Our algorithm is probabilistically approximately complete (see Theorem 1) while WalkSAT/TABU is essentially incomplete [13, 14].

2. Experimental data show that UnitWalk makes substantially less *flips* than WalkSAT/TABU on almost all instances (see tables in [12]).

4. The solver

The algorithm described in Section 3 is interesting in its own right. However, life is always more complex than theory. Therefore, writing a solver that performs well if run on a real computer is another kind of art. In particular, one has to implement efficient data structures, incorporate known heuristics, sometimes worsen the basic algorithm in theoretical sense, combine it with other approaches, etc. In this section we describe the way we did it: namely, the implementation details and

the improvements to the basic algorithm that led to better performance. The aim of this section is to describe the version of our solver that participated in SAT Competition 2002 [36].

4.1. Implementation

Since our basic algorithm is very simple (in particular, it does not use any hard to compute functions), even its first "rough" implementation was quite competitive (see Table I for a quick comparison with other solvers; the details of our experiments and more detailed tables are given in Section 5) and, in fact, the implementation details has not changed much since that time. Our data structures are similar to those used in many complete SAT algorithms (e.g., GRASP [35]).

The solver is implemented in `C` and uses only the standard `C` library. The implementation represents a formula in CNF by an array of clauses, where each clause is represented by its size (i.e., a natural number) and a link to an array of literals (i.e., integer numbers). When we substitute a value for a variable, we only change the sizes of clauses:

- the size of every satisfied clause is set to zero;
- the size of every other clause containing this variable is decreased by one, and the corresponding literal is exchanged with the last literal of this clause.

This implementation of substitution helps us to restore the original formula quickly, because all we need for that is to restore the sizes of clauses.

We also use two additional structures; supporting them is also quite inexpensive. For each variable, we maintain the list of clauses containing this variable. Also, when a unit clause appears, we put its index (in the array of clauses) on a special list of indices of unit clauses.

4.2. Adding resolvents

Paturi, Pudlák, Saks and Zane [27] suggest an extension of the original algorithm of [28], and this extension gives an improvement of the worst-case time upper bound. The extension is a preprocessing step that adds resolvents of logarithmic size to the input formula.

In practice, one cannot compute such large resolvents, and sometimes even adding all resolvents of size, say, four gives an intolerable blowup. Thus, our extension of UnitWalk limits the size of resolvent by two. Frequently, the input formula has no such resolvents. On the other hand, adding new resolvents after substitutions is useful because this is one more rule of simplification. Our implementation adds *some* of the

Table I. UnitWalk (basic algorithm) vs other solvers.

	UnitWalk substi-tutions	UnitWalk flips(time)	GSAT flips(time)	WalkSAT flips(time)	Novelty flips(time)	R-Novelty flips(time)	SDF flips(time)	IDB time
uf100-430	13,500	2,547(0.029)	*	3,652(0.009)	15,699(0.047)	1,536(0.005)	876(0.01)	
ais10	432,191	18,405(6.077)	*	173,422(1.978)	*	*	20,870(1.52)	
ais12	$6.0{\cdot}10^6$	222,967(142.5)	*	*$2.9{\cdot}10^6$(34.28)	*	*	*154,249(18.6)	
3bitadd_31	753,820	4,778(7.164)	*	28,511(1.090)	*	210,907(19.54)	*	11.0
3bitadd_32	416,921	3,743(4.757)	*	14,209(0.628)	*	*	*	8.6
par8-5-c	5,311	863(0.011)	*21,859(0.160)	*19,182(0.056)	4,318(0.012)	2,640(0.007)	3,691(0.033)	
par16-5-c	$13{\cdot}10^6$	$3.1{\cdot}10^6$(33.85)	*	*	$86 \cdot 10^6$(236.6)	$50 \cdot 10^6$(145.2)	*	
flat50-115	1,521	261(0.002)	*16,857(0.10)	3,896(0.008)	*24,421(0.027)	*9,373(0.019)	773(0.01)	
flat100-239	18,833	3,500(0.051)	*752,775(4.63)	44,900(0.164)	17,893(0.036)	12,345(0.027)	6,983(0.15)	
aim100	24,059	4,020(0.048)	*	*251,618(0.445)	*269,700(0.453)	*265,862(0.456)	*116,467(1.89)	
ii8	1,468	203(0.021)	*169,366(7.09)	495(0.021)	5,593(0.180)	*3,166(0.010)	3,946(0.80)	
ii16	9,093	1,086(0.273)	*	6,201(0.769)	*91,031(10.07)	*85,742(4.036)	7,180(1.35)	0.305
ii32	5,989	854(0.227)	*	2,284(1.040)	*88,779(6.775)	*77,589(3.736)	*	1.17
logistics.a	$40{\cdot}10^6$	$1.2{\cdot}10^6$(920.7)	*	95,205(0.432)	55,748(0.332)	45,220(0.259)		
logistics.d	103,686	4,120(0.940)	*	472,513(3.013)	136,938(1.187)	$1.1{\cdot}10^6$(12.63)	74,090(56.7)	
3blocks	116,808	3,193(4.851)	*	33,256(0.643)	13,696(0.270)	8,304(0.179)	10,561(2.60)	
f600	*	*	*	167,616(0.778)			182,985(18.3)	4.35
f1000	*	*	*	639,459(3.727)			$5.6{\cdot}10^6$(1192)	84.1
f2000	*	*	*	$4.5{\cdot}10^6$(36.51)			*	686

*An asterisk means that MAX_FLIPS(F) was reached in one or more runs. A standalone asterisk means that the algorithm failed in all runs (in some of the formulas of the given series, where applicable). The empty cell means we do not have data. For series of formulas, an average is given.

Table II. Generating resolvents during formula preprocessing vs incBinSat. The number of substitutions $\cdot 10^{-6}$ and elapsed time (sec.) is shown.

	the basic algorithm	pre-generated (size ≤ 3)	incBinSat [ZS02]
`uf100` (average)	.013(.03)	.010(.04)	.010(.04)
`logistics.a`	40(920)	2(465)	7(132)
`bw_large.c`$^{+}$	25(726)	14(2,634)	5(259)
`bw_large.d`	*	*	358(23,541)

$^{+}$For `bw_large.c`, only resolvents of size ≤ 2 were generated.

Table III. Combining UnitWalk (enhanced by incBinSat) with WalkSAT.

	UnitWalk		UnitWalk+incBinSat		UnitWalk+incBinSat +WalkSAT		WalkSat
	substi-tutions	flips(time)	substi-tutions	flips(time)	substi-tutions	flips(time)	flips(time)
aim100	24,059	4,020(0.048)	10,528	2,181(0.034)	4,062	2,288(0.021)	*251,618(0.445)
aim200	$7.0 \cdot 10^6$	$1.5 \cdot 10^6$(11.88)	$1.1 \cdot 10^6$	332,126(4.453)	76,578	18,201(0.267)	*
ii8	1,468	203(0.021)	986	193(0.017)	1,017	194(0.018)	495(0.021)
ii16	9,093	1,086(0.273)	7,771	1,113(0.405)	5,256	2,284(0.426)	6,201(0.769)
ii32	5,989	854(0.227)	6,140	964(0.592)	2,103	1,082(0.273)	2,284(1.040)
qg1-07	72,304	5,642(36.13)	22,740	2,058(16.19)	30,424	20,334(44.24)	$3.3 \cdot 10^6$(730.1)
qg1-08	*$1.5 \cdot 10^6$	*115,280(1,757)	*	*	*	*	*
qg2-07	35,843	3,045(18.35)	33,819	3,121(24.67)	28,709	32,080(58.42)	$1.7 \cdot 10^6$(375.6)
qg3-08	109,977	8,274(3.660)	62,617	5,030(3.656)	61,030	149,124(12.57)	$45 \cdot 10^6$(627.3)
qg4-09	882,454	63,602(32.79)	301,587	22,828(20.65)	756,264	109,950(55.08)	*
qg5-11	721,535	32,552(65.63)	487,678	22,813(75.91)	*	*	*
qg6-09	22,890	1,578(1.296)	13,194	1,061(1.277)	6,852	$1.1 \cdot 10^6$(150.0)	*
qg7-09	6,196	645(0.381)	2,332	439(0.242)	2,697	425,693(69.39)	*
qg7-13	*$27 \cdot 10^6$	*$1.2 \cdot 10^6$(3,598)	*	*	*	*	*

2-resolvents after each substitution using a very fast method incBinSat of Zheng and Stuckey [41]. Therefore, we trade the number of possible implications (and simplifications) for the time spent for each period.

Our choice is partially supported by our experimental results shown in Table II. It turns out that while adding the resolvents during preprocessing sometimes gives better decrease in the number of substitutions, the running time is better for the version of our solver which uses incBinSat. The formal presentation of this version is given in Fig. 2.

Input: A formula F in CNF containing n variables $x_1, \ldots, x_n$.

Output: A satisfying assignment for F, or "Not found".

Method:

For $t := 1$ to `MAX_TRIES`(F) do

$A :=$ random truth assignment for n variables;

(!) For $p := 1$ to `MAX_PERIODS`(F) do

$\pi :=$ random permutation of $1..n$;
$G := F$;
$f := 0$;
$temporary\_assignment := \emptyset$
$current\_time := 0$
For $i := 1$ to n do
While G contains a unit clause, repeat
- Pick a unit clause $\{x_j\}$ or $\{\neg x_j\}$ from G at random;
- If this clause is not satisfied by A, and G does not contain the opposite unit clause, then flip $A[j]$ and set $f := 1$;
- $G := G[x_j \leftarrow A[j]]$;
- While G contains a 2-clause $\{x, y\}$, repeat
  - $\star$ $current\_time := current\_time + 1$;
  - $\star$ If G contains a unit clause, leave the inner "while" cycle;
  - $\star$ Call IncTempPropUnit$(x, temporary_assignment)$.

If variable $x_{\pi[i]}$ still appears in G, then $G := G[x_{\pi[i]} \leftarrow A[\pi[i]]]$.
If G contains no clauses (i.e., $G \equiv$ `True`), then output A and exit;

(!!) If $f = 0$, choose j at random from $1..n$ and flip $A[j]$.

Output "Not found".

procedure incTempPropUnit(x, A)

If$(\{\neg x\} \in A$ and $time(\{\neg x\}) = current\_time)$, then return;
$A := \{A - [\{x\}, \{\neg x\}]\} \cup \{x\}$;
$time(\{x\}) := current\_time$;
For each 2-clause $\{\neg x, y\} \in G$ do
If $\{y\} \notin A$, then call $incTempPropUnit(y, A)$.

Figure 2. Algorithm UnitWalk enhanced by incBinSat

4.3. Combination with traditional local search

A careful reader may have already noticed that UnitWalk and WalkSAT are, in a sense, complementary: UnitWalk solves formulas that WalkSAT does not solve, and vice versa (see, e.g., Table I). Therefore, it is natural to unite these approaches in one solver. The version of

Input: A formula F in CNF containing n variables $x_1, \ldots, x_n$.

Output: A satisfying assignment for F, or "Not found".

Method:

$A :=$ random truth assignment for n variables.

Repeat the following steps until a satisfying assignment is found:

a) Repeat "periods" of UnitWalk enhanced with incBinSat (the cycle (!)–(!!) in Fig. 2); however, stop not when `MAX_PERIODS` is reached, but instead when $c \geq n/12$ opposite unit clauses pairs are found during a period, and $c' \leq c$ of them are found during the previous period.

b) Make a WalkSAT walk (see [33]) with cutoff=$n^2/2$ and noise=50%.

Figure 3. Solver UnitWalk 0.98

UnitWalk that participated in SAT Competition 2002 [36] corresponds to a combination of WalkSAT with UnitWalk enhanced by incBinSat.

Although a *mechanical* combination of WalkSAT and UnitWalk (say, running them in parallel) would give a provably stronger algorithm, it turns out that we can get even more: there are formulas for which *our* combination works better than both WalkSAT and UnitWalk (see Table III). This is achieved by alternating WalkSAT-like and UnitWalk-like fragments of the random walk. Unfortunately, sometimes it also leads to worse performance (mainly, on some of the `qg` formulas). While increasing the running time by a constant factor is natural for a combination of several algorithms, tuning the parameters of this combination may probably improve the performance. Fig. 3 describes the particular version that has been used for SAT Competition 2002 (note that the last assignment for a UnitWalk-like fragment is the initial assignment for the next WalkSAT-like fragment, and vice versa).

5. Experimental data

Evaluation criteria. Comparing algorithms empirically is a difficult task. If one chooses to compare algorithms w.r.t. computation time (i.e., the most natural characteristic!), several problems emerge. First, it is impossible to create an ideal computer environment; frequently, most powerful systems are multi-tasking and even multi-user, and therefore the state of the system at a given moment of time may influence the measurements (even if one measures CPU time spent purely for the investigated computational process). The second problem is that it is almost impossible to use experimental data provided by other people

since it is not easy to find a computer with the same characteristics, and scaling the data to an available computer is imprecise. The most important problem is that such a comparison is very implementation-dependent (and it also depends on the compiler version used to compile the program).

Another possibility is to compare algorithms w.r.t. some implementation-independent measure. For the case of local search algorithms, a natural measure is the number of flips made until a solution is reached. This approach has another drawback: different algorithms spend very different amounts of time per flip, because some of them (such as ours) do a non-negligible amount of work to determine a variable to flip. However, one may argue that this work takes "polynomial time" per flip while the total number of flips in all iterations is the essential "exponential" component of the running time (cf. worst-case analysis of [30, 31]). See [15] for more about this subject. For our algorithm, we give both the number of flips and the number of substitutions made which seems a more realistic measure for the current implementation of the basic algorithm (but not for the solver as a whole because it uses WalkSAT-like walks as well). We also give CPU time.

Benchmarks. The main source where we took benchmarks was the online library SATLIB[4] [16]; see references there for the description and original sources of these benchmarks. In addition, we ran our algorithm on Velev's microprocessor verification benchmarks [5] [38]. We also used randomly generated benchmarks submitted to SAT Competition 2002 [36] by the first author, including the smallest satisfiable benchmark `hgen2-v500-s1216665065` remained unsolved during the competition (as well as during our tests). Clearly, we selected only satisfiable benchmarks from all these series.

Our experiments and other sources of data. Most of our experiments were made on a 466MHz Intel Celeron Pentium II running under Linux. The DIMACS hardware benchmark program *dfmax r500.5* [17] takes 54.64 seconds on the machine. Some of the experiments were made on a 1GHz Pentium-III machine and the running time was scaled to match our basic machine. The data for other algorithms is partially taken from SAT Competition 2002 [36] results[6] and from other sources [13, 15, 29]. In particular, [13, 15] gives comprehensive data concerning the number of flips of HWSAT, GWSAT, GWSAT/TABU, WalkSAT/TABU and other algorithms from WalkSAT family. How-

[4] `http://www.satlib.org/`
[5] `http://www.ece.cmu.edu/~mvelev/`
[6] See `http://www.satlive.org/SATCompetition/`.

Table IV. The experimental results on various SATLIB benchmarks.

	UnitWalk+incBinSat+WalkSat		WalkSat	Novelty	R-Novelty	SDF	IDB
	substitutions	flips(time)	flips(time)	flips(time)	flips(time)	flips(time)	time
uf100-430	750	3,405(0.028)	3,652(0.009)	15,699(0.047)	1,536(0.005)	876(0.01)	
uf250-1065	6,406	75,101(0.677)	63,778(0.382)				
flat150-360	186,696	47,326(0.975)	254,188(0.562)	130,359(0.296)	*731,825(1.429)	39,156(0.86)	
flat200-479	$2.0 \cdot 10^6$	539,445(14.78)	869,162(3.828)				
aim100	4,062	2,288(0.021)	*251,618(0.445)	*269,700(0.453)	*265,862(0.456)	*116,467(1.89)	
aim200	76,578	18,201(0.267)	*				
ii16	5,256	2,284(0.426)	6,201(0.769)	*91,031(10.07)	*85,742(4.036)	7,180(1.35)	0.305
ii32	2,103	1,082(0.273)	2,284(1.040)	*88,779(6.775)	*77,589(3.736)	*	1.17
ssa	5,564	753(0.034)	45,181(0.171)	*650,156(7.024)	$*1.4 \cdot 10^6$(13.76)	27,238(2.15)	0.31
logistics.a	$7.1 \cdot 10^6$	256,008(132.2)	95,205(0.432)	55,748(0.332)	45,220(0.259)		
logistics.b	768,495	16,319(14.73)	229,529(1.314)				
logistics.c	$8.7 \cdot 10^6$	199,025(196.8)	554,260(3.451)				
logistics.d	73,946	3,731(1.128)	472,513(3.013)	136,938(1.187)	$1.1 \cdot 10^6$(12.63)	74,090(56.7)	
ais8	40,550	2,528(0.587)	28,528(0.152)	*	$*1.2 \cdot 10^6$(6.529)	4,645(0.17)	
ais10	301,745	15,129(8.620)	173,422(1.978)	*	*	20,870(1.52)	
ais12	$5.2 \cdot 10^6$	223,800(235.1)	*	*	*154,249(18.6)		
f600	21,552	14,085(1.988)	167,616(0.778)				4.35
f1000	22,400	653,315(10.51)	639,459(3.727)				84.1
f2000	$1.2 \cdot 10^6$	$3.9 \cdot 10^6$(118.7)	$4.5 \cdot 10^6$(36.51)				686
bw_large.a	6,779	744(0.162)	19,158(0.093)	9,699(0.050)	6,911(0.039)	2,917(0.184)	0.425
bw_large.b	83,438	5,544(2.885)	480,187(3.793)	203,813(2.005)	347,827(4.443)	38,927(6.44)	7.75
bw_large.c	$5.0 \cdot 10^6$	243,478(258.2)	$14.1 \cdot 10^6$(151.8)	$6.4 \cdot 10^6$(200.8)	*	*	
bw_large.d	$358 \cdot 10^6$	$14.1 \cdot 10^6$(23541)					
2bitadd_11	5,879	286(0.021)	1,174(0.005)	*99,489(0.357)	*139,062(0.498)	166,089(11.1)	0.026
2bitadd_12	4,248	252(0.017)	751(0.004)	*99,478(0.411)	*53,076(0.194)	111,328(8.07)	0.028

Table V. The number of solved instances generated by hgen2 out of 5 instances generated for the same number of variables $v = 250, 300, \ldots$ An instance is counted as solved if it was solved in at least 50% of the runs. UnitWalk+... and WalkSAT were run for 10 times (by the authors), and other algorithms were run once (during SAT Competition 2002).

	UnitWalk +incBinSat +WalkSAT	dlmsat1	oksolver	saturn	usat10	WalkSAT
hgen2-v250	5	5	5	5	5	5
hgen2-v300	4	5	2	2	2	3
hgen2-v350	5	5	–	3	5	5
hgen2-v400	4	4	–	4	4	4
hgen2-v450	4	4	–	3	4	4
hgen2-v500	3	3	–	–	1	2
hgen2-v600	3	1	–	–	3	3
hgen2-v650	–	–	–	–	–	2
hgen2-v700	1	–	–	–	–	1

ever, in many cases it does not give CPU time; therefore, we had to make some experiments ourselves. Namely, we have gathered the data for WalkSAT, Novelty and R-Novelty by running on our machine their implementations taken from `http://www.cs.washington.edu/homes/kautz/walksat/`. We have also made the experiments with implementations of GSAT and SDF taken from `http://logos.uwaterloo.ca/~dale/`. The data for IDB is taken from [29] where the experiments were made on a 400 Mhz Intel Pentium II. Saturn is a newer implementation of the same algorithm; the data for it (as well as the data for OKSolver, dlmsatX, zChaff, limmat and usat10) is taken from SAT Competition 2002 web site[6] (note that OKSolver was the winner for randomly generated instances, and zChaff and limmat were the winners for industrial instances). All experiments except for the data taken from SAT Competition 2002 results were repeated at least 10 times, and the median was taken. For the data where the machine was substantially different to our basic machine, scaling was made using dfmax [17] program where possible.

In most of the experiments of [13, 15], the number of restarts (from a new initial assignment) was set to one. For our experiments with UnitWalk on SATLIB instances, we have made the same thing: `MAX_TRIES`$(F) = 1$ and `MAX_PERIODS`$(F) = +\infty$, i.e., an algorithm performs just one (long) random walk. For other algorithms we could not present some data unless `MAX_FLIPS` would be set to a finite number, because they

could occasionally loop forever (especially, essentially incomplete algorithms). We set `MAX_FLIPS` $= 10,000,000$ for `par8`, `par16-c` and `ais` series and `MAX_FLIPS` $= 1,000,000$ on other benchmarks. For the following benchmarks from SAT Competition 2002, a timeout was set for all algorithms (including UnitWalk): Velev's benchmarks (2400 seconds on 450 MHz P-III machines) and hgen benchmarks (1200 seconds on Athlon 1800+ machines).

The experimental data is organized in Tables I–VII. We give 'mean' data where available; i.e., the solver terminated in the shown amount of time in at least 50% of the experiments.

6. Conclusion and Further Research

In this paper we suggested a new local search algorithm for SAT which we call *UnitWalk*. The main difference of our algorithm from other local search algorithms for SAT is the use of unit clause elimination (which is widely used in complete algorithms but seemed hard to combine with local search). We also sketched the implementation details and presented several ways to improve the practical behaviour of our algorithm. The experimental data we present in the paper show that our algorithm dominates other contemporary incomplete solvers on some sets of benchmarks (for example, satisfiable instances of `aim` [1], `ssa` [20], `par` [17] and `ii` [18] series) and is able to solve some very hard SAT instances (for example, Velev's instances). SAT Competition 2002 has shown that UnitWalk is not a specialized solver; on the contrary, it is able to solve benchmarks from various fields (note that UnitWalk was in the top five solvers list for all applicable categories of benchmarks).

The two major open quesions concerning UnitWalk are:

- Design a complete algorithm based on UnitWalk. (See [5] for survey of related derandomization issues).
- Prove upper and lower bounds on the running time of UnitWalk.

Table VI. Results on formulas generated by hgen5.

	UnitWalk+incBinSat+WalkSAT		dlmsat1	oksolver	saturn	usat10	WalkSAT
	substitutions	flips(time)	time	time	time	time	flips(time)
hgen5-v100-s1064278966	61,440	$1.7{\cdot}10^6$(26.10)	0.61	0.48	27.64	300.6	$5.3{\cdot}10^6$(44.70)
hgen5-v100-s1398869456	37,550	$1.0{\cdot}10^6$(15.94)	8.80	0.43	33.62	17.98	953,233(8.069)
hgen5-v100-s1478813564	47,640	$1.3{\cdot}10^6$(20.32)	0.39	0.09	30.11	58.84	$2.9{\cdot}10^6$(24.57)
hgen5-v100-s1818647520	101,770	$2.8{\cdot}10^6$(43.61)	0.56	1.08	95.80	17.59	$2.4{\cdot}10^6$(19.88)
hgen5-v100-s2029002754	24,560	672,272(10.54)	0.13	0.17	92.55	18.98	245,198(2.078)
hgen5-v125-s1345272240	169,150	$5.8{\cdot}10^6$(93.00)	10.01	1.00	78.52	375.8	$3.8{\cdot}10^6$(32.57)
hgen5-v125-s1840040075	685,275	$23.7{\cdot}10^6$(378.8)	15.64	10.10	738.1	92.77	$17.8{\cdot}10^6$(152.3)
hgen5-v125-s281703058	237,125	$8.1{\cdot}10^6$(129.6)	10.57	4.68	380.5	151.4	$6.4{\cdot}10^6$(54.82)
hgen5-v125-s486906609	580,125	$19.9{\cdot}10^6$(318.2)	24.57	7.54	143.5	81.64	$7.9{\cdot}10^6$(68.26)
hgen5-v125-s821831669	33,875	$1.1{\cdot}10^6$(18.17)	0.22	9.45	97.10	31.89	655,591(5.553)
hgen5-v150-s1806439773	$1.8{\cdot}10^6$	$72.7{\cdot}10^6$(1,201)	1.17	25.35	2,517	877.4	$46.3{\cdot}10^6$(398.8)
hgen5-v150-s1820487564	$1.2{\cdot}10^6$	$51.1{\cdot}10^6$(825.7)	31.98	57.67	1,032	1,230	$38.0{\cdot}10^6$(327.8)
hgen5-v150-s2035743477	*	*	26.00	44.54	27.26	746.4	$193{\cdot}10^6$(1,669)
hgen5-v150-s252403245	187,485	$7.7{\cdot}10^6$(123.4)	12.09	6.67	364.9	177.8	$4.5{\cdot}10^6$(38.46)
hgen5-v150-s378059954	447,270	$18.4{\cdot}10^6$(296.0)	0.56	44.28	3,376	1,525	$14.1{\cdot}10^6$(121.1)
hgen5-v175-s1398691205	*	*	422.9	247.8	*	*	*
hgen5-v175-s1578166233	*	*	137.3	664.1	*	1,472	$118{\cdot}10^6$(1,034)
hgen5-v175-s382577122	*	*	198.8	167.0	*	*	*
hgen5-v175-s704839520	$3.4{\cdot}10^6$	$163.4{\cdot}10^6$(2,714)	35.62	157.3	*	471.3	$127{\cdot}10^6$(1,105)
hgen5-v175-s736195112	568,732	$27.2{\cdot}10^6$(440.5)	16.34	549.6	*	171.7	$11.8{\cdot}10^6$(102.2)

Table VII. Results on Velev's microprocessor verification benchmarks $SSS-SAT-1.0$ ($2dlx\_cc\_mc\_ex\_bp\_f2\_bug*$) [38].

	UnitWalk+incBinSat+WalkSAT		dlmsat2	oksolver	saturn	usat10	WalkSat	limmat	zchaff
	substitutions	flips(time)	flips(time)	time	time	time	flips(time)	visits(time)	time
010	168,592	3,999(7.781)	57,537(8.31)	106.57	41.12	*	437,865(45.65)	466,349(1.12)	1.11
011	594,799	5,560(24.90)	374,383(12.98)	515.36	89.70	*	728,098(65.45)	$4.5{\cdot}10^6$(3.09)	1.13
012	213,115	4,484(20.90)	65,255(7.78)	200.68	*159.04	*	391,476(46.23)	$2.1{\cdot}10^6$(2.29)	4.18
013	$7.3{\cdot}10^6$	41,769(309.1)	*	*	*1652.72	*	*	$18{\cdot}10^6$(11.15)	4.94
014	304,678	5,568(14.51)	248,280(22.99)	115.01	77.45	*	204,959(24.31)	$1.9{\cdot}10^6$(2.29)	5.75
015	$6.8{\cdot}10^6$	43,958(394.7)	*	*	*	*	*	$52{\cdot}10^6$(50.05)	14.03
016	$4.8{\cdot}10^6$	27,604(205.0)	*	*	*	*	*	$381{\cdot}10^6$(143.55)	12.13
017	$1.4{\cdot}10^6$	10,444(66.94)	387,842(18.51)	*	35.02	*	$1.2{\cdot}10^6$(130.3)	$4.5{\cdot}10^6$(3.68)	1.45
018	$1.8{\cdot}10^6$	11,976(78.42)	*	*	*	*	*	$475{\cdot}10^6$(169.85)	4.69
019	$6.1{\cdot}10^6$	35,650(259.4)	*	*	*	*	*	$24{\cdot}10^6$(12.15)	5.14

References

1. Asahiro, F., K. Iwama, and E. Miyano: 1996, 'Random Generation of Test Instances with Controlled Attributes'. In [17].
2. Culberson, J., I. P. Gent, and H. H. Hoos: 2000, 'On the Probabilistic Approximate Completeness of WalkSAT for 2-SAT'. Technical Report APES-15A-2000, Department of Computer Science, University of Strathclyde.
3. Dantsin, E.: 1981, 'Two propositional proof systems based on the splitting method (in Russian)'. *Zapiski Nauchnykh Seminarov LOMI* **105**, 24–44. English translation: *Journal of Soviet Mathematics*, 22(3):1293–1305, 1983.
4. Dantsin, E., A. Goerdt, E. A. Hirsch, R. Kannan, J. Kleinberg, C. Papadimitriou, P. Raghavan, and U. Schöning: 2002, 'A deterministic $(2-2/(k+1))^n$ algorithm for k-SAT based on local search'. *Theoretical Computer Science*. To appear.
5. Dantsin, E., E. A. Hirsch, S. Ivanov, and M. Vsemirnov: 2001, 'Algorithms for SAT and Upper Bounds on Their Complexity'. Technical Report 01-012, ECCC. Electronic address: `ftp://ftp.eccc.uni-trier.de/pub/eccc/reports/2001/TR01-012/index.html`. A Russian version appears in *Zapiski Nauchnykh Seminarov POMI*, vol. 277, pp. 14–46, 2001.
6. Gent, I. P. and T. Walsh: 1993, 'Towards an Understanding of Hill-climbing Procedures for SAT'. In: *Proceedings of the 11th National Conference on Artificial Intelligence, AAAI'93*. pp. 28–33.
7. Gent, I. P. and T. Walsh: 1995, 'Unsatisfied variables in local search'. In: J. Hallam (ed.): *Hybrid Problems, Hybrid Solutions*. IOS Press, pp. 73–85.
8. Gu, J.: 1992, 'Efficient local search for very large-scale satisfiability problems'. *ACM SIGART Bulletin* **3**(1), 8–12.
9. Gu, J. and Q.-P. Gu: 1994, 'Average Time Complexity of The SAT1.2 Algorithm'. In: *Proceedings of the 5th Annual International Symposium on Algorithms and Computation, ISAAC'94*, Vol. 834 of *Lecture Notes in Computer Science*. pp. 146–154.
10. Gu, J., P. W. Purdom, J. Franco, and B. W. Wah: 1997, 'Algorithms for Satisfiability (SAT) Problem: A Survey'. In: D. Du, J. Gu, and P. M. Pardalos (eds.): *Satisfiability Problem: Theory and Applications (DIMACS Workshop March 11-13, 1996)*, Vol. 35 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. AMS, pp. 19–152.
11. Hirsch, E. A.: 2000, 'SAT Local Search Algorithms: Worst-Case Study'. *Journal of Automated Reasoning* **24**(1-2), 127–143.
12. Hirsch, E. A. and A. Kojevnikov: 2001, 'UnitWalk: A new SAT solver that uses local search guided by unit clause elimination'. PDMI preprint 9/2001, Steklov Institute of Mathematics at St.Petersburg.
13. Hoos, H. H.: 1998, 'Stochastic Local Search — Method, Models, Applications'. Ph.D. thesis, Department of Computer Science, Darmstadt University of Technology.
14. Hoos, H. H.: 1999, 'On the run-time behaviour of stochastic local search algorithms for SAT'. In: *Proceedings of the 16th National Conference on Artificial Intelligence, AAAI'99*. pp. 661–666.
15. Hoos, H. H. and T. Stützle: 2000a, 'Local search algorithms for SAT: An empirical evaluation'. *Journal of Automated Reasoning* **24**(4), 421–481.
16. Hoos, H. H. and T. Stützle: 2000b, 'SATLIB: An Online Resource for Research on SAT'. In: *Highlights of Satisfiability Research in the Year 2000*, Vol. 63 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, pp. 283–292.

17. Johnson, D. S. and M. A. Tricks (eds.): 1996, *Cliques, Coloring and Satisfiability*, Vol. 26 of *DIMACS Series on Discrete Mathematics and Theoretical Computer Science*. AMS.
18. Kamath, A. P., N. K. Karmarkar, K. G. Ramakrishnan, and M. G. C. Resende: 1992, 'A continuous approach to inductive inference'. *Mathematical Programming* **57**, 215–238.
19. Koutsoupias, E. and C. H. Papadimitriou: 1992, 'On the greedy algorithm for satisfiability'. *Information Processing Letters* **43**(1), 53–55.
20. Larrabee, T.: 1992, 'Test Pattern Generation Using Boolean Satisfiability'. *IEEE Transactions on Computer-Aided Design* **11**(1), 6–22.
21. Li, C. M. and Anbulagan: 1997, 'Heuristics based on unit propagation for satisfiability problems'. In: *Proceedings of the International Joint Conference on Artificial Intelligence*. pp. 366–371.
22. Luckhardt, H.: 1984, 'Obere Komplexitätsschranken für TAUT-Entscheidungen'. In: *Proceedings of Frege Conference 1984, Schwerin*. pp. 331–337.
23. McAllester, D., B. Selman, and H. Kautz: 1997, 'Evidence in invariants for local search'. In: *Proceedings of the 14th National Conference on Artificial Intelligence, AAAI'97*. pp. 321–326.
24. Monien, B. and E. Speckenmeyer: 1985, 'Solving satisfiability in less then 2^n steps'. *Discrete Applied Mathematics* **10**, 287–295.
25. Moskewicz, M. W., C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik: 2001, 'Chaff: Engineering an Efficient SAT Solver'. In: *Proceedings of the 38th Design Automation Conference (DAC'01)*. pp. 530–535.
26. Papadimitriou, C. H.: 1991, 'On selecting a satisfying truth assignment'. In: *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science, FOCS'91*. pp. 163–169.
27. Paturi, R., P. Pudlák, M. E. Saks, and F. Zane: 1998, 'An improved exponential-time algorithm for k-SAT'. In: *Proceedings of the 39th Annual IEEE Symposium on Foundations of Computer Science, FOCS'98*. pp. 628–637.
28. Paturi, R., P. Pudlák, and F. Zane: 1997, 'Satisfiability coding lemma'. In: *Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science, FOCS'97*. pp. 566–574.
29. Prestwich, S. D.: 2001, 'Local Search and Backtracking vs Non-Systematic Backtracking'. In: *AAAI 2001 Fall Symposium on Using Uncertainty within Computation*. To appear.
30. Schöning, U.: 1999, 'A probabilistic algorithm for k-SAT and constraint satisfaction problems'. In: *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science, FOCS'99*. pp. 410–414.
31. Schuler, R., U. Schöning, and O. Watanabe: 2001, 'An improved randomized algorithm for 3-SAT'. Technical Report TR-C146, Dept. of Mathematical and Computing Sciences, Tokyo Inst. of Tech.
32. Schuurmans, D. and F. Southey: 2000, 'Local search characteristics of incomplete SAT procedures'. In: *Proc. of AAAI'2000*. pp. 297–302.
33. Selman, B., H. A. Kautz, and B. Cohen: 1994, 'Noise strategies for improving local search'. In: *Proceedings of the 12th National Conference on Artificial Intelligence, AAAI'94*. pp. 337–343.
34. Selman, B., H. Levesque, and D. Mitchell: 1992, 'A new method for solving hard satisfiability problems'. In: *Proceedings of the 10th National Conference on Artificial Intelligence, AAAI'92*. pp. 440–446.

35. Silva, J. and K. Sakallah: 1996, 'GRASP – A New Search Algorithm for Satisfiability'. Technical Report CSE-TR-292-96.
36. Simon, L., D. L. Berre, and E. A. Hirsch: 2002, 'SAT-2002 Competition Report'. Submitted to this volume.
37. Simon, L. and P. Chatalic: 2001, 'SATEx: a Web-based Framework for SAT Experimentation'. In: H. Kautz and B. Selman (eds.): *Proceedings of the Workshop on Theory and Applications of Satisfiability Testing (SAT2001)*. Boston University, Massachusetts, USA.
38. Velev, M. N. and R. E. Bryant: 2001, 'Effective Use of Boolean Satisfiability Procedures in the Formal Verification of Superscalar and VLIW Microprocessors'. In: *Proceedings of the 38th Design Automation Conference (DAC'01)*. pp. 226–231.
39. Zhang, H.: 1993, 'A decision procedure for propositional logic'. *Assoc. Automated Reasoning Newslett.* **22**, 1–3.
40. Zhang, H. and M. Stickel: 2000, 'Implementing the Davis-Putnam method'. *Journal of Automated Reasoning* **24**(1-2), 277–296.
41. Zheng, L. and P. J. Stuckey: 2002, 'Improving SAT Using 2SAT'. In: M. J. Oudshoorn (ed.): *Proceedings of the Twenty-Fifth Australasian Computer Science Conference (ACSC2002)*. Melbourne, Australia.