PDMI PREPRINT — 09/2001

UNITWALK: A NEW SAT SOLVER THAT USES LOCAL SEARCH GUIDED BY UNIT CLAUSE ELIMINATION[1]

Edward A. HIRSCH[2]
Russian Academy of Sciences
Steklov Mathematical Institute
St. Petersburg Department
27 Fontanka, St. Petersburg, 191011
Russia

E-mail: `hirsch@pdmi.ras.ru`

Arist KOJEVNIKOV
St.Petersburg State University
Department of Mathematics and Mechanics
St.Petersburg
Russia

E-mail: `arist@logic.pdmi.ras.ru`

December 14, 2001

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. For instance, compared to other contemporary incomplete SAT solvers, it dominates on satisfiable instances of "`aim`" [AIM96], "`ssa`" [Lar92], "`par`" [JT96] and "`ii`" [KKRR92] series. It is also able to solve some of the "`qg`" [ZH94] instances, all satisfiable Velev's microprocessor verification instances from series "`SSS.1.0`" [Velb], "`SSS.1.0a`" [Velc] and "`SSS-SAT.1.0`" [Vela], and most instances from "`VLIW-SAT.1.1`" [Veld].

Our algorithm is inspired by two randomized algorithms having the best current worst-case upper bounds ([PPSZ98, PPZ97] and [Sch99, SSW01]). In our paper 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 [SLM92] and WalkSAT [SKC94]) 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 first computational experiments. We also prove that our algorithm is probabilistically approximately complete (PAC).

---

[1] An extended abstract of this paper appeared in Proceedings of the 7th International Conference on Principles and Practice of Constraint Programming, CP'01.

[2] Partially supported by grant #1 of the 6th RAS contest-expertise of young scientists projects (1999) and a grant from NATO.

1 Introduction

1.1 Background

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.

Weakly exponential worst-case upper bounds. "Weakly exponential" upper bounds on the worst-case running time of SAT algorithms are 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 [Dan81, Luc84, MS85]. Currently, the best known deterministic algorithm for k-SAT (i.e., when the length of a clause is bounded by k) has the bound approaching $p(n)(2-2/(k+1))^n$ [DGH$^+$02, DGHS00]; for 3-SAT the cited papers give the bound $p(n)1.481^n$. For general SAT, no "less-than-2^n" algorithm is known, however, there are algorithms achieving non-trivial bounds w.r.t. the total number l of variable occurrences. The best current bound is $p(l)1.074^l$ [Hir00a]. Randomized algorithms achieve even better bounds: $p(n)(2-2/k)^n$ for k-SAT and $p(n)1.3303^n$ for 3-SAT [Sch99, SSW01]. 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 [PPSZ98]. Non-trivial bounds are also known for other $\mathcal{NP}$-complete subproblems of SAT [Hir00c, KL99], for MAX-SAT [BR99], MAX-2-SAT [GHNR00] and other $\mathcal{NP}$-complete problems [BE95, Rob86]. Regarding the details, see the survey [DHIV01].

"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 [Hir00b], average-case bounds [GG94, KP92], or properties related to their completeness [CGH00, Hoo98, Hoo99]. Some of these algorithms are surveyed in [GPFW97, HS00a].

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 quite successful SAT solvers based on complete algorithms (e.g., rel_sat [BS97], satz [LA97], SATO [Zha93, ZS00]; see comprehensive experimental results at SAT-Ex web-site

[Sim]). However, in general, solvers based on incomplete algorithms are faster.

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 [Gu92] and Selman, Levesque and Mitchell [SLM92] and theoretical work by Koutsoupias and Papadimitriou [KP92, Pap91]). 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 [SLM92], GWSAT (aka GSAT+w) [SKC94], HSAT [GW93], HWSAT [GW95], SDF [SS00], WalkSAT [SKC94] and various algorithms within WalkSAT framework such as WalkSAT/TABU, Novelty, R-Novelty [MSK97], Novelty+ and R-Novelty+ [Hoo99].

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* [Hoo99] (e.g., GSAT, WalkSAT/TABU, Novelty, R-Novelty [Hoo98, Hoo99]). Other algorithms are *probabilistically approximately complete* (*PAC*) [Hoo99], 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 [Hoo98, Hoo99]).

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 [Pre01]. Our local search algorithm also uses unit clause elimination, though in a different way.

1.2 Our algorithm

Procedure of Paturi, Pudlák, Saks, and Zane. In this paper, we suggest a new incomplete SAT algorithm UnitWalk. Similarly to the experimentally best incomplete algorithms from WalkSAT family [MSK97] and the theoretically best algorithms by Schöning [Sch99] and Schuler, Schöning and Watanabe [SSW01], 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 [PPSZ98].

The core of the algorithm in [PPSZ98] is the following procedure [PPZ97]. 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 [PPZ97] 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 ([PPZ97], see also [DHIV01] for a simpler construction).

Paturi, Pudlák, Saks and Zane [PPSZ98] 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).

Our results. In [PPZ97, PPSZ98], 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 [DHIV01]. 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 [MSK97].

We prove that our algorithm is probabilistically approximately complete. We also implemented a SAT solver based on our algorithm. This "rough" implementation (not optimized in any sense) already shows experimental results that are competitive with other SAT solvers. We provide the experimental results we obtained so far on some of the benchmarks available from SATLIB web-site [HS] (cf. [HS00b]) and on Velev's microprocessor verification benchmarks [Vela, Velb, Velc, Veld].

The paper is organized as follows. In Section 2 we give definitions and describe notation related to Boolean satisfiability and SAT algorithms. Our 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. In Section 4 we present experimental data describing the execution of our algorithm on various benchmarks. Finally, in Section 5 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[1] 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.

[1] 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.

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 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*) [Hoo99] 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 Algorithm UnitWalk

3.1 Description of the algorithm

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 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 [PPZ97, PPSZ98].

A more formal description of the algorithm follows.

Algorithm UnitWalk.

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".

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 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 $b(n) + (1 - b(n))b(n) + (1 - b(n))^2 b(n) + \ldots = 1$. $\square$

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 [MSK97]. 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 WalkSAT family [MSK97], 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 it 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 [Hoo98, Hoo99].

2. The experimental data presented in the next section show that UnitWalk makes substantially less flips than WalkSAT/TABU on almost all instances.

4 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 or scale the data to an available computer. 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 [Sch99, SSW01]). See [HS00a] 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.

Benchmarks. The main source where we took benchmarks was SATLIB [HS] (cf. [HS00b]); see references there for the description and original sources of these benchmarks. First of all, we selected for our experiments the benchmarks on which experimental data for other algorithms was available. In addition, we ran our algorithm on Velev's microprocessor verification benchmarks [Vela, Velb, Velc, Veld]. Clearly, we selected only satisfiable benchmarks from all these series.

Our experiments and other sources of data. All our experiments were made on a 466MHz Intel Celeron Pentium II running under Linux. The DIMACS hardware benchmark program *dfmax r500.5* [JT96] takes 43,71 seconds on the machine.

The data for other algorithms is partially taken from other sources [Hoo98, HS00a, Pre01]. In particular, [Hoo98, HS00a] gives comprehensive data concerning the number of flips of HWSAT, GWSAT, GWSAT/TABU, WalkSAT/TABU and other algorithms from WalkSAT family. However, 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 [Pre01] where the experiments were made on a 400 Mhz Intel Pentium II.

In most of the experiments of [Hoo98, HS00a], the number of restarts (from a new initial assignment) was set to one. For almost all our experiments with UnitWalk, we have made the same thing: `MAX_TRIES`$(F) = 1$ and `MAX_PERIODS`$(F) = +\infty$: fortunately, for the benchmarks considered in the tables below, it always returned an answer. 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). Therefore, we have set `MAX_FLIPS` $= 10,000,000$ for `par8`, `par16-c` and `ais12` series and `MAX_FLIPS` $= 1,000,000$ on other benchmarks.

Modified versions of our algorithm. We have also collected a small amount of data about the experimental study of a modified version of our algorithm that includes a preprocessing step similar to the one used in [PPSZ98]. Namely, we extended the original formulas by resolvents of their clauses. Since RAM is an expensive resource, we limited preprocessing to adding resolvents of size one, two or three; the obtained resolvents were not processed again, i.e., we did not add resolvents of resolvents etc.

Another incomplete direction of research is setting finite values for `MAX_PERIODS` (and setting `MAX_TRIES` to $+\infty$ so that the algorithm is allowed to run until if finds a solution).

We have not yet had much CPU time for the thorough investigation of these two research directions, however, we present the data collected so far.

Description of tables. The experimental data are presented in the tables given in the appendix. The column "mean" contains the average number of substitutions or flips made until a solution is reached (note that UnitWalk was allowed to run until the solution is reached, i.e., there were no failures). The column "median" contains the number t of flips (resp., substitutions) such that in 50% of the experiments a solution is reached after at most t flips (substitutions). We also present CPU time in seconds, where available. Where we have data concerning running UnitWalk with a finite `MAX_PERIODS` setting, we include it with a line marked `P=cn` meaning that `MAX_PERIODS` was set to cn where n is the number of variables in the input formula. Similarly, where we have data about the modified version of the algorithm including the preprocessing step, we mark the corresponding line with `R=c`, where c is the maximum size of a resolvent. Finally, an asterisk means that the corresponding algorithm did not find a solution in some of the runs.

Conclusion concerning our experiments. Our experiments show that even the current "rough" implementation of UnitWalk is very competitive on many benchmarks. Note that UnitWalk makes substantially less flips on the average than other algorithms on `par`, `aim`, `ii`, `jnh`, `ssa`, `logistics.d`, `bw_large.a`, `bw_large.b` and some of the Bejing benchmarks. On `par8-5`, `par16-5-c`, `aim` and `ssa` it even makes less substitutions than the number of flips made by the other algorithms. Regarding CPU time, UnitWalk "wins" on `aim`, `par8-5`, `par16-5-c` and some of the Bejing benchmarks. It also shows competitive results on many other benchmarks. Note that UnitWalk solved some of Zhang's QG instances [ZH94], all satisfiable Velev's `SSS-SAT.1.0` [Vela], `SSS.1.0` [Velb] and `SSS.1.0a` [Velc] instances, and most of `VLIW-SAT.1.1` [Veld] instances.

The benchmarks that appeared the most hard (but still solvable) for UnitWalk were `logistics.a`, `bw_large.c`, `ais12` and some of the microprocessor verification benchmarks. Adding resolvents of size at most three helped significantly in solving `logistics.a`. The same technique improved the number of flips and substitutions on `uf100-430` benchmarks; however, the preprocessing step takes a substantial amount of CPU time which does not look reasonable on relatively easy benchmarks. We could solve `bw_large.c` only by adding adding resolvents of size at most two and limiting the `MAX_PERIODS` to $0.5n$. These two techniques, however, did not give substantial improvement on `4blocks` benchmark, `hanoi` and `f` (`LRAN`) series, and Velev's microprocessor verification benchmarks.

UnitWalk was unable to solve `f` (`LRAN`), `hanoi` and `bmc` series, some of `qg`, `9vliw` and large `par` benchmarks. Unfortunately, we did not throughly tested it on large `uf` benchmarks. Altough we know it is able to solve `uf225-960` benchmarks, the limit of its ability to solve benchmarks of this type (with or without adding resolvents) is currently unknown.

5 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). Surprisingly, even the first implementation of our algorithm shows very competitive results; the experimental data we present in the paper show that our algorithm dominates on some sets of benchmarks and is able to solve some very hard SAT instances. It shows substantially worse behaviour on some other benchmarks. In these cases sometimes modifications of our algorithm (like adding resolvents or limiting the number of periods) help.

The following directions of research seem promising:

1. Design an *efficient* implementation of UnitWalk.

2. Continue experiments with including polynomial-time preprocessing (such as adding resolvents) into UnitWalk and with setting finite values to `MAX_PERIODS`.

3. Many of the complete algorithms based on the method of Davis, Logemann and Loveland [DLL62] prune search space using various rules of formula transformation (unit clause elimination is just one of them). Such transformation rules are also widely used in the deterministic SAT and MAX-SAT algorithms having the best current worst-case upper bounds ([BR99, DGH^{+}02, DGHS00, GHNR00, Hir00a], see [Kul98] for a detailed treatment). It would be interesting to add these transformation rules in the same way as we use the elimination of unit clauses; see [DHIV01] about the application of this idea to the procedure of Paturi, Pudlák, Saks and Zane [PPZ97, PPSZ98].

4. It is known [SSW01] that changing the distribution of initial assignment (from a uniform distribution to a different one) results in improved worst-case upper bounds on the running time of a local search algorithm. Also it is easy to see that Papadimitriou's algorithm for 2-CNF [Pap91] can choose initial assignment from the set of only two assignments (all-`True` and all-`False`) without worsening its worst-case running time. On the other hand, it is shown in [DHIV01] that one can choose a random permutation from a set of naturally constructed $O(n^3)$ permutations without worsening the upper bound on the running time of the procedure of Paturi, Pudlák and Zane [PPZ97] for 3-CNF (this also helps in derandomizing this procedure). Therefore, we conjecture that changing the distributions of permutations and initial assignments may improve the behavior of our algorithm at least on certain benchmarks.

5. A formula in CNF can be considered either as a set or as a multiset of clauses. In particular, the random choice of a unit clause can either take into account identical unit clauses or treat them as the same clause (resulting in a different distribution). In the current implementation, we treat a formula as a multiset. It would be interesting to make experiments with the other version.

6. Design a complete algorithm based on UnitWalk. (See [DHIV01] for survey of related derandomization issues).

7. Proving upper and lower bounds on the running time of UnitWalk seems a challenging problem.

The C source code of an implementation of UnitWalk is available from `http://logic.pdmi.ras.ru/~arist/UnitWalk/`.

References

[AIM96] F. Asahiro, K. Iwama, and E. Miyano. Random generation of test instances with controlled attributes. In [JT96], 1996.

[BE95] R. Beigel and D. Eppstein. 3-coloring in time $o(1.3446^n)$: a no-MIS algorithm. In *Proceedings of the 36nd Annual IEEE Symposium on Foundations of Computer Science, FOCS'95*, pages 444–452, 1995.

[BR99] N. Bansal and V. Raman. Upper bounds for MaxSat: Further improved. In A. Aggarwal and C. Pandu Rangan, editors, *Proceedings of the 10th Annual Conference on Algorithms and Computation, ISAAC'99*, volume 1741 of *Lecture Notes in Computer Science*, pages 247–258. Springer-Verlag, 1999.

[BS97] R. Bayardo and R. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the 14th National Conference on Artificial Intelligence, AAAI'97*, pages 203–208, 1997.

[CGH00] J. Culberson, I. P. Gent, and H. H. Hoos. On the probabilistic approximate completeness of WalkSAT for 2-SAT. Technical Report APES-15A-2000, Department of Computer Science, University of Strathclyde, 2000.

[Dan81] E. Dantsin. Two propositional proof systems based on the splitting method (in Russian). *Zapiski Nauchnykh Seminarov LOMI*, 105:24–44, 1981. English translation: *Journal of Soviet Mathematics*, 22(3):1293–1305, 1983.

[DGH$^+$02] E. Dantsin, A. Goerdt, E. A. Hirsch, R. Kannan, J. Kleinberg, C. Papadimitriou, P. Raghavan, and U. Schöning. A deterministic $(2-2/(k+1))^n$ algorithm for k-SAT based on local search. *Theoretical Computer Science*, 2002. To appear.

[DGHS00] E. Dantsin, A. Goerdt, E. A. Hirsch, and U. Schöning. Deterministic algorithms for k-SAT based on covering codes and local search. In *Proceedings of the 27th International Colloquium on Automata, Languages and Programming, ICALP'00*, volume 1853 of *Lecture Notes in Computer Science*, pages 236–243. Springer-Verlag, 2000.

[DHIV01] E. Dantsin, E. A. Hirsch, S. Ivanov, and M. Vsemirnov. Algorithms for SAT and upper bounds on their complexity. Technical Report 01-012, ECCC, 2001. 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.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, July 1962.

[GG94] J. Gu and Q.-P. Gu. Average time complexity of the SAT1.2 algorithm. In *Proceedings of the 5th Annual International Symposium on Algorithms and Computation, ISAAC'94*, volume 834 of *Lecture Notes in Computer Science*, pages 146–154. Springer, 1994.

[GHNR00] J. Gramm, E. A. Hirsch, R. Niedermeier, and P. Rossmanith. New worst-case upper bounds for MAX-2-SAT with application to MAX-CUT. Technical Report 00-037, ECCC, June 2000. Electronic address: `ftp://ftp.eccc.uni-trier.de/pub/eccc/reports/2000/TR00-037/index.html`. To appear in *Discrete Applied Mathematics*.

[GPFW97] J. Gu, P. W. Purdom, J. Franco, and B. W. Wah. Algorithms for satisfiability (SAT) problem: A survey. In D. Du, J. Gu, and P. M. Pardalos, editors, *Satisfiability Problem: Theory and Applications (DIMACS Workshop March 11-13, 1996)*, volume 35 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 19–152. AMS, 1997.

[Gu92] J. Gu. Efficient local search for very large-scale satisfiability problems. *ACM SIGART Bulletin*, 3(1):8–12, 1992.

[GW93] I. P. Gent and T. Walsh. Towards an understanding of hill-climbing procedures for SAT. In *Proceedings of the 11th National Conference on Artificial Intelligence, AAAI'93*, pages 28–33, 1993.

[GW95] I. P. Gent and T. Walsh. Unsatisfied variables in local search. In J. Hallam, editor, *Hybrid Problems, Hybrid Solutions*, pages 73–85. IOS Press, 1995.

[Hir00a] E. A. Hirsch. New worst-case upper bounds for SAT. *Journal of Automated Reasoning*, 24(4):397–420, 2000.

[Hir00b] E. A. Hirsch. SAT local search algorithms: Worst-case study. *Journal of Automated Reasoning*, 24(1-2):127–143, February 2000.

[Hir00c] E. A. Hirsch. Separating sings in the propositional satisfiability problem. *Journal of Mathematical Sciences*, 98(4):442–463, 2000.

[Hoo98] H. H. Hoos. *Stochastic Local Search — Method, Models, Applications*. PhD thesis, Department of Computer Science, Darmstadt University of Technology, 1998.

[Hoo99] H. H. Hoos. On the run-time behaviour of stochastic local search algorithms for SAT. In *Proceedings of the 16th National Conference on Artificial Intelligence, AAAI'99*, pages 661–666, 1999.

[HS] H. H. Hoos and T. Stützle. SATLIB. A collection of SAT-related resources, `http://www.satlib.org/`.

[HS00a] H. H. Hoos and T. Stützle. Local search algorithms for SAT: An empirical evaluation. *Journal of Automated Reasoning*, 24(4):421–481, 2000.

[HS00b] H. H. Hoos and T. Stützle. SATLIB: An online resource for research on SAT. In *Highlights of Satisfiability Research in the Year 2000*, volume 63 of *Frontiers in Artificial Intelligence and Applications*, pages 283–292. IOS Press, 2000.

[JT96] D. S. Johnson and M. A. Tricks, editors. *Cliques, Coloring and Satisfiability*, volume 26 of *DIMACS Series on Discrete Mathematics and Theoretical Computer Science*. AMS, 1996.

[KKRR92] A. P. Kamath, N. K. Karmarkar, K. G. Ramakrishnan, and M. G. C. Resende. A continuous approach to inductive inference. *Mathematical Programming*, 57:215–238, 1992.

[KL99] O. Kullmann and H. Luckhardt. Algorithms for SAT/TAUT decision based on various measures. Preprint, 71 pages, available from `http://www.cs.toronto.edu/\~{}kullmann`, February 1999.

[KP92] E. Koutsoupias and C. H. Papadimitriou. On the greedy algorithm for satisfiability. *Information Processing Letters*, 43(1):53–55, 1992.

[Kul98] Oliver Kullmann. Heuristics for SAT algorithms: Searching for some foundations. Submitted to Discrete Applied Mathematics, 23 pages, September 1998.

[LA97] C. M. Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 366–371, 1997.

[Lar92] T. Larrabee. Test pattern generation using boolean satisfiability. *IEEE Transactions on Computer-Aided Design*, 11(1):6–22, January 1992.

[Luc84] H. Luckhardt. Obere Komplexitätsschranken für TAUT-Entscheidungen. In *Proceedings of Frege Conference 1984, Schwerin*, pages 331–337. Akademie-Verlag Berlin, 1984.

[MS85] B. Monien and E. Speckenmeyer. Solving satisfiability in less then 2^n steps. *Discrete Applied Mathematics*, 10:287–295, 1985.

[MSK97] D. McAllester, B. Selman, and H. Kautz. Evidence in invariants for local search. In *Proceedings of the 14th National Conference on Artificial Intelligence, AAAI'97*, pages 321–326, 1997.

[Pap91] C. H. Papadimitriou. On selecting a satisfying truth assignment. In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science, FOCS'91*, pages 163–169, 1991.

[PPSZ98] R. Paturi, P. Pudlák, M. E. Saks, and F. Zane. An improved exponential-time algorithm for k-SAT. In *Proceedings of the 39th Annual IEEE Symposium on Foundations of Computer Science, FOCS'98*, pages 628–637, 1998.

[PPZ97] R. Paturi, P. Pudlák, and F. Zane. Satisfiability coding lemma. In *Proceedings of the 38th Annual IEEE Symposium on Foundations of Computer Science, FOCS'97*, pages 566–574, 1997.

[Pre01] S. D. Prestwich. Local search and backtracking vs non-systematic backtracking. In *AAAI 2001 Fall Symposium on Using Uncertainty within Computation*, 2001. To appear.

[Rob86] J. M. Robson. Algorithms for maximum independent set. *Journal of Algorithms*, 7:425–440, 1986.

[Sch99] U. Schöning. 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*, pages 410–414, 1999.

[Sim] L. Simon. The SAT-Ex Site. A collection of experimental results about complete SAT solvers, `http://www.lri.fr/~simon/satex/satex.php3`.

[SKC94] B. Selman, H. A. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of the 12th National Conference on Artificial Intelligence, AAAI'94*, pages 337–343, 1994.

[SLM92] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the 10th National Conference on Artificial Intelligence, AAAI'92*, pages 440–446, 1992.

[SS00] D. Schuurmans and F. Southey. Local search characteristics of incomplete SAT procedures. In *Proc. of AAAI'2000*, pages 297–302, 2000.

[SSW01] R. Schuler, U. Schöning, and O. Watanabe. An improved randomized algorithm for 3-SAT. Technical Report TR-C146, Dept. of Mathematical and Computing Sciences, Tokyo Inst. of Tech., 2001.

[Vela] M. N. Velev. Benchmark suite SSS-SAT.1.0. Available from: `http://www.ece.cmu.edu/~mvelev/`.

[Velb] M. N. Velev. Benchmark suite SSS.1.0. Available from: `http://www.ece.cmu.edu/~mvelev/`.

[Velc] M. N. Velev. Benchmark suite SSS.1.0a. Available from: `http://www.ece.cmu.edu/~mvelev/`.

[Veld] M. N. Velev. Benchmark suite VLIW-SAT.1.1. Available from: `http://www.ece.cmu.edu/~mvelev/`.

[ZH94] H. Zhang and J. Hsiang. Solving open quasigroup problems by propositional reasoning. In *Proceedings of International Computer Symposium, Hsinchu, Taiwan*, 1994.

[Zha93] H. Zhang. A decision procedure for propositional logic. *Assoc. Automated Reasoning Newslett.*, 22:1–3, March 1993.

[ZS00] H. Zhang and M. Stickel. Implementing the Davis-Putnam method. *Journal of Automated Reasoning*, 24(1-2):277–296, 2000.

Appendix

algorithm	substitutions		flips	
	mean	median	mean(time)	median
WalkSAT(0.3)			95,205(0.432)	65,566
WalkSAT/TABU(1)			177,169(1.053)	139,647
Novelty(0.45)			55,748(0.332)	46,366
R-Novelty(0.65)			45,220(0.259)	33,713
UnitWalk	39,863,811	30,393,396	1,235,891(920.7)	941,571
UnitWalk, `R=3`	2,385,832	1,761,984	53,057(464.1)	38,303

Table 1: Experimental data for `logistics.a` based on 100 tries.

algorithm	substitutions		flips	
	mean	median	mean(time)	median
WalkSAT(0.4)			472,513(3.013)	382,823
WalkSAT/TABU(2)			400,005( )	332,498
Novelty(0.35)			136,938(1.187)	111,814
R-Novelty(0.6)			1,115,760(12.63)	839,196
SDF			74,090(56.72)	36,139
UnitWalk	103,686	85,176	4,120(0.940)	3,814

Table 2: Experimental data for `logistics.d` based on 100 tries.

algorithm	substitutions		flips	
	mean	median	mean(time)	median
GWSAT(0.6)			26,994( )	18,653
GSAT/TABU(10)			10,635( )	6,977
WalkSAT(0.5)			19,158(0.093)	14,155
WalkSAT/TABU(3)			10,603( )	7,563
Novelty(0.4)			9,699(0.050)	7,093
R-Novelty(0.6)			6,911(0.039)	4,497
SDF			2,917(0.184)	2,173
IDB			(0.425)	
UnitWalk	15,174	11,676	1,215(0.159)	968

Table 3: Experimental data for `bw_large.a` based on 100 tries.

algorithm	substitutions		flips	
	mean	median	mean(time)	median
WalkSAT(0.35)			480,187(3.793)	426,364
WalkSAT/TABU(2)			152,104( )	107,586
Novelty(0.3)			203,813(2.005)	139,618
R-Novelty(0.55)			347,827(4.443)	187,184
SDF			38,927(6.440)	27,926
IDB			(7.750)	
UnitWalk	215,889	145,521	11,848(3.562)	7,847

Table 4: Experimental data for `bw_large.b` based on 100 tries.

algorithm	substitutions		flips	
	mean	median	mean(time)	median
WalkSAT(0.2)			$14.06 \cdot 10^6$(151.8)	$9.76 \cdot 10^6$
WalkSAT/TABU(2)			$2.52 \cdot 10^6$(38.21)	$0.84 \cdot 10^6$
Novelty			$6.4 \cdot 10^6$(200.8)	$1.12 \cdot 10^6$
UnitWalk, `R=2`, `P=0.5n`	$13.8 \cdot 10^6$	$9.3 \cdot 10^6$	544,469(2634)	373,783

Table 5: Experimental data for `bw_large.c` based on 10 tries.

algorithm	substitutions		flips	
	mean	median	mean(time)	median
GSAT			21,859(0.160)	15,419
GWSAT(0.6)			27,143( )	18,634
HWSAT			18,631( )	12,538
WalkSAT(0.2)			19,182(0.056)	13,345
WalkSAT/TABU(9)			12,338( )	8,388
Novelty(0.8)			4,318(0.012)	2,931
R-Novelty(0.65)			2,640(0.007)	1,579
SDF			3,691(0.033)	2,358
UnitWalk	5,311	3,781	863(0.011)	603

Table 6: Experimental data for `par8-5-c` based on 100 tries.

algorithm	substitutions		flips	
	mean	median	mean(time)	median
Novelty(0.8)			$2.0 \cdot 10^6$(3.875)	$1.1 \cdot 10^6$
R-Novelty(0.65)			271,632(0.577)	196,839
SDF			$1.0 \cdot 10^6$(19.11)	876,772
UnitWalk	13,078	8,329	1,022(0.022)	997

Table 7: Experimental data for `par8-5` based on 100 tries.

algorithm	substitutions		flips	
	mean	median	mean(time)	median
Novelty(0.8)			$85.9{\cdot}10^6$ (236.6)	$64.8{\cdot}10^6$
R-Novelty(0.65)			$49.9{\cdot}10^6$ (145.2)	$21.4{\cdot}10^6$
UnitWalk	$12.7{\cdot}10^6$	$9.3{\cdot}10^6$	$3.1{\cdot}10^6$ (33.85)	$1.0{\cdot}10^6$

Table 8: Experimental data for `par16-5-c` based on 100 tries.

algorithm	substitutions		flips	
	mean	median	mean(time)	median
GWSAT			6,532( )	3,398
WalkSAT			3,652(0.009)	1,981
WalkSAT/TABU			2,485( )	1,041
Novelty			15,699(0.047)	962
R-Novelty			1,536(0.005)	581
SDF			876(0.013)	515
UnitWalk	13,500	9,468	2,547(0.029)	1,862
UnitWalk, `R=3`	9,822	6,871	1,672(0.032)	1,170

Table 9: Experimental data for `uf100-430` based on 100 tries.

algorithm	substitutions		flips	
	mean	median	mean(time)	median
GWSAT(0.4)			64,167( )	43,128
GWSAT/TABU(10)			43,421( )	29,545
WalkSAT(0.4)			28,528(0.152)	19,291
*WalkSAT/TABU(20)				9,472
*Novelty(0.5)				140
*R-Novelty(0.8)			$1.2{\cdot}10^6$ (6.529)	99,880
SDF			4,645(0.169)	3,469
UnitWalk	56,370	40,317	2,875(0.229)	1,993

Table 10: Experimental data for `ais8` based on 250 tries.

algorithm	substitutions		flips	
	mean	median	mean(time)	median
GWSAT(0.4)			494,718( )	336,556
*GWSAT/TABU(10)				203,635
WalkSAT(0.4)			173,422(1.978)	122,283
*WalkSAT/TABU(20)				130,502
SDF			20,870(1.521)	15,231
UnitWalk	432,191	275,844	18,405(6.077)	11,348

Table 11: Experimental data for `ais10` based on 100 tries.

algorithm	substitutions		flips	
	mean	median	mean(time)	median
GWSAT(0.4)			494,718( )	336,556
*GWSAT/TABU(10)				203,635
WalkSAT(0.4)			173,422(1.978)	122,283
*WalkSAT/TABU(20)				130,502
SDF			20,870(1.521)	15,231
UnitWalk	$6.0{\cdot}10^6$	$4.6{\cdot}10^6$	222,967(142.5)	191,570

Table 12: Experimental data for `ais12` based on 100 tries.

algorithm	substitutions		flips	
	mean	median	mean(time)	median
*GSAT			169,366(7.095)	3,324
WalkSAT(0.5)			495(0.021)	319
Novelty(0.55)			5,593(0.180)	4,941
*R-Novelty(0.8)			3,166(0.010)	2,247
SDF			3,946(0.804)	3,119
UnitWalk	1,468	1,196	203(0.021)	193

Table 13: Experimental data for `ii8` based on 100 tries.

algorithm	substitutions		flips	
	mean	median	mean(time)	median
WalkSAT(0.5)			6,201(0.769)	4,541
*Novelty(0.45)			91,031(10.07)	27,163
*R-Novelty(0.7)			85,742(4.036)	22,469
SDF			7,180(1.346)	5,961
IDB			(0.305)	
UnitWalk	9,093	8,218	1.086(0.273)	815

Table 14: Experimental data for `ii16` based on 100 tries.

algorithm	substitutions		flips	
	mean	median	mean(time)	median
WalkSAT(0.5)			2,284(1.040)	1,763
*Novelty(0.6)			88,779(10.07)	20,608
*R-Novelty(0.7)			77,509(3.736)	18,671
IDB			(1.170)	
UnitWalk	5,989	4,440	854(0.227)	738

Table 15: Experimental data for `ii32` based on 100 tries.

algorithm	substitutions		flips	
	mean	median	mean(time)	median
*GSAT			291,729(2.585)	93,067
WalkSAT(0.5)			5,449(0.049)	3,910
Novelty(0.6)			3,329(0.031)	2,491
R-Novelty(0.8)			1,943(0.020)	1,201
SDF			732(0.039)	541
UnitWalk	3,625	1,627	484(0.026)	396

Table 16: Experimental data for `jnh` based on 100 tries.

algorithm	substitutions		flips	
	mean	median	mean(time)	median
WalkSAT(0.5)			45,181(0.171)	30,195
*Novelty(0.6)			650,156(7.024)	200,619
*R-Novelty(0.8)			$1{,}3 \cdot 10^6$(13.76)	431,859
SDF			27,238(2.146)	20,069
IDB			(0.310)	
UnitWalk	6,992	4,689	707(0.023)	694

Table 17: Experimental data for `ssa` based on 100 tries.

algorithm	substitutions		flips	
	mean	median	mean(time)	median
GSAT			16,857(0.100)	10,253
GWSAT			7,023( )	4,846
GSAT/TABU(10)			1,040( )	787
HWSAT			2,641( )	2,042
WalkSAT			3,896(0.008)	3,078
WalkSAT/TABU			61,393( )	1,265
Novelty			24,421(0.027)	934
R-Novelty			7,109(0.019)	739
SDF			773(0.010)	537
UnitWalk	1,521	1,076	261(0.002)	200

Table 18: Experimental data for `flat50-115` based on 1,000 tries.

algorithm	substitutions		flips	
	mean	median	mean(time)	median
GSAT			752,775(4.631)	158,381
WalkSAT			44,900(0.164)	31,728
Novelty			17,893(0.036)	2,217
R-Novelty			12,345(0.027)	1,893
UnitWalk	18,833	13,140	3,500(0.051)	2,448

Table 19: Experimental data for `flat100-239` based on 100 tries.

algorithm	substitutions		flips	
	mean	median	mean(time)	median
WalkSAT			254,188(0.562)	159,386
Novelty			130,359(0.296)	87,893
R-Novelty			731,825(1.429)	331,246
UnitWalk	208,719	173,972	46,096(0.499)	32,792

Table 20: Experimental data for `flat150-360` based on 100 tries.

benchmark	substitutions		flips	
	mean	median	mean(time)	median
qg5-11	777,304	777,304	35,198(3327)	35,198
qg3-08	86,016	86,016	6,539(1992)	6,539
qg2-07	40,240	29,155	3,363(8446)	2,394
qg1-07	60,745	52,136	4,791(6938)	4,050

Table 21: Experimental results for some benchmarks from `QG` family based on 10 tries.

algorithm	substitutions		flips	
	mean	median	mean(time)	median
*WalkSAT(0.5)			251,618(0.445)	76,501
*Novelty(0.5)			269,700(0.453)	140
*R-Novelty(0.8)			265,862(0.456)	89,428
*SDF			116,467(1.889)	23,281
UnitWalk	24,059	16,682	4,020(0.048)	2,798

Table 22: Experimental data for `aim100` based on 100 tries.

benchmark	substitutions		flips	
	mean	median	mean(time)	median
2bitadd_11	6,762	5,841	297(0.011)	281
2bitadd_12	4,566	4,248	254(0.009)	241
2bitcomp_5	358	250	45(0.001)	43
2bitmax_6	965	756	107(0.001)	101
3bitadd_31	753,820	649,264	4,778(7.164)	4,376
3bitadd_32	416,921	356,864	3,743(4.757)	3,495
3blocks	116,808	95,088	3,193(4.851)	2,574
e0ddr2-10-by-5-1	349,245	253,500	10,498(7.872)	9,978
e0ddr2-10-by-5-4	145,275	97,500	9,079(3.224)	8,941
enddr2-10-by-5-1	235,980	165,600	10,251(10.98)	9,873
enddr2-10-by-5-8	181,440	147,000	10,053(4.411)	9,869
ewddr2-10-by-5-1	161,320	109,000	10,375(10.59)	10,157
ewddr2-10-by-5-8	144,450	112,500	10,584(10.25)	10,444
4blocksb	1,195,232	1,005,320	22,020(208.2)	19,308
4blocks	2,316,448	2,316,448	37,886(501.7)	37,886

Table 23: Experimental results for Bejing test set. The number of tries varies from 1 to $1,000$.

benchmark	substitutions		flips	
	mean	median	mean(time)	median
dlx2_cc_a_bug3	$22.2 \cdot 10^6$	$17.4 \cdot 10^6$	$44,209(328.0)$	$34,079$
dlx2_cc_a_bug4	$8.5 \cdot 10^6$	$5.9 \cdot 10^6$	$17,000(131.7)$	$13,021$
dlx2_cc_a_bug5	$3.8 \cdot 10^6$	$1.7 \cdot 10^6$	$8,323(58.10)$	$4,844$
dlx2_cc_a_bug17	$2.5 \cdot 10^6$	$1.4 \cdot 10^6$	$5,496(47.52)$	$4,510$
dlx2_cc_a_bug38	$13.4 \cdot 10^6$	$4.1 \cdot 10^6$	$25,095(197.4)$	$10,072$
dlx2_cc_a_bug39	$4.4 \cdot 10^6$	$3.0 \cdot 10^6$	$10,197(64.12)$	$8,392$
dlx2_cc_a_bug40	$3.3 \cdot 10^6$	$1.4 \cdot 10^6$	$7,364(48.89)$	$4,350$
dlx2_cc_a_bug54	$13.1 \cdot 10^6$	$9.9 \cdot 10^6$	$27,444(303.2)$	$27,546$
dlx2_cc_a_bug59	$6.7 \cdot 10^6$	$2.2 \cdot 10^6$	$10,954(118.4)$	$4,595$

Table 24: Experimental data for test set `sss-1.0a` based on 10 tries.

benchmark	substitutions		flips	
	mean	median	mean(time)	median
dlx2_cc_bug01	$3.1 \cdot 10^6$	$2.0 \cdot 10^6$	10,822(54.42)	7,604
dlx2_cc_bug02	$2.4 \cdot 10^6$	$1.8 \cdot 10^6$	8,874(42.09)	6,773
dlx2_cc_bug03	$8.4 \cdot 10^6$	$5.9 \cdot 10^6$	28,408(143.7)	19,310
dlx2_cc_bug04	$1.3 \cdot 10^6$	728,715	5,142(22.34)	3,336
dlx2_cc_bug05	$2.2 \cdot 10^6$	$1.4 \cdot 10^6$	8,428(38.51)	5,882
dlx2_cc_bug06	848,384	733,260	3,714(14.59)	3,279
dlx2_cc_bug07	$11.4 \cdot 10^6$	$9.4 \cdot 10^6$	38,265(195.5)	32,706
dlx2_cc_bug08	860,353	627,210	3,623(14.59)	2,654
dlx2_cc_bug09	307,692	184,464	1,825(5.234)	1,436
dlx2_cc_bug10	498,612	335,664	2,491(8.842)	2,014
dlx2_cc_bug11	$1.1 \cdot 10^6$	853,332	4,715(20.89)	3,815
dlx2_cc_bug12	$1.4 \cdot 10^6$	759,526	5,634(26.20)	3,133
dlx2_cc_bug13	654,614	417,588	3,006(12.77)	2,429
dlx2_cc_bug14	$12.3 \cdot 10^6$	$7.3 \cdot 10^6$	40,985(208.4)	23,791
dlx2_cc_bug15	532,704	344,760	2,620(9.523)	2,109
dlx2_cc_bug16	$10.8 \cdot 10^6$	$7.9 \cdot 10^6$	36,405(165.0)	26,050
dlx2_cc_bug17	$1.0 \cdot 10^6$	568,344	3,789(18.41)	2,720
dlx2_cc_bug18	176,492	65,504	1,554(3.288)	1,380
dlx2_cc_bug19	150,584	48,512	1,307(2.293)	1,014
dlx2_cc_bug20	259,101	145,248	1,600(3.878)	1,420
dlx2_cc_bug21	131,458	43,674	1,267(2.004)	1,035
dlx2_cc_bug22	162,272	69,736	1,351(2.499)	1,180
dlx2_cc_bug23	149,650	48,416	1,291(2.297)	1,088
dlx2_cc_bug24	182,966	63,672	1,381(2.801)	1,071
dlx2_cc_bug25	291,941	170,016	1,806(4.453)	1,475
dlx2_cc_bug26	748,950	505,494	3,292(11.38)	2,929
dlx2_cc_bug27	305,169	168,960	1,817(4.344)	1,456
dlx2_cc_bug28	843,040	550,400	3,494(13.60)	2,601
dlx2_cc_bug29	$40.4 \cdot 10^6$	$23.6 \cdot 10^6$	105,965(492.6)	78,565
dlx2_cc_bug30	$3.4 \cdot 10^6$	$2.5 \cdot 10^6$	12,280(51.93)	9,716
dlx2_cc_bug31	$2.1 \cdot 10^6$	$1.2 \cdot 10^6$	7,937(33.35)	5,408
dlx2_cc_bug32	$2.6 \cdot 10^6$	$1.7 \cdot 10^6$	9,401(41.15)	7,061
dlx2_cc_bug33	$18.7 \cdot 10^6$	$13.2 \cdot 10^6$	61,723(285.2)	41,341
dlx2_cc_bug34	$7.0 \cdot 10^6$	$4.5 \cdot 10^6$	24,013(106.5)	15,198
dlx2_cc_bug35	$1.5 \cdot 10^6$	$1.0 \cdot 10^6$	5,681(23.14)	4,476
dlx2_cc_bug36	$1.5 \cdot 10^6$	$1.0 \cdot 10^6$	5,760(23.56)	4,312
dlx2_cc_bug37	391,249	177,372	2,103(6.145)	1,585
dlx2_cc_bug38	$4.4 \cdot 10^6$	$2.8 \cdot 10^6$	15,247(68.94)	9,879
dlx2_cc_bug39	$1.6 \cdot 10^6$	$1.2 \cdot 10^6$	6,694(24.72)	4,856
dlx2_cc_bug40	956,764	728,080	4,104(14.98)	3,268

Table 25: Experimental data for satisfiable instances of test set `sss-1.0` based on 10 tries.

benchmark	substitutions		flips	
	mean	median	mean(time)	median
2dlx_cc_mc_ex_bp_f2_bug001	$9.0 \cdot 10^6$	$2.2 \cdot 10^6$	29,618(216.0)	9,855
2dlx_cc_mc_ex_bp_f2_bug002	$11.6 \cdot 10^6$	$8.9 \cdot 10^6$	38,106(242.9)	31,327
2dlx_cc_mc_ex_bp_f2_bug003	$9.4 \cdot 10^6$	$5.9 \cdot 10^6$	31,027(199.0)	19,629
2dlx_cc_mc_ex_bp_f2_bug004	$7.9 \cdot 10^6$	$5.0 \cdot 10^6$	26,477(170.5)	16,684
2dlx_cc_mc_ex_bp_f2_bug005	$7.2 \cdot 10^6$	$5.9 \cdot 10^6$	25,883(150.3)	21,551
2dlx_cc_mc_ex_bp_f2_bug006	$1.3 \cdot 10^6$	$1.2 \cdot 10^6$	7,139(28.28)	6,668
2dlx_cc_mc_ex_bp_f2_bug007	471,787	260,496	4,379(9.817)	3,810
2dlx_cc_mc_ex_bp_f2_bug008	$2.3 \cdot 10^6$	$2.3 \cdot 10^6$	10,283(49.54)	10,459
2dlx_cc_mc_ex_bp_f2_bug009	929,584	448,632	7,023(19.37)	4,578
2dlx_cc_mc_ex_bp_f2_bug010	257,203	253,176	4,476(5.781)	4,200
2dlx_cc_mc_ex_bp_f2_bug011	$1.5 \cdot 10^6$	429,336	7,235(32.63)	4,431
2dlx_cc_mc_ex_bp_f2_bug012	269,858	112,166	4,622(6.355)	4,260
2dlx_cc_mc_ex_bp_f2_bug013	$2.2 \cdot 10^6$	$1.7 \cdot 10^6$	8,820(46.29)	7,503
2dlx_cc_mc_ex_bp_f2_bug014	694,968	623,205	5,119(15.87)	5,036
2dlx_cc_mc_ex_bp_f2_bug015	$13.0 \cdot 10^6$	$5.4 \cdot 10^6$	32,492(378.4)	27,833
2dlx_cc_mc_ex_bp_f2_bug016	$6.6 \cdot 10^6$	$2.0 \cdot 10^6$	22,399(138.8)	10,178
2dlx_cc_mc_ex_bp_f2_bug017	$1.1 \cdot 10^6$	857,869	6,290(24.64)	5,675
2dlx_cc_mc_ex_bp_f2_bug018	$1.5 \cdot 10^6$	699,480	8,069(32.35)	5,600
2dlx_cc_mc_ex_bp_f2_bug019	$6.4 \cdot 10^6$	$6.2 \cdot 10^6$	22,396(134.0)	20,683
2dlx_cc_mc_ex_bp_f2_bug020	567,678	183,122	4,960(11.79)	3,686
2dlx_cc_mc_ex_bp_f2_bug021	$5.7 \cdot 10^6$	$2.0 \cdot 10^6$	20,982(120.1)	11,121
2dlx_cc_mc_ex_bp_f2_bug022	$5.2 \cdot 10^6$	$3.7 \cdot 10^6$	21,032(108.2)	17,357
2dlx_cc_mc_ex_bp_f2_bug023	$33.7 \cdot 10^6$	$21.3 \cdot 10^6$	105,813(701.6)	62,735
2dlx_cc_mc_ex_bp_f2_bug024	$6.8 \cdot 10^6$	$5.8 \cdot 10^6$	24,274(142.9)	18,466
2dlx_cc_mc_ex_bp_f2_bug025	$7.8 \cdot 10^6$	$5.6 \cdot 10^6$	26,052(163.4)	19,404
2dlx_cc_mc_ex_bp_f2_bug026	$9.5 \cdot 10^6$	$3.9 \cdot 10^6$	30,613(198.8)	15,644
2dlx_cc_mc_ex_bp_f2_bug027	$6.1 \cdot 10^6$	$3.2 \cdot 10^6$	21,763(126.9)	12,372
2dlx_cc_mc_ex_bp_f2_bug028	526,598	292,104	5,745(14.70)	5,505
2dlx_cc_mc_ex_bp_f2_bug029	$42.4 \cdot 10^6$	$35.6 \cdot 10^6$	126,262(884.0)	107,462
2dlx_cc_mc_ex_bp_f2_bug030	$44.6 \cdot 10^6$	$25.0 \cdot 10^6$	136,510(931.1)	79,269
2dlx_cc_mc_ex_bp_f2_bug031	$8.5 \cdot 10^6$	$7.2 \cdot 10^6$	28,515(178.7)	26,410
2dlx_cc_mc_ex_bp_f2_bug032	$24.6 \cdot 10^6$	$15.4 \cdot 10^6$	75,111(515.1)	44,734
2dlx_cc_mc_ex_bp_f2_bug033	$43.5 \cdot 10^6$	$9.2 \cdot 10^6$	130,348(908.4)	31,653
2dlx_cc_mc_ex_bp_f2_bug034	$74.2 \cdot 10^6$	$36.6 \cdot 10^6$	222,219(1,547)	111,331
2dlx_cc_mc_ex_bp_f2_bug035	$60.9 \cdot 10^6$	$41.5 \cdot 10^6$	177,153(1,271)	122,146
2dlx_cc_mc_ex_bp_f2_bug036	$132.6 \cdot 10^6$	$37.3 \cdot 10^6$	408,227(2,764)	116,553
2dlx_cc_mc_ex_bp_f2_bug037	$9.1 \cdot 10^6$	$7.1 \cdot 10^6$	30,539(190.7)	26,634
2dlx_cc_mc_ex_bp_f2_bug038	$15.9 \cdot 10^6$	$9.5 \cdot 10^6$	50,481(332.8)	37,471
2dlx_cc_mc_ex_bp_f2_bug039	$54.2 \cdot 10^6$	$23.9 \cdot 10^6$	164,756(1,130)	72,974
2dlx_cc_mc_ex_bp_f2_bug040	$71.9 \cdot 10^6$	$74.7 \cdot 10^6$	220,732(1,499)	221,820
2dlx_cc_mc_ex_bp_f2_bug041	$13.7 \cdot 10^6$	$10.7 \cdot 10^6$	45,218(285.5)	34,839
2dlx_cc_mc_ex_bp_f2_bug042	$8.2 \cdot 10^6$	$2.6 \cdot 10^6$	28,414(172.6)	11,290
2dlx_cc_mc_ex_bp_f2_bug043	$36.6 \cdot 10^6$	$31.9 \cdot 10^6$	113,633(763.6)	96,088
2dlx_cc_mc_ex_bp_f2_bug044	$37.0 \cdot 10^6$	$35.8 \cdot 10^6$	112,398(772.7)	105,035
2dlx_cc_mc_ex_bp_f2_bug045	$11.7 \cdot 10^6$	$7.7 \cdot 10^6$	39,594(244.0)	29,536
2dlx_cc_mc_ex_bp_f2_bug046	$16.9 \cdot 10^6$	$15.3 \cdot 10^6$	54,722(354.5)	47,072
2dlx_cc_mc_ex_bp_f2_bug047	$6.2 \cdot 10^6$	$3.0 \cdot 10^6$	15,623(182.0)	12,536
2dlx_cc_mc_ex_bp_f2_bug048	$1,9 \cdot 10^6$	589,745	8,664(51.51)	7,164
2dlx_cc_mc_ex_bp_f2_bug049	$1.2 \cdot 10^6$	520,240	7,202(13.57)	3,698
2dlx_cc_mc_ex_bp_f2_bug050	$6.4 \cdot 10^6$	637,692	22,168(133.4)	6,055

Table 26: Experimental data for test set `sss-sat-1.0` based on 10 tries (part 1).

benchmark	substitutions		flips	
	mean	median	mean(time)	median
2dlx_cc_mc_ex_bp_f2_bug051	$1.1 \cdot 10^6$	$1.2 \cdot 10^6$	5,948(16.55)	6,684
2dlx_cc_mc_ex_bp_f2_bug052	$3.8 \cdot 10^6$	$2.1 \cdot 10^6$	15,717(86.91)	7,659
2dlx_cc_mc_ex_bp_f2_bug053	$79.2 \cdot 10^6$	$61.1 \cdot 10^6$	236,486(1,651)	170,469
2dlx_cc_mc_ex_bp_f2_bug054	652,222	352,665	4,773(11.50)	3,315
2dlx_cc_mc_ex_bp_f2_bug055	885,447	176,225	4,546(13.11)	2,907
2dlx_cc_mc_ex_bp_f2_bug056	$197.1 \cdot 10^6$	$179.0 \cdot 10^6$	579,572(4,111)	510,344
2dlx_cc_mc_ex_bp_f2_bug057	$53.3 \cdot 10^6$	$39.1 \cdot 10^6$	158,815(1,110)	126,677
2dlx_cc_mc_ex_bp_f2_bug058	$8.3 \cdot 10^6$	$3.9 \cdot 10^6$	27,050(173.2)	12,758
2dlx_cc_mc_ex_bp_f2_bug059	$9.7 \cdot 10^6$	$3.3 \cdot 10^6$	30,792(200.7)	14,994
2dlx_cc_mc_ex_bp_f2_bug060	$2.4 \cdot 10^6$	$1.3 \cdot 10^6$	11,526(39.59)	6,826
2dlx_cc_mc_ex_bp_f2_bug061	$3.4 \cdot 10^6$	$2.5 \cdot 10^6$	14,115(64.07)	11,915
2dlx_cc_mc_ex_bp_f2_bug062	$170.6 \cdot 10^6$	$137.9 \cdot 10^6$	502,696(3,563)	407,847
2dlx_cc_mc_ex_bp_f2_bug063	$73.4 \cdot 10^6$	$48.8 \cdot 10^6$	221,083(1,526)	156,752
2dlx_cc_mc_ex_bp_f2_bug064	$21.1 \cdot 10^6$	$6.8 \cdot 10^6$	66,269(442.6)	27,501
2dlx_cc_mc_ex_bp_f2_bug065	$20.3 \cdot 10^6$	$14.1 \cdot 10^6$	62,769(424.3)	48,419
2dlx_cc_mc_ex_bp_f2_bug066	$8.6 \cdot 10^6$	$2.6 \cdot 10^6$	30,247(194.3)	12,912
2dlx_cc_mc_ex_bp_f2_bug067	$203.4 \cdot 10^6$	$164.5 \cdot 10^6$	606,728(4,268)	508,419
2dlx_cc_mc_ex_bp_f2_bug068	$54.7 \cdot 10^6$	$41.5 \cdot 10^6$	165,404(1,144)	121,492
2dlx_cc_mc_ex_bp_f2_bug069	$129.9 \cdot 10^6$	$47.9 \cdot 10^6$	398,825(2,722)	152,788
2dlx_cc_mc_ex_bp_f2_bug070	$30.0 \cdot 10^6$	$24.1 \cdot 10^6$	89,546(629.9)	61,245
2dlx_cc_mc_ex_bp_f2_bug071	$30.9 \cdot 10^6$	$14.4 \cdot 10^6$	93,248(647.5)	46,444
2dlx_cc_mc_ex_bp_f2_bug072	$47.5 \cdot 10^6$	$23.2 \cdot 10^6$	141,931(999.5)	67,962
2dlx_cc_mc_ex_bp_f2_bug073	$36.5 \cdot 10^6$	$14.0 \cdot 10^6$	108,912(769.1)	48,748
2dlx_cc_mc_ex_bp_f2_bug074	$17.6 \cdot 10^6$	$9.5 \cdot 10^6$	60,301(367.0)	30,279
2dlx_cc_mc_ex_bp_f2_bug075	$13.3 \cdot 10^6$	$8.0 \cdot 10^6$	44,023(278.0)	26,859
2dlx_cc_mc_ex_bp_f2_bug076	$44.0 \cdot 10^6$	$31.2 \cdot 10^6$	132,022(1,025)	93,041
2dlx_cc_mc_ex_bp_f2_bug077	$170.8 \cdot 10^6$	$106.6 \cdot 10^6$	506,083(3,992)	312,401
2dlx_cc_mc_ex_bp_f2_bug078	$47.6 \cdot 10^6$	$39.0 \cdot 10^6$	144,787(1,120)	118,545
2dlx_cc_mc_ex_bp_f2_bug079	$8.5 \cdot 10^6$	$4.7 \cdot 10^6$	28,073(204.9)	16,803
2dlx_cc_mc_ex_bp_f2_bug080	$5.1 \cdot 10^6$	$5.1 \cdot 10^6$	18,384(122.6)	18,274
2dlx_cc_mc_ex_bp_f2_bug081	$3,5 \cdot 10^6$	953,400	14,336(84.04)	6,219
2dlx_cc_mc_ex_bp_f2_bug082	$1.4 \cdot 10^6$	484,704	7,967(35.55)	5,050
2dlx_cc_mc_ex_bp_f2_bug083	$5.5 \cdot 10^6$	$2.5 \cdot 10^6$	18,891(132.6)	10,459
2dlx_cc_mc_ex_bp_f2_bug084	$4.0 \cdot 10^6$	$1.5 \cdot 10^6$	15,012(96.95)	8,470
2dlx_cc_mc_ex_bp_f2_bug085	$2.9 \cdot 10^6$	$2.0 \cdot 10^6$	11,089(74.21)	8,769
2dlx_cc_mc_ex_bp_f2_bug086	$1.4 \cdot 10^6$	$1.0 \cdot 10^6$	7,555(33.77)	6,755
2dlx_cc_mc_ex_bp_f2_bug087	729,151	141,240	5,505(19.04)	4,071
2dlx_cc_mc_ex_bp_f2_bug088	$2.6 \cdot 10^6$	$1.5 \cdot 10^6$	11,238(64.51)	7,904
2dlx_cc_mc_ex_bp_f2_bug089	$2.0 \cdot 10^6$	$1.1 \cdot 10^6$	8,856(49.20)	6,307
2dlx_cc_mc_ex_bp_f2_bug090	$2.2 \cdot 10^6$	$1.0 \cdot 10^6$	9,589(53.98)	7,062
2dlx_cc_mc_ex_bp_f2_bug091	$24.9 \cdot 10^6$	$15.2 \cdot 10^6$	73,430(650.0)	50,034
2dlx_cc_mc_ex_bp_f2_bug092	$13.1 \cdot 10^6$	$9.4 \cdot 10^6$	39,524(340.9)	30,876
2dlx_cc_mc_ex_bp_f2_bug093	$2.7 \cdot 10^6$	$1.7 \cdot 10^6$	13,410(60.03)	9,079
2dlx_cc_mc_ex_bp_f2_bug094	$1.4 \cdot 10^6$	443,880	7,132(39.41)	4,738
2dlx_cc_mc_ex_bp_f2_bug095	$201.5 \cdot 10^6$	$80.7 \cdot 10^6$	680,399(5,367)	249,237
2dlx_cc_mc_ex_bp_f2_bug096	$53.8 \cdot 10^6$	$33.8 \cdot 10^6$	161,637(1,257)	99,578
2dlx_cc_mc_ex_bp_f2_bug097	$82.2 \cdot 10^6$	$77.2 \cdot 10^6$	239,358(2,029)	227,207
2dlx_cc_mc_ex_bp_f2_bug098	513,003	372,771	6,039(13.09)	6,038
2dlx_cc_mc_ex_bp_f2_bug099	56,487	18,340	859(0.387)	579
2dlx_cc_mc_ex_bp_f2_bug100	$206.3 \cdot 10^6$	$161.3 \cdot 10^6$	655,423(5,453)	479,931

Table 27: Experimental data for test set `sss-sat-1.0` based on 10 tries (part 2).

benchmark	substitutions		flips	
	mean	median	mean(time)	median
9vliw_bp_mc_bug001	$64.0 \cdot 10^6$	$41.6 \cdot 10^6$	$55,543(1,922)$	36,898
9vliw_bp_mc_bug002	$15.3 \cdot 10^6$	$12.6 \cdot 10^6$	$20,325(538.3)$	18,898
9vliw_bp_mc_bug003	$35.1 \cdot 10^6$	$38.7 \cdot 10^6$	$38,868(1,116)$	36,212
9vliw_bp_mc_bug004	$11.1 \cdot 10^6$	$13.1 \cdot 10^6$	$19,024(372.1)$	21,089
9vliw_bp_mc_bug005	$292 \cdot 10^6$	$292 \cdot 10^6$	$177,207(9,090)$	178,219
9vliw_bp_mc_bug006	$43.5 \cdot 10^6$	$43.0 \cdot 10^6$	$39,919(1,325)$	42,722
9vliw_bp_mc_bug007	$149 \cdot 10^6$	$171 \cdot 10^6$	$94,384(4,512)$	105,425
9vliw_bp_mc_bug008	$163 \cdot 10^6$	$11.4 \cdot 10^6$	$107,012(4,786)$	19,098
19vliw_bp_mc_bug009	$100 \cdot 10^6$	$43.8 \cdot 10^6$	$67,044(3,009)$	33,915
9vliw_bp_mc_bug010	$89.6 \cdot 10^6$	$59.1 \cdot 10^6$	$65,626(2,636)$	52,687
9vliw_bp_mc_bug011	$23.1 \cdot 10^6$	$17.1 \cdot 10^6$	$26,193(694.0)$	18,696
9vliw_bp_mc_bug012	$80.9 \cdot 10^6$	$2.94 \cdot 10^6$	$61,060(2,482)$	12,637
9vliw_bp_mc_bug013	$25.4 \cdot 10^6$	$11.1 \cdot 10^6$	$28,143(773.6)$	18,238
9vliw_bp_mc_bug014	$45.0 \cdot 10^6$	$57.7 \cdot 10^6$	$29,994(1,430)$	35,307
9vliw_bp_mc_bug015	$60.8 \cdot 10^6$	$56.4 \cdot 10^6$	$42,239(1,819)$	38,079
9vliw_bp_mc_bug016	$56.1 \cdot 10^6$	$33.7 \cdot 10^6$	$51,539(1,700)$	37,813
9vliw_bp_mc_bug017	$27.3 \cdot 10^6$	$26.5 \cdot 10^6$	$28,025(806.1)$	28,566
9vliw_bp_mc_bug018	$31.3 \cdot 10^6$	$5.7 \cdot 10^6$	$22,216(1,010)$	13,956
9vliw_bp_mc_bug019	$75.1 \cdot 10^6$	$74.9 \cdot 10^6$	$57,502(2,287)$	57,634
9vliw_bp_mc_bug020	$93.7 \cdot 10^6$	$80.7 \cdot 10^6$	$69,159(2,807)$	64,995
9vliw_bp_mc_bug021	$69.4 \cdot 10^6$	$93.0 \cdot 10^6$	$53,633(2,089)$	66,933
9vliw_bp_mc_bug022	$66.9 \cdot 10^6$	$58.6 \cdot 10^6$	$56,341(2,039)$	53,882
9vliw_bp_mc_bug023	$5.6 \cdot 10^6$	$4.3 \cdot 10^6$	$14,565(167.2)$	14,260
9vliw_bp_mc_bug024	$63.7 \cdot 10^6$	$53.8 \cdot 10^6$	$51,511(1,930)$	42,586
9vliw_bp_mc_bug025	$49.9 \cdot 10^6$	$16.4 \cdot 10^6$	$37,558(1,601)$	20,139
9vliw_bp_mc_bug026	$1.8 \cdot 10^6$	$2.3 \cdot 10^6$	$14,678(61.22)$	14,696
9vliw_bp_mc_bug027	$85.6 \cdot 10^6$	$59.5 \cdot 10^6$	$65,492(2,834)$	49,575
9vliw_bp_mc_bug028	$35.8 \cdot 10^6$	$36.8 \cdot 10^6$	$41,209(1,162)$	43,786
9vliw_bp_mc_bug029	$3.6 \cdot 10^6$	$4.2 \cdot 10^6$	$12,898(142.2)$	13,506
9vliw_bp_mc_bug030	$49.9 \cdot 10^6$	$5.2 \cdot 10^6$	$44,076(1,658)$	45,439
9vliw_bp_mc_bug031	$27.3 \cdot 10^6$	$22.0 \cdot 10^6$	$30,719(950.8)$	25,801
9vliw_bp_mc_bug032	$3.54 \cdot 10^6$	$2.2 \cdot 10^6$	$15,176(125.4)$	14,132
9vliw_bp_mc_bug033	$218 \cdot 10^6$	$126 \cdot 10^6$	$137,708(7,548)$	91,544
9vliw_bp_mc_bug034	$30.5 \cdot 10^6$	$10.2 \cdot 10^6$	$32,257(972.7)$	25,694
9vliw_bp_mc_bug035	$74.0 \cdot 10^6$	$57.8 \cdot 10^6$	$50,840(2,426)$	38,555
9vliw_bp_mc_bug036	$32.1 \cdot 10^6$	$28.6 \cdot 10^6$	$33,519(990.7)$	28,296
9vliw_bp_mc_bug037	$45.6 \cdot 10^6$	$36.0 \cdot 10^6$	$39,878(1,464)$	14,442
9vliw_bp_mc_bug040	$5.1 \cdot 10^6$	$3.5 \cdot 10^6$	$14,646(166.6)$	15,376
9vliw_bp_mc_bug041	$27.1 \cdot 10^6$	$26.5 \cdot 10^6$	$33,250(872.4)$	33,248
9vliw_bp_mc_bug042	$57.6 \cdot 10^6$	$52.6 \cdot 10^6$	$45,342(1,823)$	44,563
9vliw_bp_mc_bug043	$44.2 \cdot 10^6$	$23.4 \cdot 10^6$	$39,017(1,406)$	31,087
9vliw_bp_mc_bug044	$25.6 \cdot 10^6$	$31.8 \cdot 10^6$	$34,037(823.4)$	38,659
9vliw_bp_mc_bug045	$35.5 \cdot 10^6$	$42.4 \cdot 10^6$	$32,746(1,185)$	37,431
9vliw_bp_mc_bug046	$81.7 \cdot 10^6$	$76.1 \cdot 10^6$	$63,964(2,740)$	67,696
9vliw_bp_mc_bug047	$67.9 \cdot 10^6$	$45.5 \cdot 10^6$	$57,165(2,158)$	41,531

Table 28: Experimental data for some benchmarks from `vliw-sat-1.1` family based on 3 tries (part 1).

benchmark	substitutions		flips	
	mean	median	mean(time)	median
9vliw_bp_mc_bug048	$23.6{\cdot}10^6$	$5.9{\cdot}10^6$	$31,994(748.5)$	18,182
9vliw_bp_mc_bug050	$42.4{\cdot}10^6$	$30.5{\cdot}10^6$	$37,944(1,420)$	35,061
9vliw_bp_mc_bug051	$13.2{\cdot}10^6$	$6.4{\cdot}10^6$	$22,949(501.8)$	19,305
9vliw_bp_mc_bug052	$38.5{\cdot}10^6$	$7.4{\cdot}10^6$	$36,693(1,159)$	19,690
9vliw_bp_mc_bug054	$60.4{\cdot}10^6$	$57.3{\cdot}10^6$	$47,508(1,937)$	43,133
9vliw_bp_mc_bug055	$37.5{\cdot}10^6$	$20.4{\cdot}10^6$	$35,498(1,159)$	24,669
9vliw_bp_mc_bug056	$25.6{\cdot}10^6$	$7.6{\cdot}10^6$	$24,117(806.0)$	16,805
9vliw_bp_mc_bug057	$17.9{\cdot}10^6$	$19.0{\cdot}10^6$	$24,288(575.3)$	26,985
9vliw_bp_mc_bug058	$16.6{\cdot}10^6$	$6.3{\cdot}10^6$	$23,928(556.5)$	19,731
9vliw_bp_mc_bug059	$66.0{\cdot}10^6$	$73.4{\cdot}10^6$	$52,341(2,157)$	63,632
9vliw_bp_mc_bug060	$44.0{\cdot}10^6$	$20.7{\cdot}10^6$	$36,500(1,503)$	26,147
9vliw_bp_mc_bug061	$3.1{\cdot}10^6$	$3.4{\cdot}10^6$	$13,885(95.68)$	13,564
9vliw_bp_mc_bug062	$41.8{\cdot}10^6$	$26.0{\cdot}10^6$	$39,350(1,222)$	31,012
9vliw_bp_mc_bug063	$23.1{\cdot}10^6$	$24.0{\cdot}10^6$	$26,552(708.9)$	28,386
9vliw_bp_mc_bug065	$276{\cdot}10^6$	$269{\cdot}10^6$	$172,023(9,974)$	164,647
9vliw_bp_mc_bug066	$224{\cdot}10^6$	$160{\cdot}10^6$	$136,735(6,983)$	104,797
9vliw_bp_mc_bug067	$80.6{\cdot}10^6$	$64.9{\cdot}10^6$	$50,155(2,729)$	51,648
9vliw_bp_mc_bug068	$57.9{\cdot}10^6$	$61.9{\cdot}10^6$	$48,164(1,810)$	45,295
9vliw_bp_mc_bug069	$3.8{\cdot}10^6$	$4.0{\cdot}10^6$	$15,346(119.6)$	15,719
9vliw_bp_mc_bug070	$303{\cdot}10^6$	$267{\cdot}10^6$	$185,328(9,485)$	163,226
9vliw_bp_mc_bug071	$9.0{\cdot}10^6$	$7.6{\cdot}10^6$	$20,601(282.8)$	21,389
9vliw_bp_mc_bug072	$81.7{\cdot}10^6$	$47.4{\cdot}10^6$	$65,134(2,457)$	42,545
9vliw_bp_mc_bug073	$8.9{\cdot}10^6$	$11.4{\cdot}10^6$	$20,352(307.5)$	20,246
9vliw_bp_mc_bug074	$75.0{\cdot}10^6$	$73{,}3{\cdot}10^6$	$65,105(2,136)$	63,242
9vliw_bp_mc_bug075	$67.5{\cdot}10^6$	$92.1{\cdot}10^6$	$55,090(2,261)$	66,872
9vliw_bp_mc_bug076	$187.0{\cdot}10^6$	$96.1{\cdot}10^6$	$122,165(5,699)$	8,2541
9vliw_bp_mc_bug077	$6.7{\cdot}10^6$	$4.8{\cdot}10^6$	$17,037(201.7)$	15,797
9vliw_bp_mc_bug078	$34.7{\cdot}10^6$	$34.4{\cdot}10^6$	$38,382(1,101)$	44,512
9vliw_bp_mc_bug079	$147.{\cdot}10^6$	$82.6{\cdot}10^6$	$99,402(4,755)$	67,430
9vliw_bp_mc_bug080	$69.0{\cdot}10^6$	$59.4{\cdot}10^6$	$49,564(2,252)$	42,413
9vliw_bp_mc_bug081	$10.3{\cdot}10^6$	$7.1{\cdot}10^6$	$20,479(333.6)$	17,708
9vliw_bp_mc_bug082	$46.0{\cdot}10^6$	$51.2{\cdot}10^6$	$33,489(1,471)$	39,283
9vliw_bp_mc_bug083	$16.7{\cdot}10^6$	$7.2{\cdot}10^6$	$22,067(513.2)$	14,577
9vliw_bp_mc_bug084	$15.4{\cdot}10^6$	$16.0{\cdot}10^6$	$20,926(468.6)$	18,587
9vliw_bp_mc_bug085	$76.0{\cdot}10^6$	$64.9{\cdot}10^6$	$56,172(2,336)$	52,330
9vliw_bp_mc_bug086	$121{\cdot}10^6$	$119{\cdot}10^6$	$79,492(3,881)$	75,832
9vliw_bp_mc_bug087	$49.4{\cdot}10^6$	$41.6{\cdot}10^6$	$43,002(1,500)$	42,318
9vliw_bp_mc_bug088	$96.4{\cdot}10^6$	$52.5{\cdot}10^6$	$64,894(2,977)$	36,295
9vliw_bp_mc_bug091	$69.5{\cdot}10^6$	$2.0{\cdot}10^6$	$54,202(2,050)$	11,135

Table 29: Experimental data for some benchmarks from `vliw-sat-1.1` family based on 3 tries (part 2).