

QingTing: A Local Search SAT Solver Using an Effective Switching Strategy and an Efficient Unit Propagation

Xiao Yu Li, Matthias F. Stallmann, and Franc Brglez

Dept. of Computer Science, NC State Univ., Raleigh, NC 27695, USA
{xyli,mfms,brglez}@unity.ncsu.edu

Abstract. Advances in local-search SAT solvers have traditionally been presented in the context of local search solvers only. The most recent and rather comprehensive comparisons between UnitWalk and several versions of WalkSAT demonstrate that neither solver dominates on all benchmarks. QingTing2 (a ‘dragonfly’ in Mandarin) is a SAT solver script that relies on a novel switching strategy to invoke one of the two local search solvers: WalkSAT or QingTing1. The local search solver QingTing1 implements the UnitWalk algorithm with a new unit-propagation technique. The experimental methodology we use not only demonstrates the effectiveness of the switching strategy and the efficiency of the new unit-propagation implementation – it also supports, on the very same instances, statistically significant performance evaluation between local search and other state-of-the-art DPLL-based SAT solvers. The resulting comparisons show a surprising pattern of solver dominance, completely unanticipated when we began this work.

1 Introduction

SAT problems of increasing complexity arise in many domains. The performance concerns about SAT solvers continue to stimulate the development of new SAT algorithms and their implementations. The number of SAT-solver entries entered into the 2003 SAT-solver competition [1] exceeds 30. The Web has become the universal resource to access large and diverse directories of SAT benchmarks[2], SAT discussion forums [3], and SAT experiments [4], each with links to SAT-solvers that can be readily down-loaded and installed.

For the most part, this paper addresses the algorithms and performance issues related to the (stochastic) local search SAT solvers. To date, relative improvements of such solvers have been presented in the context of local search solvers only. Up to seven local search SAT algorithms have been evaluated (under benchmarks-specific ‘noise levels’) in [5], including four different versions of WalkSAT [6–8]. The most recent comparisons between several versions of WalkSAT and UnitWalk demonstrate that neither solver dominates on all problem instances [9].

The performance analysis of local search SAT solvers as reported in [5] and [9] is based on statistical analysis of multiple runs with the same inputs, using

different seeds. On the other hand, DPLL-based (deterministic) solvers such as *chaff* [10] and *sato* [11], are compared either on basis of a single run or multiple runs with incomparable inputs. This explains the lack of statistically significant performance comparisons between the two types of SAT solvers – until now. By introducing syntactical transformations of a problem instance we can now generate, for each reference cnf formula, an *equivalence class* of as many instances as we find necessary for an experiment of statistical significance involving both deterministic and stochastic solvers [12, 13]. For the stochastic local search solvers, results are (statistically) the same whether we do multiple runs with different seeds on identical inputs or with the same seed on a class of inputs that differ only via syntactical transformations.

Our research, as reported in this paper, proceeded by asking and pursuing answers to the three questions in the following order:

1. Can we accelerate UnitWalk by improving the implementation of its unit-propagation?
2. Can we analyze each problem instance to devise a switching strategy to correctly select the dominating local search algorithm?
3. How much would state-of-the-art DPLL-based solvers change the pattern of solver dominance on the very same problem instances?

Answers to these questions are supported by a large number of experiments, making extensive use of the SATbed features as reported in [12]. Raw data from these experiments as well as formatted tables and statistics are openly accessible on the Web, along with instance classes, each of minimum size of 32. All experiments can be readily replicated, verified, and extended by other researchers.

Following a suggestion of Johnson [14], we treat each algorithm as a fixed entity with no “hand-tuned” parameters. For two of the algorithms, UnitWalk2 and QingTing2, some tuning takes place during execution and is charged as part of the execution time. Extensive studies [5] have been done on how to tune WalkSAT for some of the benchmarks discussed in this paper. Given that the tuning is not part of the execution, we had to make a choice to be fair to all SAT solvers reported in this paper.^a We used the default heuristics (“best”) and noise level (0.5) for WalkSAT. We ran all our experiments on a P-II@266 MHz Linux machine with 196 MBytes of physical memory and set the timeout value for each experiment to be 1800 seconds.

This paper is organized as follows. In Section 2, we provide the motivation and background information. We describe how unit propagation is implemented in QingTing1 and its effect on reducing clause and literal visits in Section 3. We then

^a See the extended discussion about the “PET PIEVE 10. *Hand-tuned algorithm parameters*” that concludes with a recommended rule as follows [14]:

If different parameter settings are to be used for different instances, the adjustment process must be well-defined and algorithmic, the adjustment must be described in the paper, and the time for the adjustment must be included in all reported running times.

present the switching strategy and QingTing2 in Section 4. Comparisons between deterministic and stochastic solvers are presented in Section 5. We conclude the paper and describe some future research directions in Section 6.

2 Motivation and Background

In this section, we first give the motivation of our work. We then introduce some basic concepts and briefly discuss the key differences between UnitWalk and WalkSAT.

2.1 Motivation

Recently, Hirsch and Kojevnikov [9] proposed and implemented a new local search algorithm called UnitWalk. Despite its simple implementation, the UnitWalk solver outperforms WalkSAT on structured benchmarks such as planning and parity learning. Unlike GSAT and WalkSAT, UnitWalk relies heavily on unit propagation and any speedup gained on this operation will directly boost the solver’s performance. We demonstrate that such speedup is in fact possible using Zhang’s unit propagation algorithm [15] with chaff’s lazy data structure [10].

However, UnitWalk is not competitive with WalkSAT on randomly generated 3-SAT benchmarks and some benchmarks from problem domains such as micro-processor verification. We show the large performance discrepancies between the two solvers in Figure 1. For a given benchmark, it would be highly desirable to have a strategy that can find the faster algorithm before solving it. Using such a strategy a solver can simply solve the benchmark with the more efficient algorithm of the two.

Below is a summary of the two versions of UnitWalk and two versions of our solver we refer to in the rest of the paper:

- UnitWalk1 refers to version 0.944 of the UnitWalk solver proposed in [9]. It implements the UnitWalk algorithm in a straightforward way.
- UnitWalk2 refers to version 0.981 of the UnitWalk solver [16]. UnitWalk2 incorporates both WalkSAT and a 2-SAT solver into its search strategy and it also uses Zhang’s unit propagation algorithm.
- QingTing1^b refers to version 1.0 of our solver. It improves upon UnitWalk1 by implementing unit propagation more efficiently.
- QingTing2 refers to version 2.0 of our solver. It is a SAT solver script that relies on a novel switching strategy to invoke one of the two local search solvers: WalkSAT or QingTing1.

2.2 Basic Concepts

We consider algorithms for SAT formulated in conjunctive normal form (CNF). A CNF formula F is the conjunction of its clauses where each clause is a disjunction of literals. A literal x in F appears in the form of either v or its complement \bar{v} ,

^b QingTing1 is available at <http://pluto.cbl.ncsu.edu/EDS/QingTing>.

The examples in the two following tables shows that UnitWalk1 (version 0.944) and WalkSAT can outperform each other significantly on different benchmarks. In the upper table, UnitWalk1 is shown to dominate WalkSAT on the blocks-world benchmarks (`bw_large_a` and `bw_large_b`) and the parity learning benchmarks (`par16-4-c` and `par16-5-c`). However, the lower table shows that WalkSAT runs as many as 62 times faster than UnitWalk1 on the random 3-SAT benchmarks (`uf250-1065-027` and `uf250-1065-087`) and microprocessor verification benchmarks (`dlx2_cc_a_bug17` and `dlx2_cc_a_bug39`). Therefore, for the eight benchmarks, UnitWalk1 and WalkSAT can complement each other’s performance, and this can be made possible if there is an efficient strategy that can choose the faster solver from the two.

Solvers	<code>bw_large_a</code>	<code>bw_large_b</code>	<code>par16-4-c</code>	<code>par16-5-c</code>
UnitWalk1	0.13/0.17	3.90/3.58	75.3/68.6	69.0/55.8
WalkSAT	0.27/0.26	17.4/17.1	timeout	timeout

Solvers	<code>uf250-1065-027</code>	<code>uf250-1065-087</code>	<code>dlx2_cc_a_bug17</code>	<code>dlx2_cc_a_bug39</code>
UnitWalk1	6.84/6.46	12.4/12.7	61.6/61.9	86.4/85.4
WalkSAT	0.11/0.09	0.15/0.15	2.44/1.82	4.18/3.09

Fig. 1. Performance comparisons for UnitWalk1 and WalkSAT on 32 PC-class instances for each of the respective benchmarks (see details in Section 5). Table entries report the mean/standard-deviation of runtime (in seconds). The timeout is set to be 1800 seconds.

for some variable v . A clause is *satisfied* if at least one literal in the clause is true. F is satisfiable if there exists an assignment for variables that satisfies each clause in F ; otherwise, F is unsatisfiable. A clause containing only one literal is called a *unit clause*.

Consider a CNF formula F and let x be a literal in the formula. Then F can be divided into three sets:

- $A = \{x \vee A_1, \dots, x \vee A_m\}$: the clauses that contain x .
- $B = \{\bar{x} \vee B_1, \dots, \bar{x} \vee B_n\}$: the clauses that contain \bar{x} .
- $R = \{R_1, \dots, R_l\}$: the clauses that contain neither x nor \bar{x} .

When x is set true, the *unit propagation* operation, $F := F[x \leftarrow true]$, will delete \bar{x} from B and remove A from F . New unit clauses may arise and unit propagation continues as long as there are unit clauses in the formula. Next, we present the UnitWalk algorithm and the WalkSAT algorithm.

2.3 The Algorithms

Both UnitWalk and WalkSAT generate initial assignments for the variables uniformly at random. These assignments are modified by complementing (flipping) the value of some variable in each step. In WalkSAT the variable to be flipped is chosen from a random unsatisfied clause. Variations of WalkSAT vary in how the variable is chosen. If a solution is not reached after a specified number of

flips, WalkSAT tries again with a new random assignment. The number of such tries is not limited in our experiments, but the algorithm is terminated after a *timeout* of 1800 seconds.

UnitWalk takes advantage of unit propagation whenever possible, delaying variable assignment until unit clauses are resolved. An iteration of the outer loop of UnitWalk is called a *period*. At the beginning of a period, a permutation of the variables and their assignments are randomly chosen. The algorithm will start doing unit propagation using the assignment for the first variable in the permutation. The unit propagation process modifies the current assignment and will continue as long as unit clauses exist. When there are no unit clauses left and some variables remain unassigned, the first unassigned variable in the permutation along with its current assignment is chosen to continue the unit propagation process. At least one variable is flipped during each period, thus ensuring progress. If at the end of a period, the formula becomes empty, then the current assignment is returned as the satisfying solution. The parameter MAX_PERIODS determines how long the program will run. In our experiments with UnitWalk1 and QingTing1, MAX_PERIODS is set to infinity (the same setting used by Hirsch and Kojevnikov [9]), but, as with WalkSAT, the program is terminated if it runs for longer than our timeout.

3 Efficient Unit Propagation Using a Lazy Data Structure

As we have seen, UnitWalk relies heavily on unit propagation. Therefore, the performance of any UnitWalk-based solver depends on how fast it can execute this operation, which, in turn, is largely determined by the underlying data structure. In this section, we briefly review counter-based adjacency lists as a data structure for unit propagation. This appears to be the structure used in UnitWalk1. We then present the data structure used in QingTing1. Last, we show empirically that QingTing1's data structure reduces the number of memory accesses in terms of clause and literal visits.

3.1 Counter-Based Adjacency Lists

Variations of *adjacency lists* have traditionally been used as the data structure for unit propagation. One example is the counter-based approach. Each variable in the formula has a list of pointers to the clauses in which it appears. Each clause has a counter that keeps track of the number of unassigned literals in the clause. When a variable is assigned *true*, the following actions take place.

1. The clauses that contain its positive occurrences are declared *satisfied*.
2. The counters for the clauses that contain its negative occurrences are decremented by 1 and all its negative occurrences are marked *assigned*.
3. If a counter becomes 1 and the clause is not yet satisfied, then the clause is a unit clause and the surviving literal is found using linear search.

The case when the variable is assigned *false* works analogously. It is important to realize that when a variable is assigned, every clause containing that variable must be visited. Moreover, the average number of literal visits to find the surviving literal in a unit clause is half of the clause length.

3.2 A Lazy Data Structure

Recently, more efficient structures (often referred to as “lazy” data structures) have been used in complete SAT solvers to facilitate unit propagation [17]. Some examples include sato’s Head/Tail Lists [18] and chaff’s Watched Literals [10]. Head/Tail Lists and Watched Literals are very similar. In our solver QingTing1, we use the Watched Literals approach. Every clause has two watched literals, each a pointer to an unassigned literal in the clause. Each variable has a list of pointers to clauses in which it is one of the watched literals. Initially, the two watched literals are the first and last literal in the clause. When assigning variables, a clause is visited *only* when one of the two watched literals becomes false. Then the next unassigned literal will be searched for, which leads to three cases:

1. If a satisfied literal is found, the clause is declared *satisfied*.
2. If a new unassigned literal is found and it is not the same as the other watched literal, it becomes a new watched literal.
3. If the only unassigned literal is the other watched literal, the clause is declared *unit*.

A unit clause is declared *conflicted* when it is unsatisfied by the current assignment of the variable in the unit clause. As new unassigned literals are discovered, the list of pointers associated with each variable is dynamically maintained. The unit propagation method we just described is based on Zhang’s algorithm [15].

3.3 Reducing Clause and Literal Visits

Using the Watched Literals data structure, we have reduced the number of clause visits during unit propagation:

- Unlike the adjacency list approach, assignments to clause literals other than the two watched literals do not cause a clause visit: a clause does not become unit as long as the two variables of the watched literals are unassigned.
- In addition, the process of declaring a clause satisfied is implicit instead of explicit. An assignment satisfying a watched literal doesn’t result in a clause visit. However, there is a trade-off. Since the clause is not explicitly declared satisfied, a new unassigned literal in the clause will still instigate a search when the other watched literal is made false. With the traditional counter-based structure, these extra searches are avoided.

We implemented the UnitWalk algorithm with each of the data structures described above and then compared their performances with the UnitWalk1 solver. In local search solvers such as UnitWalk1 and QingTing1, significant variability of performance metrics can be observed by simply repeating experiments on the same instance, randomly choosing the seed. To ensure statistical significance, we run each solver 128 times on each benchmark with different seeds. In Figure 2 we report for each solver the sample mean and standard deviation of periods, flips and runtime (the time to find the first solution). All three solvers have approximately the same number of periods and flips for the three benchmarks from

UnitWalk1 and QingTing_AL use counter-based adjacency lists; QingTing1 uses Watched Literals. We run each solver 128 times on each benchmark with different seeds and report the mean and standard deviation (in the form `mean/stdev`) of the runtime, number of periods and flips, and number of clause and literal visits. There are two exceptions: (1) the number of clause and literal visits is not reported in the program output by UnitWalk1, and (2) runtime is not reported for QingTing_AL because it is implemented inefficiently — its execution time is not comparable with the other two solvers.

We observe that, though the reported number of periods and flips is virtually the same for UnitWalk1, QingTing_AL, and QingTing1, QingTing1 has fewer clause and literal visits, especially for `queen19` and `bw_large_b`. For `uf250-1065_087`, QingTing1 has fewer clause and literal visits, but runs slower than UnitWalk1. This slow-down is the cost of dynamically maintaining the Watched-Literals data structure when an assignment does not result in a unit propagation.

128 experiments on the <code>queen19</code> benchmark				
Solver	Periods	Flips	Visits	runtime
UnitWalk1	29/34	363/228	N/A	0.27/0.30
QingTing_AL	29/30	358/207	6.30e5/6.67e5	-/-
QingTing1	22/23	316/160	8.50e4/8.61e4	0.05/0.05

128 experiments on the <code>bw_large_b</code> benchmark				
Solver	Periods	Flips	Visits	runtime
UnitWalk1	204/207	1.20e4/1.19e4	N/A	3.00/3.03
QingTing_AL	176/173	1.03e4/9.64e3	5.71e6/5.59e6	-/-
QingTing1	200/234	1.19e4/1.35e4	3.46e6/4.03e6	1.71/2.00

128 experiments on the <code>uf250-1065_087</code> benchmark				
Solver	Periods	Flips	Visits	runtime
UnitWalk1	9.59e3/8.60e3	5.64e5/4.61e5	N/A	10.9/9.77
QingTing_AL	1.29e4/1.08e4	6.89e5/5.81e5	4.42e7/3.72e7	-/-
QingTing1	1.10e4/1.15e4	6.15e5/6.41e5	3.27e7/3.40e7	15.8/16.5

Fig. 2. Performance comparisons of UnitWalk1, QingTing_AL and QingTing1 on three benchmark. Table entries report mean/stdev of each metric.

different problem domains. However, the sample mean and standard deviation of the number of clause and literal visits are consistently less with the Watched-Literals approach. The ratio of improvement varies on different benchmarks.

To further demonstrate the effectiveness of the lazy data structures, we took one of the most challenging benchmarks for UnitWalk1, `bw_large_c`, and looked at the distribution of three random variables on 32 runs each of UnitWalk1 and QingTing1 — see Figure 3. As expected, the number of periods and flips exhibits exponential distribution with roughly the same mean and standard deviation for both solvers. However, the mean and standard deviation of runtime for QingTing1 are less than half those of UnitWalk1.

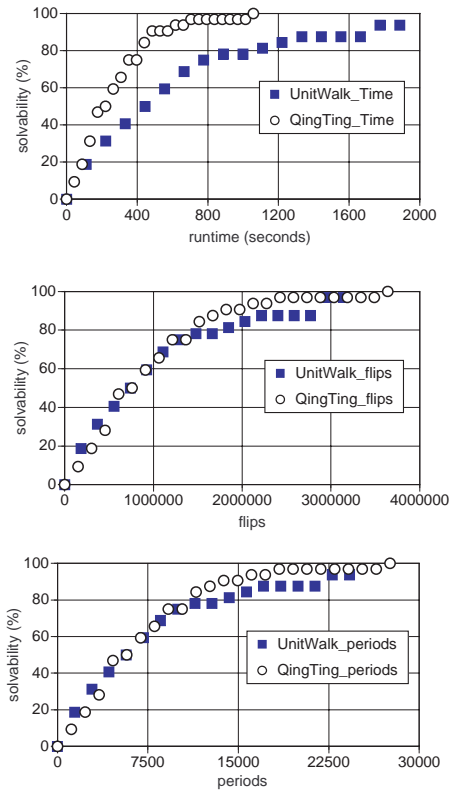


Fig. 3. Solvability functions of QingTing1 and UnitWalk1 induced by solving 32 PC-class instances of `bw_large_c`.

4 A Switching Strategy

The experimental work in [9] shows that variants of WalkSAT have better performance than UnitWalk1 on randomly generated 3-SAT benchmarks as well as some from the more realistic problem domains; however, for some highly structured benchmarks, UnitWalk1 clearly dominates. The latest release of UnitWalk has incorporated WalkSAT and it switches periodically between its internal UnitWalk and WalkSAT engines. As we will show later, UnitWalk2 is able to improve significantly upon UnitWalk1 on some benchmarks but clearly slows down the search process on many others. In this section, we propose a clearly delineated switching strategy that chooses which engine to use right at the beginning. Using this switching strategy, QingTing2 is also able to improve upon QingTing1 on many benchmarks but doesn't suffer the slowdown on the others.

Experiments with a SAT-solver A on N instances from a well-defined equivalence class of CNF formulas [19] show that parameters such as *flips* and *runtime* are random variables X with a cumulative distribution $F_X^A(x)$. For runs that *do not time-out*, we define the estimate of $F_X^A(x)$ as the *solvability function* $\mathcal{S}^A(x)$ [13]:

$$\mathcal{S}^A(x) = \frac{1}{N} (\text{num of observations} \leq x)$$

The solvability functions on the left are based on experiments with UnitWalk1 and QingTing1 on 32 PC-class instances of `bw_large_c`. We observe that for both solvers, *runtime*, *flips*, and *periods* all have *exponential distribution*, also confirmed at 5% level of significance by the χ^2 -test. The mean values of *flips* and *periods* are approximately the same while the mean values of *runtime* differ significantly for the two solvers. Sample means and standard deviations are summarized below.

	UnitWalk1 mean/stdev	QingTing1 mean/stdev
<i>runtime</i>	591/554	267/219
<i>flips</i>	986510/923651	924915/758691
<i>periods</i>	7613/7121	6988/5746

```

Algorithm QingTing2
Input: A formula  $F$  in CNF containing  $n$  variables  $v_1, \dots, v_n$ 
Output: A satisfying assignment or “No solution found”
Method:
variable_immunity = 0
random_assignments = 0
for  $i := 1$  to MAX_TRIALS do
    while  $G$  is not empty do
        assign a random value to an unassigned variable chosen at random
        random_assignments = random_assignments + 1
        do unit propagation until no unit clauses exist
    end do
end do
variable_immunity = random_assignments / n / MAX_TRIALS
if variable_immunity  $\leq 0.07$ , then return QingTing1( $F$ )
else return WalkSAT( $F$ )

```

Fig. 4. The QingTing2 algorithm.

4.1 Heuristic for the Switching Strategy

The fact that UnitWalk1 works well on highly structured benchmarks suggests that it is able to take advantage of the dependencies among the variables via unit propagation. However, when such dependencies are weak, e.g., in randomly generated 3-SAT benchmarks, WalkSAT appears to run faster. Intuitively, if we can measure such dependencies in an effective way, then we can predict which solver is faster for a given benchmark. QingTing2 shown in Figure 4, implements the switching strategy based on this idea.

Before solving a benchmark with either QingTing1 or WalkSAT, QingTing2 samples a benchmark for a fixed number of times specified by MAX_TRIALS, which we set to 128 in our experiments. During each trial, it starts by assigning a random value to an unassigned variable chosen at random. Such a step is called a random assignment. It then propagates its value through unit propagation. When the unit propagation stops, it does another random assignment. Such a process repeats and doesn’t stop until all the clauses in the formula are either conflicted or satisfied (the formula is now empty). We define *variable immunity* as the ratio between the number of random assignments in a trial and the number of variables. Intuitively, the higher the variable immunity, the less chance that variables will be assigned during unit propagation.

We considered a wide range of benchmarks and measured their variable immunities. These benchmarks include instances from the domains of blocks-world (the `bw_large` series [20]), planning (the `logistics` series [2]), scheduling (the `sched` series [12]), graph 3-coloring (the `flat` series [2]), parity learning (the `par` series [2]), randomly generated 3-SAT (the `uf250` series [2] and `hgen2` series by the `hgen2` generator [21]) and microprocessor verification (the `dlx2` series [22]). In Figure 5, the column data show the time it takes to measure the variable immunity, the sample mean, standard deviation and distribution of the variable immunity, and the QingTing1/WalkSAT runtime ratio for each benchmark

Benchmark	Time	Variable Immunity	Distribution	$\frac{\text{QingTing1}}{\text{WalkSAT}}$
sched06s_v00828	0.21	0.01/0.01	normal	$< 0.1^{(a)}$
sched07s_v01386	0.94	0.01/0.01	normal	$< 0.1^{(a)}$
bw_large_a	0.49	0.01/0.01	normal	0.48
bw_large_b	1.43	0.01/0.00	near-normal	0.17
logistics_d	4.86	0.02/0.00	normal	0.05
sched04z07s_v00655	0.57	0.04/0.01	near-normal	0.75
sched04z08s_v01094	1.14	0.04/0.01	near-normal	0.46
sched04z06s_v00354	0.26	0.06/0.02	near-normal	0.44
flat200-87	0.38	0.06/0.01	normal	3.12
flat200-33	0.38	0.07/0.00	near-normal	2.98
flat200-79	0.38	0.07/0.01	normal	1.75
par16-4-c	0.27	0.07/0.02	near-normal	0.21
par16-5-c	0.28	0.07/0.01	normal	0.22
logistics_c	1.00	0.09/0.01	normal	12.15
logistics_a	0.68	0.10/0.02	normal	83.33
logistics_b	0.67	0.11/0.02	near-normal	1.80
uf250-1065_027	0.24	0.15/0.02	normal	92.73
uf250-1065_034	0.24	0.15/0.03	normal	151.52
uf250-1065_087	0.24	0.16/0.02	normal	111.33
hgen2_v250_s26609_093	0.21	0.16/0.02	near-normal	1024.14
hgen2_v250_s53446_089	0.22	0.16/0.02	normal	17.71
dlx2_cc_a_bug39	2.59	0.18/0.05	near-normal	11.12
dlx2_cc_a_bug40	3.10	0.19/0.04	near-normal	13.67
dlx2_cc_a_bug17	7.08	0.21/0.05	near-normal	8.69

^a WalkSAT times out at 1800 seconds for the majority of instances of these classes.

Fig. 5. Time to measure variable immunity, its mean/stddev and distribution, and the runtime performance ratio of QingTing1 and WalkSAT.

(based the experiments we show in section 5). The rows are sorted based on the sample mean of variable immunity. We make the following observations.

The sample mean of variable immunity is correlated with the QingTing1/WalkSAT runtime ratio: (a) For the top section of the table where variable immunity ≤ 0.06 , the runtime for QingTing1 is a fraction of that for WalkSAT. (b) For the middle section of the table where variable immunity is between 0.06 and 0.07, either solver can dominate. (c) For the bottom section of the table where variable immunity > 0.07 , WalkSAT clearly outperforms QingTing1.

The correlation is depicted in the left scatter plot in Figure 6. It is easy to see that when variable immunity is below 0.06, the QingTing1/WalkSAT runtime ratio is less than 1, which means QingTing1 is the faster solver; and when variable immunity is above 0.07, the QingTing1/WalkSAT runtime ratio is greater than 1, which means WalkSAT is the faster solver. The same trend can be observed in the right scatter plot. Here, instead of runtime, we consider two machine independent parameters that are closely related to the solvers' runtime: the number of implications in QingTing1 and the number of flips in WalkSAT. Their ratio exhibits the same pattern as the runtime ratio.

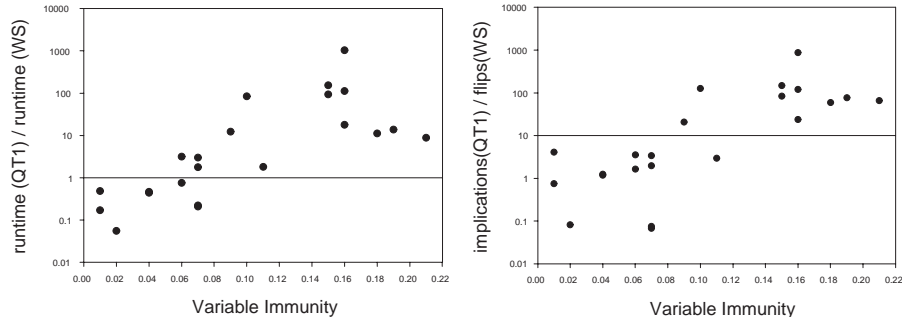


Fig. 6. The left plot shows the correlation between *variable immunity* and QingTing1(QT1)/WalkSAT(WS) runtime ratio. The right plot shows the correlation between *variable immunity* and the ratio of QingTing1(QT1) implications and WalkSAT(WS) flips.

The distributions of variable immunity are mostly normal or near-normal. Thus the mean values we report for 128 trials are relatively stable. Variable immunity is similar for benchmarks from the same problem domain (the only exception we observe is `logistics_d`). This is an indication that variable immunity has the ability to capture the intrinsic structure of the benchmarks.

In QingTing2, we choose 0.07 as the threshold for our switching strategy: if the variable immunity is less than or equal to 0.07, QingTing1 is invoked; otherwise, WalkSAT is invoked.

The overhead of the switching strategy is shown in the second column of Figure 5. Even though the overhead tends to grow as the size of the problem increases, it is still small relative to the runtime of the slower solver.

4.2 Comparisons Of Incomplete Solvers

Figure 7 shows our experiments with four local search SAT solvers on the benchmarks we considered in Figure 5. For each reference benchmark, we performed experiments on the reference as well as 32 instances from the associated PC-class (see Section 5).

First, we compare the performance of UnitWalk1 and QingTing1. QingTing1 outperforms UnitWalk1 on all benchmarks except the `uf250` and `d1x2` series. For these two series, we observe that even though we save modestly on the number of clause and literal visits, significant amount of time (around 27%) is spent on updating the dynamic data-structure in QingTing1.

For the benchmarks in the top section of Figure 7, QingTing1 is the dominating solver. For the benchmarks in the bottom section, WalkSAT is the dominating solver. QingTing2 is successful in choosing the dominating solver for all these benchmarks. For the benchmarks in the middle section, no solver clearly dominates. According to the switching strategy, QingTing2 chooses QingTing1 instead of the faster WalkSAT for the `flat` series.

Benchmark	UnitWalk1	QingTing1	UnitWalk2	WalkSAT	Dominating Solver
sched06s_v00828	0.20/0.19	0.04/0.03	0.29/0.29	1800 ^(a)	QT1
sched07s_v01386	0.40/0.42	0.10/0.09	0.67/0.70	1800 ^(a)	QT1
bw_large_a	0.13/0.17	0.13/0.12	0.22/0.22	0.27/0.26	QT1/UW1 ^(c)
bw_large_b	3.90/3.58	2.93/3.11	6.23/5.44	17.4/17.1	QT1
logistics_d	0.96/0.63	0.75/0.45	1.34/0.71	13.7/11.5	QT1/UW1 ^(c)
sched04z07s	2.58/3.23	0.73/0.98	3.36/3.76	1.59/1.53	QT1
sched04z08s	6.47/8.12	1.79/1.98	10.2/10.8	4.08/4.06	QT1
sched04z06s	0.07/0.11	0.03/0.03	0.14/0.16	0.04/0.04	QT1
flat200-87	15.9/13.9	14.9/12.1	23.9/21.2	4.77/5.08	WS ^(b)
flat200-33	3.43/3.78	4.26/3.79	5.18/5.66	1.43/1.27	WS ^(b)
flat200-79	11.1/10.1	8.80/10.9	16.9/15.7	5.04/4.88	WS ^(b)
par16-4-c	75.3/68.6	104/81.0	137/116	1800 ^(a)	UW1/QT1 ^(c)
par16-5-c	69.0/55.8	110/144	114/97.1	1800 ^(a)	UW1/QT1 ^(c)
logistics_c	281/230	107/105	307/326	8.81/8.00	WS
logistics_a	375/327	130/177	576/531	1.56/1.32	WS
logistics_b	15.4/15.6	5.25/4.41	20.0/18.9	2.91/2.47	WS
uf250-1065_027	6.84/6.46	10.2/10.1	0.48/0.44	0.11/0.09	WS
uf250-1065_034	126/118	150/112	3.80/3.78	0.99/0.95	WS
uf250-1065_087	12.4/12.7	16.7/17.2	0.45/0.36	0.15/0.15	WS
hgen2_v250...093	448/375	594/458	2.01/2.28	0.58/0.59	WS
hgen2_v250...089	835/506	487/765	59.3/61.5	27.5/24.9	WS
dlx2_cc_a_bug39	86.4/85.4	46.5/41.9	129/133	4.18/3.09	WS
dlx2_cc_a_bug40	42.2/41.7	28.7/25.4	65.9/65.3	2.10/2.15	WS
dlx2_cc_a_bug17	61.6/61.9	21.2/23.1	94.2/102	2.44/1.82	WS

^a WalkSAT times out at 1800 seconds for the majority of instances of these classes.

^b QingTing2 chooses the slower QingTing1 solver according to the switching strategy.

^c A *t*-test (at 5% level of significance) shows that QingTing1 and UnitWalk1 have the same performance on these instances.

Fig. 7. Performance comparisons of QingTing1, UnitWalk1, UnitWalk2 and WalkSAT. Numerical table entries represent mean/stdev of runtime (in seconds).

UnitWalk2 improves significantly upon UnitWalk1 on the `uf250` and `hgen2` series by utilizing its WalkSAT component. However, UnitWalk2 suffers a 10% (`logistics_c`) to 100% (`sched04z06s`) slowdown on all other benchmarks (for most of them, the slowdown is about 50%). This shows that the switching strategy implemented in UnitWalk2 is not as effective as the one introduced in this paper. Notably, QingTing2 improves upon QingTing1 for all the benchmarks in the bottom section of Figure 7 by using WalkSAT. The only slowdown it suffers is from the overhead of the switching strategy. As shown in Figure 5, this extra cost becomes a very small fraction of the runtime for harder benchmarks.

5 Comparisons with DPLL-based Solvers

As noted earlier and in the companion paper [12], our use of syntactic equivalence classes gives us an unprecedented opportunity to compare solvers across

categories. The key question we are able to ask at this point is how the local-search solvers we have evaluated compare with DPLL-based SAT solvers, the former being stochastic and the latter regarded as deterministic.

We consider three DPLL-based SAT solvers that include *chaff* [10] and two versions of the *sato* [11] solver, where *sato* uses the trie data structure and *satoL* uses the linked-list data structure. In Figure 8, we summarize the runtime data for the best solvers in both categories. It is clear from the table that

1. WalkSAT is the best solver for the randomly generated benchmarks: the *uf250* series and the *hgen2* series.
2. For the scheduling benchmarks [12], QingTing1 outperforms *chaff*.
3. For all other benchmarks, *chaff* dominates. Moreover, since *chaff* dominates for some benchmarks in each of the three benchmark categories (originally introduced in Figure 5), our switching criterion is not an effective decider when *chaff* is included.

Thus the UnitWalk-based strategy excels only for the scheduling instances, which are also shown to exhibit atypical behavior with other solvers [12]. While we have not included enough different DPLL-based solvers in this study, it appears that in almost all circumstances where a local-search solver is competitive, WalkSAT is the solver of choice. It remains to be seen whether this is an inherent limitation of doing unit propagation in the context of local search (all of the DPLL-based solvers do unit propagation as an integral part of their algorithms) or whether the current selection of benchmarks is not sufficiently rich. In other words, are the scheduling instances (unintentionally) contrived to be unlike those encountered in any other application, or are they representative of a large class of instances that naturally arise in applications heretofore not considered?

6 Conclusions and Future Research

We have introduced a new local-search-based SAT solver QingTing. With an efficient implementation using the Watched Literals data structure, QingTing1 is able to improve upon UnitWalk1. We have also experimentally measured the effect data structures have on the number of clause and literal visits required for unit propagation. Combining WalkSAT with QingTing1 using a lower-overhead switching strategy, QingTing2 has a better overall performance than either solver used exclusively.

More important, however, are the lessons learned in this process. We set out to improve UnitWalk without first considering the bigger picture — in this case: are there circumstances under which an improved version of UnitWalk would be the undisputed solver of choice? The answer in retrospect is no; the reader should regard with suspicion the fact that the only benchmarks on which QingTing dominates the performance of other solvers are those of our own creation (however unintended this may have been).

The switching strategy, an important concept presented here, also has less impact on the bigger picture than we had hoped. What is really needed is a switching strategy that also takes DPLL-based solvers into account, but the basis for such a strategy is not immediately clear. It should also be instructive to see if

Benchmark	Best Local Search Solver	Best DPLL Solver	Dominating Solver
sched06s_v00828	QingTing1 (0.03/0.04)	sato (4.52/18.3)	QingTing1
sched07s_v01386	QingTing1 (0.10/0.09)	chaff (13.4/11.2)	QingTing1
bw_large_a	QingTing1 (0.12/0.12)	chaff (0.01/0.01)	chaff
bw_large_b	QingTing1 (2.93/3.11)	chaff (0.09/0.04)	chaff
logistics_d	QingTing1 (0.75/0.45)	chaff (0.34/0.13)	chaff
sched04z07s_v00655	QingTing1 (0.73/0.98)	chaff (28.9/23.4)	QingTing1
sched04z08s_v01094	QingTing1 (1.79/1.98)	chaff (165/192)	QingTing1
sched04z06s_v00354	QingTing1 (0.03/0.03)	chaff (24.2/37.1)	QingTing1
flat200-87	WalkSAT (4.77/5.08)	chaff (1.04/1.45)	chaff
flat200-33	WalkSAT (1.43/1.27)	chaff (0.74/0.72)	chaff
flat200-79	WalkSAT (5.04/4.88)	chaff (0.85/1.01)	chaff
par16-4-c	UnitWalk1 (75.3/68.6)	chaff (1.33/0.82)	chaff
par16-5-c	UnitWalk1 (69.0/55.8)	chaff (2.19/2.07)	chaff
logistics_c	WalkSAT (8.81/8.00)	chaff (0.37/0.08)	chaff
logistics_a	WalkSAT (1.56/1.32)	chaff (0.14/0.03)	chaff
logistics_b	WalkSAT (2.91/2.47)	chaff (0.14/0.04)	chaff
uf250-1065_027	WalkSAT (0.11/0.09)	satoL (26.5/19.6)	WalkSAT
uf250-1065_034	WalkSAT (0.99/0.95)	satoL (27.5/16.0)	WalkSAT
uf250-1065_087	WalkSAT (0.15/0.15)	satoL (30.9/19.5)	WalkSAT
hgen2_v250...093	WalkSAT (0.58/0.59)	Unknown ^(a)	WalkSAT
hgen2_v250...089	WalkSAT (27.5/24.9)	Unknown ^(a)	WalkSAT
dlx2_cc_a_bug39	WalkSAT (4.18/3.09)	chaff (1.49/1.42)	chaff
dlx2_cc_a_bug40	WalkSAT (2.10/2.15)	chaff (3.59/1.12)	WalkSAT
dlx2_cc_a_bug17	WalkSAT (2.44/1.82)	chaff (0.68/1.25)	chaff

^a All DPLL solvers time out at 1800 seconds for almost all instances of these classes.

Fig. 8. Performance comparisons between local search solvers (*QingTing1*, *UnitWalk1*, *UnitWalk2*, *WalkSAT*) and DPLL solvers (*chaff*, *sato*, *satoL*). The time-out value is set to 1800 seconds.

QingTing can be significantly improved with the use of learning techniques, such as clause recording, that are behind the effectiveness of state-of-the-art DPLL solvers.

Acknowledgments. The experiments, as reported in this paper, could not have taken place without the SAT solvers *chaff*, *sato*, *satoL*, *UnitWalk* and *WalkSAT*. We thank authors for the ready access and the exceptional ease of installation of these software packages.

References

1. Daniel Le Berre and Laurent Simon. SAT Solver Competition, in conjunction with 2003 SAT Conference, May 2003. For more information, see www.satlive.org/SATCompetition/2003/comp03report/index.html.
2. H. Hoos and T. Stuetzle. SATLIB: An online resource for research on SAT, 2000. For more information, see www.satlib.org.
3. Daniel Le Berre. SAT Live! Up-to-date links for the SATisfiability problem, 2003. For more information, see www.satlive.org.

4. Laurent Simon. Sat-Ex: The experimentation web site around the satisfiability , 2003. For more information, see www.lri.fr/~simon/satex/satex.php3.
5. Holger H. Hoos and Thomas Stützle. Local Search Algorithms for SAT: An Empirical Evaluation. *Journal Of Automated Reasoning*, 24, 2000.
6. B. Selman and H. Kautz. WalkSAT Homepage: Stochastic Local Search for Satisfiability, 2002. The source code is available at www.cs.washington.edu/homes/kautz/walksat/.
7. David A. McAllester, Bart Selman, and Henry A. Kautz. Evidence for invariants in local search. In *AAAI/IAAI*, pages 321–326, 1997.
8. B. Selman, H. Kautz, and B.Cohen. Noise Strategies for Improving Local Search. In *Proceedings of AAAI'94*, pages 46–51. MIT Press, 1994.
9. E. Hirsch and A. Kojevnikov. UnitWalk: A new SAT solver that uses local search guided by unit clause elimination. In *Electronic Proceedings of Fifth International Symposium on the Theory and Applications of Satisfiability Testing*, 2002.
10. Matthew Moskewicz, Conor Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *IEEE/ACM Design Automation Conference (DAC)*, 2001. Version 1.0 of Chaff is available at www.ee.princeton.edu/~chaff/zchaff/zchaff.2001.2.17.src.tar.gz.
11. Hantao Zhang and Mark E. Stickel. Implementing the Davis-Putnam method. *Kluwer Academic Publisher*, 2000.
12. F. Brglez, M. F. Stallmann, and X. Y. Li. SATbed: An Environment For Reliable Performance Experiments with SAT Instance Classes and Algorithms. In *Proceedings of SAT 2003, Sixth International Symposium on the Theory and Applications of Satisfiability Testing, May 5-8 2003, S. Margherita Ligure - Portofino, Italy*, 2003. For a revised version, see www.cbl.ncsu.edu/publications/.
13. F. Brglez, X. Y. Li, and M. Stallmann. On SAT Instance Classes and a Method for Reliable Performance Experiments with SAT Solvers. *Annals of Mathematics and Artificial Intelligence, Special Issue on Satisfiability Testing*, 2003. Submitted to AMAI as the revision of the paper published at the Fifth International Symposium on the Theory and Applications of Satisfiability Testing, Cincinnati, Ohio, USA, May 2002. Available at www.cbl.ncsu.edu/publications/.
14. David Johnson. A Theoretician's Guide to the Experimental Analysis of Algorithms. pages 215–250, 2002.
15. H. Zhang and M. E. Stickel. An efficient algorithm for unit propagation. In *Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics (AI-MATH'96)*, Fort Lauderdale (Florida USA), 1996.
16. E. Hirsch and A. Kojevnikov. UnitWalk Home Page. See <http://logic.pdmi.ras.ru/~arist/UnitWalk/>.
17. Ines Lynce and Joao Marques-Silva. Efficient data structure for backtrack search sat solvers. In *Fifth International Symposium on the Theory and Applications of Satisfiability Testing*, 2002.
18. Hantao Zhang. SATO: An efficient propositional prover. In *Conference on Automated Deduction*, pages 272–275, 1997. Version 3.2 of SATO is available at <ftp://cs.uiowa.edu/pub/hzhang/sato/sato.tar.gz>.
19. F. Brglez, X. Y. Li, and M. Stallmann. The Role of a Skeptic Agent in Testing and Benchmarking of SAT Algorithms. In *Fifth International Symposium on the Theory and Applications of Satisfiability Testing*, May 2002. Available at www.cbl.ncsu.edu/publications/.
20. H. Hoos. SATLIB - The Satisfiability Library, 2003. See <http://www.satlib.org>.
21. E. Hirsch. Random generator hgen2 of satisfiable formulas in 3-CNF. See <http://logic.pdmi.ras.ru/~hirsch/benchmarks/hgen2.html>.
22. M.N. Velev. Benchmark suite SSS1.0. See <http://www.ece.cmu.edu/~mvelev>.