QingTing: A Fast SAT Solver Using Local Search and 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. In this paper, we present a new SAT solver that combines a recently proposed local search algorithm — *unitwalk* — with efficient unit propagation techniques. Unlike many other local-search SAT algorithms, *unitwalk*'s search relies heavily on unit propagation. In our solver, *QingTing*[1], unit propagation is implemented with an efficient unit propagation algorithm using an underlying lazy data structure. By comparing it to a more basic data structure, we empirically show how our approach is able to significantly reduce memory access in terms of clause and literal visits. Experiments also show that *QingTing* is up to five times faster than the original *unitwalk* solver on a wide range of benchmarks and competitive with other state-of-the-art SAT solvers.

1 Introduction

The propositional satisfiability problem, SAT, is probably the most fundamental and well-studied combinatorial problem. The Web has become the universal resource to access large and diverse directories of SAT problem instances [1], SAT discussion forums [2], and SAT experiments [3], each with links to SAT-solvers that can be readily down-loaded and installed.

In general, SAT solvers fall into two categories. In our experimental context, *complete* SAT solvers have three possible outcomes: (i) they find a satisfying assignment for the input formula, (ii) they declare the formula unsatisfiable, or (iii) they time out. Some state-of-the-art complete solvers include GRASP [4], *sato* [5], and *chaff* [6]. *Incomplete* SAT solvers only have two outcomes, finding a solution or timing out. The latter can mean either that the formula is unsatisfiable or that the search was terminated too early. Well known incomplete SAT solvers include GSAT [7] and *walksat* [8].

New competitive SAT solvers continue to be introduced. Two sophisticated examples are OKsolver [9] and 2clseq [10], both of which employ extensive computation at each node of the search tree. Recently, Hirsch and Kojevnikov [11] proposed and implemented a new local search (incomplete) algorithm called *unitwalk*. Despite its simple implementation, the *unitwalk* solver dominates some of the state-of-the-art solvers on benchmarks such as scheduling, microprocessor verification, and random 3-SAT. Unlike many other local search algorithms, *unitwalk* relies heavily on unit propagation and any speedup gained on this operation will directly boost the solver's performance. Our solver *QingTing* is based

[1] The source code is available at `http://pluto.cbl.ncsu.edu/EDS/QingTing`

on the *unitwalk* algorithm and implements unit propagation efficiently using Zhang's unit propagation algorithm [12] with *chaff*'s lazy data structure [6].

This paper is organized as follows. In the rest of this section, we introduce basic notation and the *unitwalk* algorithm. We describe how unit propagation is implemented in *QingTing* and its effect on reducing clause and literal visits in Section 2. The experimental setup and results for *QingTing* are presented in Section 3. We conclude the paper and describe some future research directions in Section 4.

1.1 Basic Notation

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 $\overline{v}$, 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:

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

When x is set true, the *unit propagation* operation, $F := F[x \leftarrow true]$, will delete $\overline{x}$ 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.

1.2 The *unitwalk* Algorithm

The *unitwalk* algorithm is shown in Figure 1. Like any typical local search algorithm, *unitwalk* generates an initial assignment for the variables at random and then modifies it by complementing (flipping) the value of some variable in each step. An iteration of the outer loop is called a *period*. At the beginning of a period, a random permutation of the variables is chosen and 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. The algorithm is guaranteed to make local moves by flipping at least one variable assignment within a period. 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 *unitwalk* and *QingTing*, MAX_PERIODS is set to infinity (the same setting used by Hirsch [11]), but the program is terminated if it runs for longer than 30 minutes (our "time out" setting).

```
Algorithm unitwalk
   Input: A formula F in CNF containing n variables v_1, ···, v_n
   Output: A satisfying assignment or "No solution found"
   Method:
   A := random truth assignment for n variables
   for p := 1 to MAX_PERIODS do
      π := random permutation of 1, ···, n
      G := F; flipped := false
      for i := 1 to n do
         while G contains a unit clause do
            Pick a unit clause {v_j} or {¬v_j}
            if this clause is not satisfied by A and G doesn't contain the
               opposite unit clause then flip A[j] and set flipped := true
            G := G[v_j ← A[j]]
         end do
         if variable v_π[i] still appears in G then G := G[v_π[i] ← A[π[i]]]
      end do
      if G contains no clauses, then output A and exit
      if flipped = false, then choose j randomly from 1, ···, n and flip A[j]
   end do
   Output "No solution found"
```

Fig. 1. The *unitwalk* algorithm.

2 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 the *unitwalk* solver[2]. We then present the data structure used by *QingTing*. Last, we show empirically that *QingTing*'s data structure reduces the number of memory accesses in terms of clause and literal visits.

2.1 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*.

[2] All data in this section are based on *unitwalk* 0.944. It is available at `http://logic.pdmi.ras.ru/~arist/UnitWalk`.

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.

2.2 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 [13]. Some examples include *sato*'s Head/Tail Lists [14] and *chaff*'s Watched Literals [6]. Head/Tail Lists and Watched Literals are very similar. In our solver *QingTing*, 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 [12].

2.3 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.

Unitwalk and *QingTing_AL* use counter-based adjacency lists; *QingTing* 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 *unitwalk*, 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 *unitwalk*, *QingTing_AL*, and *QingTing*, *QingTing* has fewer clause and literal visits, especially for `queen19` and `bw_large_b`. For `uf250-1065_087`, *QingTing* has fewer clause and literal visits, but runs slower than *unitwalk*. 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 `queen19` benchmark

Solver	Periods	Flips	Visits	runtime
unitwalk	29/34	363/228	N/A	0.27/0.30
QingTing_AL	29/30	358/207	6.30e5/6.67e5	–/–
QingTing	22/23	316/160	8.50e4/8.61e4	0.05/0.05

128 experiments on the `bw_large_b` benchmark

Solver	Periods	Flips	Visits	runtime
unitwalk	204/207	1.20e4/1.19e4	N/A	3.00/3.03
QingTing_AL	176/173	1.03e4/9.64e3	5.71e6/5.59e6	–/–
QingTing	200/234	1.19e4/1.35e4	3.46e6/4.03e6	1.71/2.00

128 experiments on the `uf250-1065_087` benchmark

Solver	Periods	Flips	Visits	runtime
unitwalk	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	–/–
QingTing	1.10e4/1.15e4	6.15e5/6.41e5	3.27e7/3.40e7	15.8/16.5

Fig. 2. Performance comparisons of *unitwalk*, *QingTing_AL* and *QingTing* on three benchmark instances.Table entries report mean/standard-deviation of each metric.

We devised two versions of *QingTing*, implementing the *unitwalk* algorithm with each of the data structures described above. We compared their performances with the *unitwalk* solver. We ran all our experiments on a P-II@266 MHz Linux machine with 196 MBytes of physical memory. In local search solvers such as *unitwalk* and *QingTing*, significant variability of performance metrics can be observed by simply repeating experiments on the same instance, randomly choosing the seed. In order to make our experiments have much statistical significance, we run each solver 128 times on each benchmark with different seeds and report the sample mean and standard deviation of periods, flips and runtime (the time to find the first solution) for each solver in Figure 2. It shows that all three solvers have approximately the same number of periods and flips for the three benchmarks from 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.

3 Experimental Setup and Results

In this section, we evaluate the performance of *QingTing* more thoroughly by comparing it with two versions of *unitwalk* and two other state-of-the-art complete solvers, *sato* and *chaff*. But first we take a more detailed look at the behavior of *unitwalk* (version 0.944) and *QingTing*.

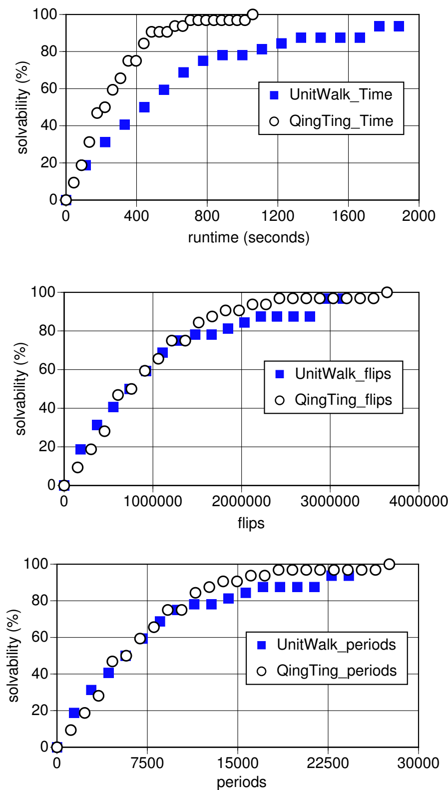
Drawing on our experiences with solver variability (as reported in [15]) experiments involving multiple runs were done in three different ways: (1) identical input but with varying random seed, (2) multiple instances in which clauses and literals are randomly permuted (an I class), and (3) multiple instances in which variables are randomly renamed and complemented, and clauses and literals permuted (a PC class). For *unitwalk* and *QingTing* all of these methods yielded statistically equivalent results. However, the other two solvers are deterministic, ruling out method (1). Furthermore, only the PC class gives a true average for various performance measures (see [15]). Thus, all results reported in this section are for PC class experiments.

3.1 Detailed Comparison of *unitwalk* and *QingTing*

To demonstrate the effectiveness of the lazy data structures, we took one of the most challenging benchmarks for *unitwalk*, *bw_large_c*, and looked at the distribution of three random variables on 32 runs each of *unitwalk* and *QingTing* — 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 *QingTing* are less than half those of *unitwalk*. As the chart at the top left of Figure 3 illustrates, within just over 200 seconds, more than half of the *QingTing* executions have finished, while only a third of the *unitwalk* ones are done. At around 500 seconds *QingTing*'s executions are almost all finished and only half of *unitwalk*'s are. At 1800 seconds, all *QingTing*'s executions are finished and two instances time out for UnitWalk.

3.2 Comparisons with Other Solvers

Our experiments comparing *unitwalk* and *QingTing* with other solvers are based on two sets of reference benchmarks, all satisfiable. For the benchmarks in the first set, *unitwalk* and *QingTing* perform significantly better than the complete solvers: the queens problem subset (`queen16` and `queen19`), the scheduling problem subset (`sched06` and `sched07`), and the random 3-SAT problem subset (`uf250-1065_027` and `uf250-1065_087` are two instances from the `uf250-1065` benchmark set). For the benchmarks in the second set, *unitwalk* and *QingTing* perform relatively poorly when compared to the complete solvers, although the performance of *QingTing* is consistently better: the blocks-world problem subset (`bw_large_b` and `bw_large_c`), and the logistic planning problem subset



Experiments with a SAT-solver A on N instances from a well-defined equivalence class of CNF formulas [15] show that parameters such as *flips* and *runtime* are random variables X with a cumulative distribution $F_X^A(x)$. In the revised version of [15], we define the estimate of $F_X^A(x)$ as the *solvability function* $\mathcal{S}^A(x)$:

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

The solvability functions on the left are based on experiments with *unitwalk* and *QingTing* 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.

	unitwalk mean/stdev	*QingTing* mean/stdev
runtime	591/554	267/219
flips	986510/923651	924915/758691
periods	7613/7121	6988/5746

Fig. 3. Solvability functions of *QingTing* and *unitwalk* induced by solving 32 PC-class instances of `bw_large_c`.

(`logistics_a`, `logistics_b` and `logistics_c`). For each reference benchmark, we performed experiments on the reference as well as 32 instances from the associated PC-class. Results for *sato* and *chaff* on all but the logistics subset are from earlier experiments with an identical setup [15].

We also include the latest version (0.981) of *unitwalk* and denote it by *UW2* in the tables (version 0.944 is called *UW1*). *UW2* combines the *unitwalk* algorithm with the *walksat* algorithm[3].

Figure 4 shows the results (execution time only) of experiments with other satisfiable benchmarks. The most dramatic results are those for the queens-problem and scheduling instances, where *QingTing* is nearly five times faster than *unitwalk* and becomes the new champion among the solvers we tested. The lack of improvement over *unitwalk* for random instances, as already noted in Figure 2, appears to be due to the overhead of frequent data-structure updates. We observe that even though we save modestly on the number of clause and

[3] See `http://logic.pdmi.ras.ru/~arist/UnitWalk` for more information.

Table entries report the mean/standard-deviation of runtime.

Solvers	queen16	queen19	sched06	sched07	uf250..27	uf250..87
QT	0.04/0.03	0.08/0.05	0.04/0.03	0.10/0.09	10.2/10.8	16.7/17.2
UW1	0.10/0.10	0.39/0.37	0.20/0.19	0.40/0.42	6.84/6.46	12.3/12.5
UW2	0.18/0.18	0.52/0.49	0.29/0.29	0.67/0.70	0.48/0.41	0.39/0.29
sato	0.07/0.09	0.21/0.58	4.52/18.3	433/1153	93.3/106	148/213
chaff	1.11/0.48	98.4/46.4	18.0/20.5	13.4/11.2	69.4/62.0	614/311

Solvers	bw_large_b	bw_large_c	logistics_a	logistics_b	logistics_c
QT	2.93/3.11	267/219	130/177	5.25/4.41	107/105
UW1	4.27/3.02	591/555	374/388	12.4/11.6	206/216
UW2	6.23/5.44	904/599	576/531	20.0/18.9	307/326
sato	0.84/0.32	78.5/31.0	1.54/1.67	0.96/0.51	19.1/63.7
chaff	0.09/0.04	3.96/2.18	0.14/0.03	0.14/0.04	0.37/0.08

Fig. 4. Performance comparisons for *unitwalk* 0.944 (*UW1*), *unitwalk* 0.981 (*UW2*), *QingTing* (*QT*), *sato*, and *chaff* on 32 PC-class instances for each of the respective benchmarks.

literal visits, significant amount of time (around 27%) is spent on updating the dynamic data-structure in *QingTing*.

UW2 is slower than *UW1* on queens problems and scheduling instances, but improves dramatically on the random 3-SAT problem instances. Intuitively, *walksat*, which doesn't utilize unit propagation, can't take advantage of the structure of the benchmark instances; therefore, it works poorly on highly-structured benchmarks such as queens-problem and scheduling. However, *walksat* appears to be very effective on random 3-SAT problem instances, which lack structure.

The lower table in Figure 4 shows results on highly-structured benchmarks that have traditionally put local-search solvers at a disadvantage over *complete* solvers. Even though neither *QingTing* nor *unitwalk* is competitive with the other solvers on these benchmarks, *QingTing* consistently outperforms both versions of *unitwalk*. Again, as expected, *UW2* runs slower than *UW1* on all these benchmarks. It actually times out on four instances from `bw_large_c` and two instances from `logistics_a`. We hope to incorporate some of the better algorithmic features of the complete solvers into future versions of *QingTing*, as we have already done with data structures.

4 Conclusions and Future Research

We have introduced a new solver *QingTing* based on the *unitwalk* algorithm. With an efficient implementation using the Watched Literals data structure, *QingTing* runs up to five times faster than the original *unitwalk* SAT solver. We have also experimentally measured the effect data structures have on the number of clause and literal visits required for unit propagation. We believe the following research directions are worth pursuing:

– Improve *QingTing*'s performance on benchmarks from the domain of block world and logistic planning by analyzing the structure of these benchmarks and *QingTing*'s corresponding behaviors.
– Incorporate learning techniques such as clause recording; these have been widely used in complete solvers. It would be interesting to see their impact on local-search-based solvers such as *QingTing*.
– Study how the *walksat* algorithm is used in *UW2*. It appears to be very effective on random 3-SAT problems.

References

1. Hoos, H., Stuetzle, T.: Satlib: An online resource for research on sat (2000)
2. Le Berre, D.: SAT Live! Up-to-date links for the SATisfiability problem (2003) For more information, see `http://www.satlive.org`.
3. Simon, L.: Sat-Ex: The experimentation web site around the satisfiability (2003) For more information, see `http://www.lri.fr/~simon/satex/satex.php3`.
4. Silva, J.P.M.: Search algorithms for satisfiability problems in combinational switching circuits. Ph.D. Dissertation, EECS Department, University of Michigan (1995)
5. Zhang, H., Stickel, M.E.: Implementing the Davis-Putnam method. Kluwer Academic Publisher (2000)
6. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: IEEE/ACM Design Automation Conference (DAC). (2001) Version 1.0 of Chaff is available from `http://www.ee.princeton.edu/ chaff/zchaff/zchaff.2001.2.17.src.tar.gz`.
7. Selman, B., Kautz, H.: Domain-independent extension to GSAT: Solving large structured satisfiability problems. In: Proceedings of IJCAI-93, 13th International Joint Conference on Artificial Intelligence, Sidney, AU (1993) 290–295
8. Selman, B., Kautz, H.: Stochastic Local Search for Satisfiability (2002) For more information, see `http://www.cs.washington.edu/homes/kautz/walksat`.
9. Kullmann, O.: OKsolver Version 1.2 (2002) For more information, see `http://cs-svr1.swan.ac.uk/~csoliver/OKsolver.html` .
10. Bacchus, F.: Exploring the computational tradeoff of more reasoning and less searching. In: Fifth International Symposium on the Theory and Applications of Satisfiability Testing. (2002)
11. Hirsch, E., Kojevnikov, A.: UnitWalk: A new SAT solver that uses local search guided by unit clause elimination (2001) PDMI preprint 9/2001, Steklov Institute of Mathematics at St.Petersburg, 2001.
12. Zhang, H., Stickel, M.E.: 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)
13. Lynce, I., Marques-Silva, J.: Efficient data structure for backtrack search sat solvers. In: Fifth International Symposium on the Theory and Applications of Satisfiability Testing. (2002)
14. Zhang, H.: SATO: An efficient propositional prover. In: Conference on Automated Deduction. (1997) 272–275 Version 3.2 of SATO is available from `ftp://cs.uiowa.edu/pub/hzhang/sato/sato.tar.gz`.
15. Brglez, F., Li, X., Stallmann, M.: 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. (2002) A revised version, under review with AMAI, is available on the Web as a technical report 'On SAT Instance Classes and a Method for Reliable Performance Experiments with SAT Solvers'. See `http://www.cbl.ncsu.edu/publications/` .