NobleCount: Revisiting a Forgotten Data Structure for Local Search in SAT

Anonymous Author(s)
Anonymous Affiliation

Abstract

Local Search (LS) solvers for SAT can surpass CDCL solvers on some prominent satisfiable industrial and combinatorial benchmarks. They operate by *flipping* variable assignments and maintaining per-variable scores incrementally to guide variable selection, so performance is largely determined by how efficiently each flip is executed. We revisit a largely forgotten data structure that reduces per-flip overhead by eliminating costly clause scans. We dub this scheme `NobleCount`, integrate it into the leading LS solver ProbCCAnr, and evaluate it on industrial FCC spectrum-repacking instances and the SC22 anniversary track, two standard testbeds for modern LS solvers. ProbCCAnr with `NobleCount` achieves new state-of-the-art results on both benchmarks. On FCC, where it already overtook not only other LS solvers but also leading CDCL solvers, employing `NobleCount` further reduces PAR2 by 24.8% at a 20s timeout, with consistent PAR2 gains across all timeouts up to our final 300s timeout. On SC22, at longer timeouts where another LS solver, TaSSAT, originally overtakes ProbCCAnr, `NobleCount` lifts ProbCCAnr to outscore TaSSAT.

2012 ACM Subject Classification Mathematics of computing → Solvers

Keywords and phrases SAT solving, Local Search, Data Structures

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

1 Introduction

The Boolean Satisfiability Problem (SAT) is a foundational problem in Computer Science that has been studied extensively due to its wide range of applications, including hardware verification, combinatorial optimization, and planning [10].

Conflict-Driven Clause Learning (CDCL) [13] is the dominant paradigm in modern SAT solving, excelling at systematically exploring the search space and solving structured industrial instances.

Local Search (LS) solvers adopt a complementary, incomplete strategy that operates on complete variable assignments. Starting from an initial assignment, LS iteratively *flips* the value of selected variables to reduce the number of unsatisfied clauses, maintaining a complete assignment throughout the search. Although unable to prove unsatisfiability, LS techniques have significantly influenced solver design: leading CDCL solvers such as CaDiCaL [6] routinely integrate LS phases to enhance performance.

Moreover, stand-alone LS solvers are state of the art on certain classes of satisfiable industrial and combinatorial benchmarks [14]. In particular, the ProbCCAnr [11] LS solver outperforms leading CDCL solvers on the US Federal Communications Commission (FCC) benchmarks [14], a prominent family of real-world spectrum repacking instances arising from the FCC's multi-billion-dollar incentive auction.

This work improves ProbCCAnr, the current state-of-the-art LS solver for SAT. By revisiting a largely forgotten data structure that reduces per-flip overhead, we achieve improved results on both the FCC and SC22 anniversary [3] benchmarks. The improvements are observed across all evaluated timeouts, as `NobleCount` delivers a consistent per-instance speedup.

1.1 Local Search and `NobleCount`

The core operation of any Local Search (LS) solver is the *flip*: at each step, a variable's assignment is inverted. Because LS operates on complete assignments, every literal is either satisfied or falsified, and every clause is either satisfied or unsatisfied at all times.

The choice of which variable to flip is guided by two per-variable scores. The first, `make`(x), denotes the number of currently unsatisfied clauses that would become satisfied if variable x were flipped. The second, `break`(x), denotes the number of currently satisfied clauses that would become unsatisfied. Different LS algorithms combine these scores, typically together with randomization, to guide the search and to escape local optima. After each flip, `make` and `break` must be updated for all variables affected.

To correctly maintain these scores, the solver uses a full occurrence list that maps each literal to the clauses in which it appears, together with a per-clause Satisfied Literals Count (SLC), which records how many literals in the clause are currently true. The SLC is stored in a compact, contiguous array independent of the clause buffer and therefore highly cache-friendly. When a variable is flipped, the solver consults the SLC for each clause in which it appears to determine what score updates are required — both for itself and for other variables appearing in those clauses — accessing the clause buffer only when necessary. In particular, if the SLC exceeds two, no update is required and the clause access is skipped entirely.

As a result, flip performance is largely determined by how often the solver is forced to access the clause buffer rather than resolving updates using only the SLC. This implies that the dominant cost comes from transitions between small SLC values in satisfied clauses. It turns out that the cost of the key $1 \to 2$ and $2 \to 1$ transitions can be further reduced by storing additional per-clause data alongside the SLC.

Specifically, Fukunaga [12] introduced the `watch1` scheme, which caches the *critical variable*—the sole satisfying literal of a 1-satisfied clause—and thereby makes the $1 \to 2$ transition cheap by avoiding clause access.

However, `watch1` leaves the $2 \to 1$ transition costly. This transition is frequent: Fukunaga [12] measured it at 28–41% of all clause-state transitions across benchmarks, and on hard combinatorial instances it can approach almost 50% [1].

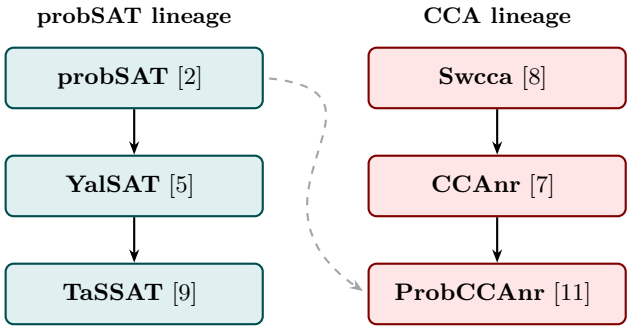

Figure 1 Two independent LS solver lineages. The probSAT lineage carries `NobleCount` throughout; the CCA lineage never adopted it.

This limitation can be addressed: first introduced in the SAT solver NanoSAT [4], augmenting the SLC with the XOR of all satisfied literal identifiers allows clause access during the $2 \to 1$ transition to be omitted. We dub this scheme `NobleCount`. Balint et al. [1] adapted `NobleCount` to the LS setting, integrating it into probSAT [2] and Sparrow and reporting speedups exceeding 2× on hard combinatorial instances from the SAT Challenge

2012. `NobleCount` has since been retained in LS solvers derived from that line of work: YalSAT [5] inherits it from probSAT, and TaSSAT [9], built on YalSAT, carries it forward.

Yet, `NobleCount` was never adopted by the independent CCA solver lineage [8, 7], which now dominates LS SAT solving through ProbCCAnr [11]. ProbCCAnr adopted the probabilistic selection strategy of probSAT [2] but not the data structure improvement introduced later by Balint et al. [1]. Figure 1 illustrates the two independent lineages.

1.2 Our Contribution

In summary, our contributions are as follows:

- We revisit `NobleCount`, a core data structure absent from the current state-of-the-art LS solver ProbCCAnr, and provide in-depth intuition and analysis of its effectiveness. Given its minimal integration effort and the consistent gains we observe, we make a strong, data-driven case for its broader adoption in modern LS solvers and beyond.
- We integrate `NobleCount` into ProbCCAnr and achieve new state-of-the-art results on the FCC benchmark family, improving over the best LS performance reported in [11], which already exceeded leading CDCL solvers. `NobleCount` reduces PAR2 by 24.8% at a 20s timeout, with consistent gains across all timeouts.
- Furthermore, we evaluate `NobleCount` on the SC22 anniversary benchmarks. Originally, TaSSAT overtook the ProbCCAnr baseline at longer timeouts, but with `NobleCount` integrated, ProbCCAnr outscores TaSSAT, establishing a new state-of-the-art on this benchmark family for LS solvers and confirming that `NobleCount` is a data-structure optimization largely orthogonal to heuristic design.

The remainder of this paper is organized as follows. Section 2 introduces LS and score maintenance. Section 3 presents the `NobleCount` scheme. Section 4 reports our experimental results. Section 5 concludes.

2 Preliminaries

2.1 Problem Definitions

The Boolean Satisfiability Problem (SAT) asks whether there exists an assignment $\alpha : V \to \{1, 0\}$ to a set of Boolean variables $V = \{x_1, \ldots, x_n\}$ that satisfies a propositional formula in Conjunctive Normal Form (CNF). A CNF formula $F = C_1 \wedge \cdots \wedge C_m$ is a conjunction of m clauses, where each clause $C_i = (l_1 \vee \cdots \vee l_k)$ is a disjunction of literals, and each literal l is either a variable x_j or its negation $\neg x_j$. A clause is *satisfied* under α if at least one of its literals evaluates to true; otherwise it is *unsatisfied*. In practice, literals are represented as integers: for example, variable x_j as j and $\neg x_j$ as $-j$; we denote by $\mathrm{int}(l)$ the integer representation of literal l.

2.2 Local Search for SAT

LS solvers maintain per-variable scores that guide variable selection. Formally, for a variable x under the current assignment α, the ***`break`*** value is the number of currently satisfied clauses that would become unsatisfied if x were flipped, and the ***`make`*** value is the number of currently unsatisfied clauses that would become satisfied. The ***`score`*** of x is defined as $\texttt{score}(x) = \texttt{make}(x) - \texttt{break}(x)$. Different algorithms combine these quantities differently. WalkSAT [15] uses only `break`, while probSAT [2] weights variable selection inversely by `break`. ProbCCAnr employs a two-mode strategy: a greedy mode guided by `score` via

configuration-checking heuristics, and a diversification mode using `break`-based probabilistic selection [11]. TaSSAT, built on YalSAT [5], retains probabilistic selection and augments it with a clause weight transfer rule to escape local minima [9].

2.3 Clause-State Transitions

When variable x is flipped, the solver iterates over all the clauses in x's occurrence lists: clauses where x appears as a now-satisfied literal have their SLC incremented by one, and clauses where x appears as a now-falsified literal have their SLC decremented by one. Following Fukunaga [12], we partition all clause-state changes caused by a literal flip into the following six exhaustive transitions. When the flip falsifies a literal in clause C, the SLC drops from 1 to 0 ($\underline{1\to0}$), from 2 to 1 ($\underline{2\to1}$), or from more than 2 by one ($\underline{2^-}$). When the flip satisfies a literal, the SLC rises from 0 to 1 ($\underline{0\to1}$), from 1 to 2 ($\underline{1\to2}$), or increases from 2 or more ($\underline{2^+}$).

Not all transitions require the same amount of work. Transitions with SLC > 2 (i.e., 2^+ and 2^-) require no score updates and no access to the clause's literal array. The boundary transitions $0\to1$ and $1\to0$ require updating `make` for all variables in the clause, an $O(k)$ operation, where k denotes the clause length, as the clause crosses the satisfied/unsatisfied boundary. The transitions $1\to2$ and $2\to1$ require updating `break` only for the critical variable of C (i.e., the sole satisfying literal of a 1-satisfied clause): on $1\to2$, the current critical variable loses its `break` responsibility, and on $2\to1$, a new critical variable must assume it.

Fukunaga's empirical analysis (Table 2 in [12]) shows that the $2\to1$ and $1\to2$ transitions together account for 57–82% of all clause-state changes, with $2\to1$ alone responsible for 28–41%. The `watch1` scheme handles the $1\to2$ transition in $O(1)$ by caching the critical variable, but leaves $2\to1$ at $O(k)$, as recovering the new critical variable requires scanning the clause. Fukunaga's `watch2` scheme extends this by caching a second satisfied variable, reducing the $2\to1$ transition to $O(1)$ at the cost of rendering 2^- transitions $O(k)$. `NobleCount`, described in Section 3, achieves $O(1)$ for both the $1\to2$ and $2\to1$ transitions while keeping 2^- transitions $O(1)$, with minimal overhead.

3 The NobleCount Scheme

`NobleCount` is a clause-maintenance scheme that uses full occurrence lists together with a compact per-clause data structure to track satisfied literals and to enable $O(1)$ recovery of the critical variable. The following subsections describe the data structure, its update operations, and its integration into ProbCCAnr.

3.1 The Data Structure

As described in Section 1, `NobleCount` was first introduced in NanoSAT [4] and adapted to LS by Balint et al. [1]. It maintains two per-clause fields under the current assignment α: the satisfied literal count $C(\#)$ and the XOR of the integer representations of its satisfied literals $C(\oplus)$. Formally, for clause C:

$$C(\#) = \big|\{\, \ell \in C \mid \ell \text{ true under } \alpha \,\}\big|$$

$$C(\oplus) = \bigoplus_{\ell \in C,\, \ell \text{ true under } \alpha} \mathrm{int}(\ell)$$

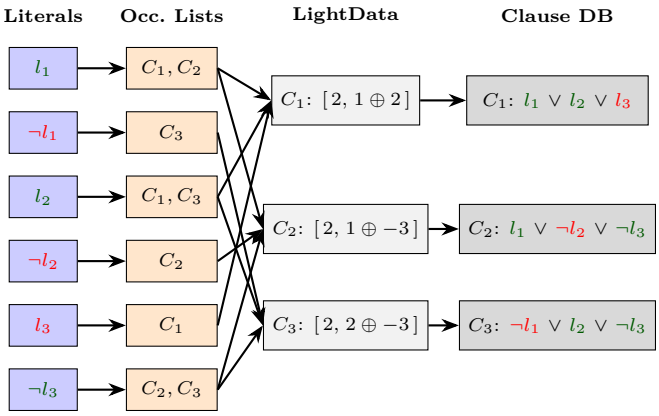

Figure 2 The `NobleCount` scheme that represents the clauses C_1, C_2, and C_3 (on the right) under the assignment $x_1 = x_2 = 1$, $x_3 = 0$. Green literals are true; red literals are false. Following the convention from Section 2, $\text{int}(\ell)$ encodes a positive literal x_j as j and its negation $\neg x_j$ as $-j$. The second field of each `LightData` pair stores the XOR of the integer representations of the true literals in that clause.

where $\text{int}(\ell)$ is the signed integer representation of literal ℓ, as defined in Section 2. The key property follows immediately: when $C(\#) = 1$, the field $C(\oplus)$ holds $\text{int}(\ell^*)$, where ℓ^* is the sole satisfied literal in C, identifying the critical variable directly as $|\,\text{int}(\ell^*)|$ with no need to scan the clause. We call this pair the ***`LightData`*** of a clause.

Two natural layouts exist for storing the `LightData` pair $[C(\#), C(\oplus)]$. In the *separate-arrays* layout, used by YalSAT and TaSSAT, the two fields are stored in independent arrays. This allows the count field to be type-compressed — to `char` (8 bit), `short` (16 bit), or `int` (32 bit) depending on the maximum clause length — reducing memory footprint, but requiring two independent array accesses that may fall in different cache lines. In the *interleaved* layout, the two fields are packed as a single 64-bit struct (32 bits each), stored contiguously per clause, keeping both fields in the same cache line on every access. The 32-bit count is sufficient for any practical clause length, and the paired layout is faster in practice; we adopt it as our default. Note that `watch1` stores a single integer per clause (the critical variable), while `NobleCount` stores two (count and XOR), doubling the per-clause metadata. In practice, this increase is small relative to the clause database and occurrence lists, and the elimination of clause-buffer accesses on the $2 \to 1$ transition more than compensates. See Figure 2 for an illustration.

3.2 Update Operations

When variable x is flipped, the `LightData` is updated uniformly across all clauses in the occurrence lists of x and $\neg x$. When literal ℓ becomes true, every clause C in its occurrence list receives:

$$C(\#) \leftarrow C(\#) + 1, \qquad C(\oplus) \leftarrow C(\oplus) \oplus \text{int}(\ell)$$

When ℓ becomes false:

$$C(\#) \leftarrow C(\#) - 1, \qquad C(\oplus) \leftarrow C(\oplus) \oplus \text{int}(\ell)$$

Since XOR is its own inverse, the same operation ($\oplus\, \text{int}(\ell)$) applies in both directions. Crucially, the update requires no conditional branching on the current clause state and no

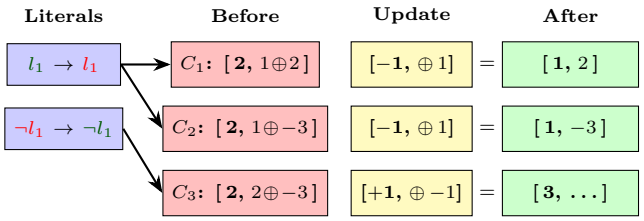

Figure 3 `LightData` update when flipping x_1 from 1 to 0. ℓ_1 becomes false: clauses C_1, C_2 receive $[-1, \oplus\, \text{int}(\ell_1)]$ where $int(\ell_1) = 1$. $\neg\ell_1$ becomes true: clause C_3 receives $[+1, \oplus\, \text{int}(\neg\ell_1)]$ where $int(\neg\ell_1) = -1$. After the update, $C_1(\#) = C_2(\#) = 1$; the XOR fields directly identify the surviving literals: $C_1(\oplus) = 2 = \text{int}(\ell_2)$ and $C_2(\oplus) = -3 = \text{int}(\neg\ell_3)$.

access to the clause literal array. After the update, if $C(\#) = 1$ then $C(\oplus)$ directly identifies the surviving literal whose `break` value must be updated, all in $O(1)$.

Continuing the example from Figure 2, Figure 3 illustrates flipping x_1 from 1 to 0: clauses C_1 and C_2 drop to $C(\#) = 1$ and immediately identify the surviving literals l_2 and $\neg l_3$, respectively, enabling direct `break` updates without touching the clause database.

3.3 Integration into ProbCCAnr

`NobleCount` can be integrated into ProbCCAnr by replacing the per-clause critical-variable field with the `LightData` pair; no changes to the variable selection heuristic, clause weighting, or any other solver component are required. The flip routine follows the structure of Balint et al. [1] (Algorithm 2), adapted for the interleaved struct layout.

4 Evaluation

4.1 Setup

All experiments were run on a server equipped with an Intel Xeon Gold 6230 CPU at 2.10 GHz with a 32 GB memory limit, compiled at optimization level `O3`. Each solver was run once per instance with a fixed random seed, with a maximum timeout of 300s. PAR2 scores penalize unsolved instances at twice the timeout. We evaluate three configurations throughout: **NC** (`ProbCCAnr` with `NobleCount`, interleaved struct), **Base** (unmodified `ProbCCAnr`), and **TS** (TaSSAT). This naming is used consistently in all tables and figures.

4.2 FCC Benchmark

The FCC benchmark comprises 10000 instances arising from US Federal Communications Commission spectrum repacking problems, including both satisfiable and unsatisfiable instances.[1] As LS solvers cannot prove unsatisfiability, results reflect satisfiable instances only. This benchmark is industrially important, and ProbCCAnr already outperforms all leading SAT solvers on it, including both LS and CDCL, establishing state-of-the-art performance [11]. Our **Base** solves slightly fewer instances overall than reported in the original ProbCCAnr paper, consistent with the different hardware used in our experiments.

Table 1 shows that **NC** consistently achieves the best PAR2 score across all timeouts, reducing PAR2 by 24.8% relative to **Base** at 20s and by 9.5% at 300s. The gains are uniform

[1] `https://www.cs.ubc.ca/labs/beta/www-projects/SATFC/cacm_cnfs.tar.gz`

Table 1 PAR2 and solve rate on the FCC benchmark. Bold indicates best per column.

	20s		100s		300s	
	Solved	**PAR2**	**Solved**	**PAR2**	**Solved**	**PAR2**
NC	**7819**	**14586**	**7948**	**26743**	**7979**	**86377**
Base	7699	19415	7939	32233	7971	95479
TS	6600	61599	7273	187648	7675	362836

across time limits and are most pronounced at short timeouts, where additional solved instances have the largest effect on PAR2.

Figure 4 shows the per-instance flip throughput (flips/s) of **NC** versus **Base** across all FCC instances at a 300s timeout, confirming that the throughput improves over almost all instances in this set.

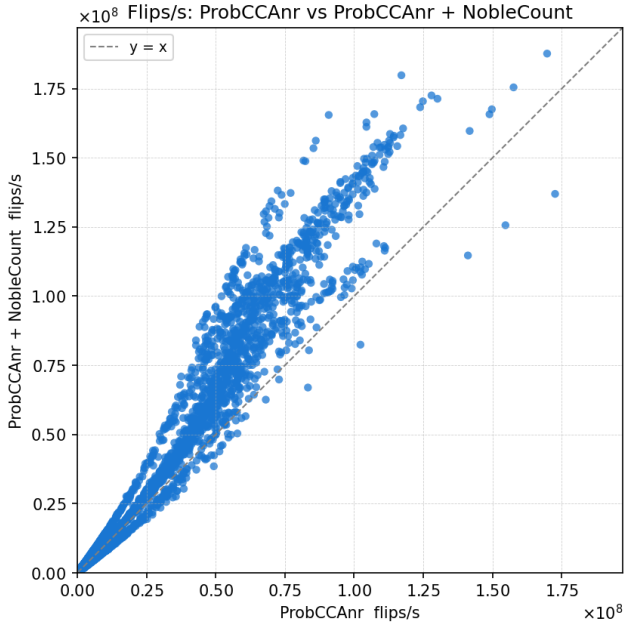

Figure 4 Per-instance flip throughput (flips/s) of **NC** (`ProbCCAnr + NobleCount`) vs. **Base** (`ProbCCAnr`) on FCC instances (300s timeout). Points above the diagonal indicate instances where **NC** achieves higher throughput.

To characterize the instance-level transition structure, we measured clause-state transition frequencies on the 7,979 solved FCC instances. The $2 \to 1$ transition accounts for an average of 24.5% of all transitions, with $1 \to 2$ at 59.4%—together comprising 83.9% of all clause-state changes, consistent with Fukunaga's measurements [12].

4.3 SC22 Anniversary Benchmark

The SC22 anniversary track [3] comprises 5355 instances from the previous 20 years of SAT competition, covering a broad range of structured and combinatorial problems. We include this benchmark as TaSSAT's original evaluation [9] used it as a primary benchmark.

Table 2 reveals a consistent pattern across short timeouts that changes at 300s. At 20s and 100s, **NC** leads on both PAR2 and solve count, with **Base** second and **TS** third—

Table 2 PAR2 and solve rate on the SC22 anniversary benchmark. Bold indicates best per column.

	20s		100s		300s	
	Solved	**PAR2**	**Solved**	**PAR2**	**Solved**	**PAR2**
NC	**710**	**3235**	**815**	**20913**	881	**74330**
Base	700	3748	808	22990	865	83244
TS	639	6061	776	31179	**885**	82902

the same ordering as on FCC. The PAR2 reduction from **NC** over **Base** is 13.7% at 20s, narrowing to 9.0% at 100s. At 300s the picture shifts: **TS** reaches 885 solved instances, drawing level with **NC** (881) and pulling ahead of **Base** (865); **NC** nevertheless retains a better PAR2 (74330 vs. 82902), confirming it solves instances faster even as solve counts converge. Figure 5 shows this progression: **NC** leads throughout the short-timeout regime, **TS** converges in solve count at 300s, and **Base**—without `NobleCount`—falls behind **TS** at this timeout. The four-instance gap at 300s suggests complementary heuristic coverage rather than strict inferiority, reinforcing that `NobleCount` is a data-structure optimization orthogonal to heuristic design. Transition statistics on the 881 solved SC22 instances show an average $2 \to 1$ frequency of 28.0%, with higher-order transitions (2^- and 2^+) accounting for a larger share (14.1% and 12.3% respectively), consistent with the more diverse instance structure of this benchmark and the smaller but still positive gains from `NobleCount`.

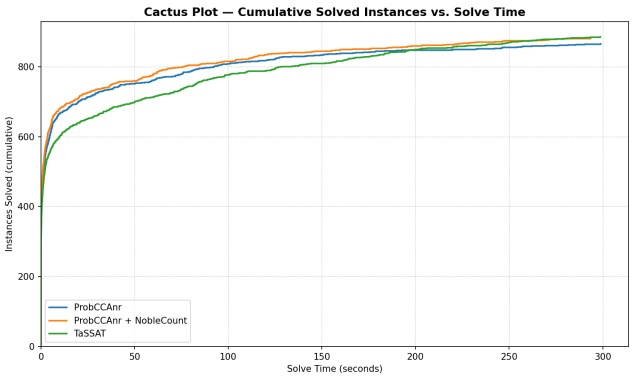

Figure 5 Cactus plot of instances solved over time on the SC22 anniversary benchmark (300s timeout). **NC** leads throughout the short-timeout regime; **TS** converges in solve count at 300s while remaining at a higher PAR2.

5 Conclusion

Local Search (LS) solvers play an increasingly important role in SAT solving and can now outperform CDCL solvers on some prominent satisfiable industrial and combinatorial benchmarks. `NobleCount` is a simple scheme for LS that, despite having been proposed over two decades ago, has remained absent from leading LS solvers. We implemented `NobleCount` in ProbCCAnr, the current leading LS solver for SAT, and obtained new state-of-the-art performance on both the industrial FCC and the SC22 anniversary benchmarks.

More broadly, `NobleCount`'s uniform, branch-free update applies the same arithmetic operation to every clause in an occurrence list, making it a promising candidate for parallel clause-state maintenance in SIMD- or GPU-based implementations for both LS and CDCL SAT solvers.

References

1. Adrian Balint, Armin Biere, Andreas Fröhlich, and Uwe Schöning. Improving implementation of SLS solvers for SAT and new heuristics for k-SAT with long clauses. In *Theory and Applications of Satisfiability Testing – SAT 2014*, volume 8561 of *Lecture Notes in Computer Science*, pages 302–316. Springer, 2014. doi:10.1007/978-3-319-09284-3_23.
2. Adrian Balint and Uwe Schöning. Choosing probability distributions for stochastic local search and the role of make versus break. In *Theory and Applications of Satisfiability Testing – SAT 2012*, volume 7317 of *Lecture Notes in Computer Science*, pages 16–29. Springer, 2012. doi:10.1007/978-3-642-31612-8_3.
3. Tomas Balyo, Marijn J. H. Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors. *Proceedings of SAT Competition 2022: Solver and Benchmark Descriptions*, volume B-2022-1 of *Department of Computer Science Series of Publications B*. Department of Computer Science, University of Helsinki, 2022.
4. Armin Biere. The evolution from LIMMAT to NANOSAT. Technical Report ETH-TR-444-2004, ETH Zurich, Formal Methods Group, 2004.
5. Armin Biere. Yet another local search solver and Lingeling and friends entering the SAT Competition 2014. In *Proceedings of SAT Competition 2014: Solver and Benchmark Descriptions*, volume B-2014-2 of *Department of Computer Science Series of Publications B*, pages 39–40. University of Helsinki, 2014.
6. Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. Cadical 2.0. In *Computer Aided Verification – CAV 2024*, Lecture Notes in Computer Science, pages 133–152. Springer, 2024. doi:10.1007/978-3-031-65627-9_7.
7. Shaowei Cai, Chuan Luo, and Kaile Su. CCAnr: A configuration checking based local search solver for non-random satisfiability. In *Theory and Applications of Satisfiability Testing – SAT 2015*, volume 9340 of *Lecture Notes in Computer Science*, pages 1–8. Springer, 2015. doi:10.1007/978-3-319-24318-4_1.
8. Shaowei Cai and Kaile Su. Configuration checking with aspiration in local search for SAT. In *Proceedings of the 26th AAAI Conference on Artificial Intelligence*, pages 434–440. AAAI Press, 2012. doi:10.1609/aaai.v26i1.8133.
9. Md Solimul Chowdhury, Cayden R. Codel, and Marijn J. H. Heule. TaSSAT: Transfer and share SAT. In *Tools and Algorithms for the Construction and Analysis of Systems – TACAS 2024*, volume 14570 of *Lecture Notes in Computer Science*, pages 34–42. Springer, 2024. doi:10.1007/978-3-031-57246-3_3.
10. Johannes K. Fichte, Daniel Le Berre, Markus Hecher, and Stefan Szeider. The silent (r)evolution of SAT. *Communications of the ACM*, 66(6):64–72, 2023. doi:10.1145/3560469.
11. Huimin Fu, Shaowei Cai, Guanfeng Wu, Jun Liu, Xin Yang, and Yang Xu. Improving two-mode algorithm via probabilistic selection for solving satisfiability problem. *Information Sciences*, 653:119751, 2024. doi:10.1016/j.ins.2023.119751.
12. Alex S. Fukunaga. Efficient implementations of SAT local search. In *Theory and Applications of Satisfiability Testing – SAT 2004*, volume 3542 of *Lecture Notes in Computer Science*, pages 287–292. Springer, 2005. doi:10.1007/11527695.
13. Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Annual Design Automation Conference (DAC'01)*, pages 530–535. ACM Press, 2001. doi:10.1145/378239.379017.
14. Neil Newman, Alexandre Fréchette, and Kevin Leyton-Brown. Deep optimization for spectrum repacking. *Communications of the ACM*, 61(1):97–104, 2018. doi:10.1145/3107548.
15. Bart Selman, Henry A. Kautz, and Bram Cohen. Noise strategies for improving local search. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*, pages 337–343. AAAI Press, 1994.