TaSSAT: Transfer and Share SAT*

Md Solimul Chowdhury, Cayden R. Codel, and Marijn J. H. Heule

Carnegie Mellon University, Pittsburgh, PA, USA
{mdsolimc,ccodel,mheule}@cs.cmu.edu

Abstract. We present TaSSAT, a powerful local search SAT solver that effectively solves hard combinatorial problems. Its unique approach of transferring clause weights in local minima enhances its efficiency in solving problem instances. Since it is implemented on top of YalSAT, TaSSAT benefits from practical techniques such as restart strategies and thread parallelization. Our implementation includes a parallel version that shares data structures across threads, leading to a significant reduction in memory usage. Our experiments demonstrate that TaSSAT outperforms similar solvers on a vast set of SAT competition benchmarks. Notably, with the parallel configuration of TaSSAT, we improve lower bounds for several van der Waerden numbers.

Keywords: Local Search for SAT · Weight Transfer · Memory Efficiency

1 Introduction

The SAT problem asks if there exists a satisfying truth assignment for a given formula in propositional logic. SAT is known to be intractable [10], but modern SAT solvers, particularly conflict-driven clause learning (CDCL) solvers, have made significant progress in solving large formulas from various application domains. When it comes to combinatorial problems, stochastic local search (SLS) solvers are often more effective than CDCL. Because SLS and CDCL solvers have complementary strengths, some SAT solvers like Kissat [7] and CryptoMiniSAT [16] combine SLS and CDCL techniques, and SLS methods play a key role in shaping the capabilities of modern SAT solvers.

SLS solvers explore truth assignments by flipping the truth value of individual variables until a solution is found or until timeout. The solver generally tries to flip variables that will minimize the number of falsified clauses. When a solver determines that no variable flip will lead to an improvement according to some heuristic or metric, it has reached a local minimum.

To escape local minima, the solver can either make random flips or adjust its internal state until improvement is possible. Despite being an effective family of algorithms for escaping local minima, Dynamic Local Search (DLS) has attracted

* The authors were supported by NSF grant CCF-2229099. Md Solimul Chowdhury was partially supported by a NSERC Postdoctoral Fellowship.

limited attention in the recent years. DLS algorithms assign weights to clauses, search to find a solution by minimizing the total amount of weight held by falsified clauses, and adjust these weights in local minima as a means of escaping them.

The tool we present in this paper is ultimately based on DDFW [15] (divide and distribute fixed weights), a DLS algorithm that dynamically transfers weight from satisfied to falsified clauses along neighborhood relationships in local minima. DDFW is remarkably effective at solving hard combinatorial problems, such as matrix multiplication [13], graph coloring [12], edge matching [11], the coloring of the Pythagorean triples [14], and finding bounds for van der Waerden numbers [3]. Notably, DDFW solves satisfiable instances of the Pythagorean triples problem in under a minute, whereas CDCL solvers take CPU years.

In this paper, we introduce Transfer and Share SAT (TaSSAT), a novel parallel SLS solver. TaSSAT implements LiWeT, a simplification of the algorithm from our recent work [9] modifying DDFW. Our implementation of TaSSAT is built on top of a leading SLS solver YalSAT [5], and it adds two new features. First, it incorporates the weight-transfer methods from LiWeT, leading to more efficient solving. Specifically, a new weight-transfer parameter allows TaSSAT to shift more clause weight in local minima, enhancing its adaptability during the search. Second, TaSSAT's parallel mode shares data structures among threads to reduce its memory footprint by up to 80%.

Our results show that TaSSAT substantially outperforms YalSAT on an extensive benchmark set of 5355 anniversary instances from the 2022 SAT Competition. Further, TaSSAT's parallel version improves the lower bounds for nine van der Waerden numbers, surpassing prior work by Ahmed et al. [3] that used 29 algorithms (including DDFW) and extensive parallelization. Our results demonstrate the clear algorithmic and practical improvements of TaSSAT.

2 Preliminaries

A SAT formula in conjunctive normal form (CNF) is a conjunction of clauses, each of which is a disjunction of literals (Boolean variables or their negations). A clause C is satisfied by a truth assignment α if α satisfies at least one of its literals, and is otherwise falsified. A formula F is satisfied by α when all of its clauses are. Clauses C and D are *neighbors* if they share a common literal.

In DLS, clauses are assigned weights, denoted as $W : \mathcal{C} \to \mathbb{R}_{\geq 0}$, representing the cost of leaving a clause falsified. The total weight of the falsified clauses is the *falsified weight*. Variables that reduce the falsified weight when flipped are called *weight-reducing variables*, while those that do not impact the falsified weight when flipped are called *sideways variables*.

DDFW starts with a random initial truth assignment and sets all clause weights to parameter w_0 ($w_0 = 8$ in the original paper [15]). It then flips weight-reducing variables until none remain. Upon reaching a local minimum, DDFW randomly chooses between making a sideways flip (if possible, and with a 15% chance) or entering the weight transfer phase. During weight transfer, each falsi-

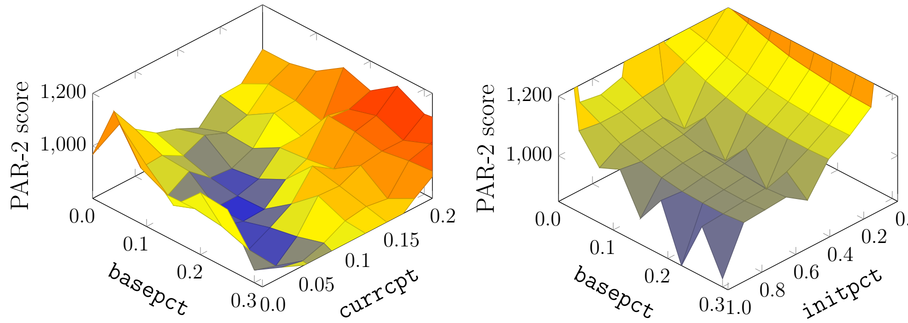

Fig. 1: PAR-2 scores for parameter searches on `initpct`, `basepct`, and `currpct`. The plots are oriented to best show the performance trends, so the axes vary.

fied clause receives a fixed weight from a maximum-weight satisfied neighbor C_S (except for 1% of the time, when a random satisfied clause is chosen instead). The amount of weight transferred from C_S depends on its weight: if $W(C_S) > w_0$, then a weight of 2 is taken; otherwise, a weight of 1 is taken.

3 LiWeT: The Linear Weight Transfer Algorithm

TaSSAT takes ideas from DDFW and distills them into an algorithm called LiWeT (Linear Weight Transfer), which is a simplification of our prior work [9]. LiWeT uses a novel linear weight transfer rule to determine how much weight to move in local minima. The rule takes three parameters: `currpct`, a multiplier on the current clause's weight; `basepct`, a multiplier on the initial weight w_0; and `initpct`, a multiplier for clauses with exactly w_0 weight. For most clauses C_s, the amount of weight that is transferred is $\texttt{currpct} \cdot W(C_S) + \texttt{basepct} \cdot w_0$. For clauses with $W(C_S) = w_0$, the amount taken is $\texttt{initpct} \cdot w_0$. As a result, `initpct` controls how much weight is initially taken from a clause.

The weight transfer rule offers two key advantages. First, the use of floating-point parameters rapidly establishes distinct weights for clauses, eliminating the need for tie-breaking near local minima and, consequently, explicit sideways flips. Second, the `initpct` parameter enables LiWeT to release a larger proportion of the total clause weight, enhancing its adaptability to challenging formulas. In DDFW and LiWeT, maximum-weight neighbors are selected for each falsified clause within local minima. Clauses with weights less than w_0 are unlikely to contribute more weight, artificially reducing the total amount of weight LiWeT can move around. The `initpct` parameter prevents this from happening.

LiWeT differs from DDFW in one other respect: in local minima, it increases the probability of choosing a randomly satisfied clause, rather than a maximum-weight neighbor, to 10%. We found that this improves overall performance.

Algorithm 1 shows LiWeT's pseudocode.

Algorithm 1: The LiWeT algorithm

Input: CNF formula F, w_0, `initpct`, `basepct`, `currpct`
Output: Satisfiability of F

$W(C) \leftarrow w_0$ for all $C \in F$
$\alpha \leftarrow$ random truth assignment on the variables in F
for *1 to MAXFLIPS* **do**
  if α *satisfies* F **then** return "SAT"
  else
    if *a weight reducing variable is available* **then**
      flip the variable that reduces the falsified weight the most
    else
      foreach *clause* $C \in F$ *falsified under* α **do**
        $C_S \leftarrow$ select a satisfied clause
        if $W(C_S) = w_0$ **then** $w \leftarrow \texttt{initpct} \cdot w_0$
        else $w \leftarrow \texttt{currpct} \cdot W(C_S) + \texttt{basepct} \cdot w_0$
        transfer w from C_S to C
return "No SAT"

To determine the effect of the three parameters, we conducted parameter searches across them. We ranged `basepct` $\in [0, 0.3]$, `currpct` $\in [0, 0.2]$, and `initpct` $\in [0, 1.0]$ with increments of 0.1, 0.05 and 0.2, respectively. Our searches were done on a combined 168 instances from the 2019 SAT Race and the 2021 and 2022 SAT competitions, each with a 900-second timeout. We picked these instances because they were solved by previous versions of LiWeT and DDFW, and thus were less likely to result in timeout.

Figure 1 shows the PAR-2 scores for two parameter searches, where a lower score indicates better performance.[1] The left plot shows that TaSSAT performs better with higher values of both `basepct` and `currpct` when `initpct` $= 1$. The optimal configuration is (`basepct`, `currpct`) $= (0.175, 0.075)$. The right plot shows that LiWeT performs best when `initpct` $= 1$ for any `basepct` value when `currpct` $= 0$. This suggests that taking all weight from satisfied clauses early in the search is crucial for better performance. We ran all subsequent TaSSAT experiments with (`initpct`, `basepct`, `currpct`) $= (1, 0.175, 0.075)$.

We conclude this section by outlining the distinctions between the algorithm presented in [9] and LiWeT, underscoring the simplifications introduced in the latter compared to the former. Compared to the algorithm from our previous work [9], LiWeT has two fewer parameters. Previously, the algorithm used two pairs of (a, c) parameters to transfer $a * W(C_S) + c$ weight from satisfied clauses C_S in local minima. One pair of (a, c) values was used when $W(C_S) > w_0$, and the other for when $W(C_S) = w_0$. In LiWeT, we replaced the second pair with `initpct`. Then based on the observation in the right plot of Figure 1, we set

[1] The PAR-2 score is defined as the average solving time, with twice the timeout as the time for unsolved instances.

initpct to 1 for performance reasons. This adjustment eliminates initpct from line 11 of Algorithm 1, transforming it into a two-parameter algorithm.

Another simplification was the the removal of sideways variable flips from LiWeT. DDFW and previous versions of our algorithm would flip sideways variables, but we found that they rarely occured with floating-point weights, and refusing to flip them didn't affect performance. Notably, these simplifications enhance the algorithmic power of LiWeT over the previous algorithm, which we demonstrate in section 5.

4 Implementation of TaSSAT and PaSSAT

We implemented TaSSAT on top of YalSAT [6], a state-of-the-art SLS solver that implements the ProbSAT algorithm [4]. As a result, our implementation benefits from the practical techniques present in YalSAT, including restart techniques. Our TaSSAT implementation[2] includes a parallel version, called PaSSAT, that improves the memory management of the parallel version of YalSAT.

Because LiWeT is computationally expensive when there are a higher number of falsified clauses, TaSSAT has an optional mode to run ProbSAT until the number of falsified clauses drops beneath a dynamically computed threshold based on the formula's size, at which point it resumes LiWeT. By default, we ran TaSSAT with this option disabled in our experiments, but we enabled it for the van der Waerden experiments.

We also improve on the parallel features in YalSAT. The main issue in the parallel version of YalSAT was that the formula data structures were not shared. As a result, each thread had to independently parse, store, and simplify the input formula, resulting in redundant computation and a bloated memory footprint. We solved this problem in PaSSAT by nominating a primary thread to parse and simplify the formula and to allocate the core data structures. Once the primary thread finishes, it hands solving off to the secondary threads, which can then jointly refer to the shared data structures.

5 Evaluation

We now present our experimental results[3] of TaSSAT against similar algorithms. Our baseline solvers are the original YalSAT (YalSAT-Prob); our DDFW-inspired, YalSAT-based solver from previous work [9] (YalSAT-Lin); a YalSAT-based implementation of DDFW (YalSAT-DDFW); and the UBCSAT implementation of DDFW (UBCSAT-DDFW). We include two DDFW implementations to check that the YalSAT version performs similarly to the UBCSAT one, despite being implemented with a different base solver.

We ran these four solvers on two benchmark sets: a set of 5355 instances from the 2022 SAT Competition's anniversary track (the anni set) [1] covering instances from the previous 20 years of competition, and a set of nine van

[2] TaSSAT source code is available at https://github.com/solimul/tassat.

[3] Details are available at https://github.com/solimul/TACAS-24-solve_details.

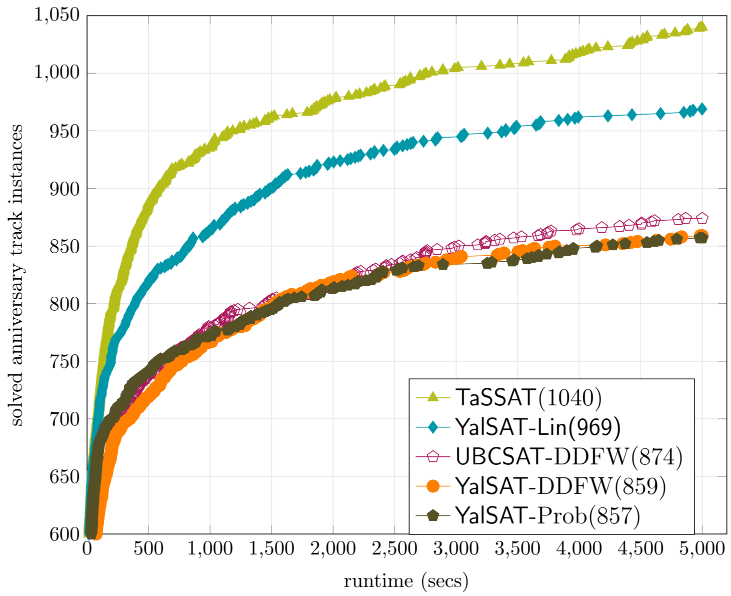

Fig. 2: Performance profiles for solver modifications on the `anni` benchmark set show that TaSSAT significantly outperforms the others. Since all solvers can quickly solve 600 instances, we start the y-axis at 600 to improve readability.

der Waerden number instances.[4] For reproducibility, we set all randomization seeds to 0. For the `anni` instances, we ran TaSSAT and our baseline solvers in the StarExec Cluster [2] with a 5000-second timeout. For the van der Waeren instances, we ran the parallel version of TaSSAT with and without the ProbSAT-LiWeT option with a 48-hour timeout on the Bridges-2 cluster [**?**] with AMD EPYC 7742 CPUs (128 cores, 512GB RAM).

Figure 2 illustrates our results for the `anni` dataset. TaSSAT performed the best by solving 1040 problem instances, surpassing YalSAT-Lin, UBCSAT-DDFW, YalSAT-DDFW, and YalSAT-Prob with 969, 874, 859, and 857 solved instances, respectively. In particular, TaSSAT solved 71 more instances than YalSAT-Lin, the solver from our previous work, showing that our algorithmic changes are, in fact, improvements. The slight difference in solve counts between UBCSAT-DDFW and YalSAT-DDFW (874 vs. 859) can be attributed to random noise.

Notably, TaSSAT exclusively solved 12 instances that no 2022 SAT Competition solver could. However, YalSAT-Prob, YalSAT-Lin, UBCSAT-DDFW, and YalSAT-DDFW solved 73, 42, 40, and 38 `anni` instances, respectively, that TaSSAT could not.

We also present new lower bounds for van der Waerden numbers by running PaSSAT. The van der Waerden number $w(2; 3, t)$ is the smallest natural number n

[4] Available at `https://github.com/solimul/vdw9`.

Table 1: Lower bounds for van der Waerden numbers $w(2;3,t)$.

t	31	32	33	34	35	36	37	38	39
Ahmed et al. [3]	930	1006	1063	1143	1204	1257	1338	1378	1418
Our work	**953**	**1011**	**1071**	**1145**	**1208**	**1260**	**1341**	**1380**	**1419**

where for any partition of $\{1,\ldots,n\}$ into P_0 and P_1, either P_0 contains a 3-term arithmetic progression or P_1 contains a t-term arithmetic progression. In Table 1, we present in the top row previously-known lower bounds for $w(2;3,t)$ for $31 \leq t \leq 39$.

The best lower bounds are obtained when PaSSAT leverages TaSSAT with the activation of the ProbSAT-LiWeT toggle and integrates YalSAT-style restarts. This configuration solves all 9 `vdw` benchmarks, pushing the lower bounds of these 9 numbers to values that are highlighted in the bottom row of Table 1. In contrast, using the default TaSSAT configuration, PaSSAT solves 7 `vdw` benchmarks, establishing same lower bounds for all the numbers shown in the bottom row of Table 1, except for $w(2;3,32)$ and $w(2;3,37)$. Hence, this version enhances the lower bounds for $w(2;3,32)$ and $w(2;3,37)$ to 1010 and 1340, respectively, just 1 short of their best-evaluated lower bounds. The performance of TaSSAT-Prob-LiWeT compared to TaSSAT-LiWeT is evident in their respective average PAR-2 scores, with values of 31,943 and 91,744.

Putting these results into perspective, Ahmed et al. [3] were unable to solve any of these `vdw` instances, despite employing 29 algorithms and extensive parallelization. Notably, the best result attained by Ahmed et al. using only SLS methods for $w(2;3,31)$ was 919. We improved this bound to 953 These results emphasize the unique algorithmic strengths of our solver.

In addition to improved solving, PaSSAT achieves significant memory reduction compared to our previous parallel solver [9]. Across the seven `vdw` benchmarks solved by both PaSSAT and the parallel solver, the average memory reduction is substantial, decreasing from 3.2 GB to 686.17 MB, a nearly 80% reduction. The reduction held even for the largest problem instance ($t = 39$), where the memory footprint decreased by nearly 80%, from 4.42 GB to 966 MB.

Code and Data Availability Statement

The code and data that support the contributions of this work are openly available in the "Artifact for TaSSAT: A Stochastic Local Search Solver for SAT" at https://zenodo.org/records/10042124 [8]. The authors confirm that the data supporting the findings of this study are available within the article and the artifact.

References

1. SAT Competition 2022. https://satcompetition.github.io/2022/downloads.html, 2022.
2. Cesare Tinelli Aaron Stump, Geoff Sutcliffe. StarExec. https://www.starexec.org/starexec/public/about.jsp, 2013.
3. Tanbir Ahmed, Oliver Kullmann, and Hunter S. Snevily. On the van der Waerden numbers w(2; 3, t). *Discrete Applied Mathematics*, 174:27–51, 2014.
4. Adrian Balint. *Engineering stochastic local search for the satisfiability problem.* PhD thesis, University of Ulm, 2014.
5. 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 *Proceedings of SAT-2014*, pages 302–316, 2014.
6. Armin Biere. YalSAT: Yet Another Local Search Solver. http://fmv.jku.at/yalsat/, 2010.
7. Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximilian Heisinger. CADICAL, KISSAT, PARACOOBA, PLINGELING and TREENGELING entering the SAT Competition. In *Proceedings of SAT Competition*, pages 50–53, 2020.
8. Md Solimul Chowdhury, Cayden Codel, and Marijn Heule. Artifact for tassat: A stochastic local search solver for sat.
9. Md Solimul Chowdhury, Cayden R. Codel, and Marijn J.H. Heule. A linear weight transfer rule for local search. In *NASA Formal Methods*, 2023.
10. Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158, 1971.
11. Marijn J. H. Heule. Solving edge-matching problems with satisfiability solvers. In *SAT 2009 competitive events booklet*, pages 69–82, 2009.
12. Marijn J. H. Heule, Anthony Karahalios, and Willem-Jan van Hoeve. From cliques to colorings and back again. In *Proceedings of CP-2022*, pages 26:1–26:10, 2022.
13. Marijn J. H. Heule, Manuel Kauers, and Martina Seidl. New ways to multiply 3x3-matrices. *J. Symb. Comput.*, 104:899–916, 2019.
14. Marijn J. H. Heule and Oliver Kullmann. The science of brute force. *Commun. ACM*, 60(8):70–79, 2017.
15. Abdelraouf Ishtaiwi, John Thornton, Abdul Sattar, and Duc Nghia Pham. Neighbourhood clause weight redistribution in local search for SAT. In *Proceedings of CP-2005*, Lecture Notes in Computer Science, pages 772–776, 2005.
16. Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In Oliver Kullmann, editor, *Proceedings of SAT-2009*, pages 244–257, 2009.
17. David Tompkins. *Dynamic Local Search for SAT: Design, Insights and Analysis.* PhD thesis, The University of British Columbia, 2010.