Structure-Aware SAT Heuristics for Hales–Jewett Instances

Nayda Farnsworth
nfarnsworth@colgate.edu
Colgate University

Abstract

In this work, we study Hales–Jewett SAT instances and design heuristics tailored to their structure. We observe that disabling activity updates in VSIDS significantly improves CDCL solver performance, suggesting that conflict-driven branching induces an inefficient ordering of points when coloring the hypercube. Motivated by this, we develop an Incidence-Based Decision Making (IBDM) heuristic that prioritizes variables based on the number of combinatorial lines passing through each point, favoring structurally central variables. We implement IBDM in CaDiCaL and evaluate it on Hales–Jewett instances, where it consistently reduces conflict counts and yields runtime improvements on larger instances. These results demonstrate that exploiting combinatorial structure can yield meaningful gains over general-purpose heuristics on domain-specific SAT instances.

1 Introduction

The Boolean Satisfiability (SAT) problem is a fundamental problem in theoretical computer science. As the first problem shown to be NP-complete [5], it plays a central role in complexity theory: every problem in NP can be reduced to it.

Despite the absence of known polynomial-time algorithms for SAT, modern SAT solvers routinely solve Boolean formulas with millions of variables and clauses. This practical efficiency has led to widespread applications in areas such as circuit verification [3, 4], cryptography [12, 13], and combinatorics [7, 11].

SAT solvers have become a powerful tool for studying problems in Ramsey theory, a branch of combinatorics studying the emergence of order within large, chaotic systems. By overcoming the limitations of traditional structural methods, SAT solvers have enabled the computation of exact values and improved bounds for Ramsey-type numbers, including Schur, van der Waerden, and Hales–Jewett numbers [7, 9, 10].

Boolean formulas arising from Ramsey-theoretic problems often exhibit strong, problem-specific regularity, allowing heuristic parameters to be fine-tuned for entire classes of instances. Such fine-tuning has been shown to significantly improve SAT solver performance on formulas arising from the van der Waerden problem [7]. We investigate whether similar improvements can be achieved for the Hales–Jewett problem.

Using the Kissat [2] SAT solver, we demonstrate that disabling variable activity updates in the VSIDS branching heuristic yields significant performance improvements on Hales–Jewett SAT instances. This suggests that conflict-driven branching induces a poor, structure-agnostic ordering of points when coloring the hypercube, leading to frequent conflicts and inefficient search.

Motivated by the limitations of conflict-driven branching, we develop a new heuristic, Incidence-Based Decision Making (IBDM). IBDM leverages the Hales–Jewett domain by ordering variables according to the number of lines passing through their associated points in the hypercube. This ordering ensures that points in highly constrained regions of the hypercube are colored first.

To our knowledge, this is among the first approaches to explicitly incorporate combinatorial structure from a Ramsey-type problem into a SAT heuristic. We implement IBDM by modifying the CaDiCaL [1] SAT solver and evaluate its performance on several Hales–Jewett instances against an unmodified CaDiCaL baseline. Our method reduces conflicts relative to the baseline across all tested instances.

Our results show that improved performance on Hales–Jewett SAT instances arises from both careful fine-tuning of existing heuristics and the design of new, structure-aware methods. More broadly, we outline a framework for accelerating SAT solving on domain-specific instances by incorporating structural information into heuristic design, demonstrating that such approaches can outperform well-established general-purpose methods.

1.1 Outline

The remainder of the paper is organized as follows. In Section 2, we introduce the necessary background on the Boolean Satisfiability (SAT) problem, Conflict-Driven Clause Learning (CDCL) and its branching heuristics, Hales–Jewett numbers, their SAT encodings, and related work. In Section 3, we demonstrate the effect of disabling activity updates in VSIDS and provide intuition for the observed behavior. In Section 4, we introduce Incidence-Based Decision Making (IBDM) and describe its design, implementation, and experimental results. In Section 5, we summarize our contributions, discuss limitations, and outline directions for future work.

2 Background

In this section, we review the necessary background on SAT solving, CDCL heuristics, and the Hales–Jewett problem, along with relevant prior work.

2.1 Boolean Satisfiability (SAT)

A Boolean formula over Boolean variables $b_1, \ldots, b_n$ is constructed using the logical connectives conjunction ($\wedge$), disjunction ($\vee$), and negation ($\neg$). An assignment $\alpha : \{b_1, \ldots, b_n\} \to \{0, 1\}$ maps each variable to a truth value, and a formula evaluates to either true or false under such an assignment.

Boolean formulas are often expressed in *conjunctive normal form* (CNF). A CNF formula is a conjunction of clauses, where each clause is a disjunction of literals, and each literal is either a variable or its negation. For example,

$$(b_1 \vee \neg b_2) \ \wedge \ (\neg b_3 \vee b_4 \vee \neg b_5)$$

is a CNF formula consisting of two clauses.

The Boolean Satisfiability (SAT) problem asks whether there exists an assignment of truth values to the variables of a given

Boolean formula such that the formula evaluates to true. A formula is said to be *satisfiable* if such an assignment exists, and *unsatisfiable* otherwise.

2.2 Conflict-Driven Clause Learning (CDCL)

The dominant paradigm for solving SAT is *Conflict-Driven Clause Learning* (CDCL) [3], which extends the classical DPLL [6] algorithm with clause learning and non-chronological backtracking. Algorithm 1 offers a high-level view of the CDCL procedure.

Algorithm 1 Conflict-Driven Clause Learning (CDCL)

```
M ← ∅, dl ← 0
while M does not assign all variables do
    UnitPropagate(M)
    if Conflict(M) then
        if dl = 0 then
            return UNSAT
        end if
        C ← AnalyzeConflict(M)
        Backtrack(M, C)
    else
        Decide(M)                          ▷ sets dl ← dl + 1
    end if
end while
return SAT
```

The solver maintains a partial assignment M together with a current decision level dl. As shown in Algorithm 1, CDCL alternates between propagation, conflict analysis, and branching decisions until it either finds a satisfying assignment or proves unsatisfiability.

The procedure UnitPropagate(M) extends the current partial assignment by repeatedly applying unit propagation. Whenever a clause has all but one literal falsified under M, the remaining literal must be assigned true. These forced assignments are called *implications*. Propagation continues until no further implications can be derived or until a clause becomes falsified.

If propagation falsifies a clause, then Conflict(M) holds. In that case, the solver invokes AnalyzeConflict(M) to derive a new clause, called a *learned clause*, that explains the conflict and prevents the same assignment pattern from recurring. The learned clause is then used by Backtrack(M, C) to perform *non-chronological backtracking*, returning directly to an earlier decision level determined by the conflict rather than undoing assignments one level at a time.

If no conflict occurs and M is still incomplete, the solver calls Decide(M), which selects an unassigned variable, assigns it a truth value, and thereby creates a new decision level. The process then repeats until all variables are assigned, in which case the solver returns SAT, or until a conflict occurs at decision level 0, in which case it returns UNSAT.

2.3 Branching Heuristics

While Algorithm 1 presents a high-level view of the CDCL procedure, its practical performance depends critically on the heuristics used to implement its subroutines. In particular, the Decide(M) step relies on branching heuristics to select the next decision variable and its assigned value.

2.3.1 VSIDS. The seminal variable selection heuristic is *Variable State Independent Decaying Sum* (VSIDS). VSIDS assigns each variable b_i an activity score $\mathrm{act}[b_i]$, which determines its priority in a decision queue. At each decision step, the solver selects the variable

$$b^* = \arg\max_{b_i \in U} (\mathrm{act}[b_i]),$$

where U denotes the set of unassigned variables.

The activity scores are initialized uniformly and are dynamically updated during the search. In particular, when a variable b_i appears in a learned clause, its activity is increased according to

$$\mathrm{act}[b_i] \leftarrow \mathrm{act}[b_i] + \Delta,$$

where $\Delta > 0$ is a fixed increment. To prevent unbounded growth and to emphasize recent conflicts, activity values are periodically decayed:

$$\mathrm{act}[b_i] \leftarrow \alpha \cdot \mathrm{act}[b_i], \quad \text{where } 0 < \alpha < 1.$$

2.3.2 Polarity. Once a variable is selected, a *polarity heuristic* is used to determine the truth value assigned to the chosen variable. Common strategies include fixed polarity (always assigning true or false), phase saving (reusing the variable's most recent assignment), and biased schemes that favor one value over the other.

While CDCL solvers employ a variety of heuristics, we focus on branching heuristics; see [3] for a comprehensive overview.

2.4 Hales–Jewett Numbers

The Hales–Jewett Theorem is a cornerstone of Ramsey theory. It asserts that for any $k, r \in \mathbb{Z}^+$, there exists an integer n such that for all $d \geq n$, every r-coloring of the discrete hypercube $[k]^d$ admits a monochromatic combinatorial line.

Here, a *combinatorial line* in $[k]^d$ is a set of k points obtained by selecting a nonempty subset of coordinates to vary over all values in $[k]$, while keeping the remaining coordinates fixed. For example, in $[3]^2$, the set

$$\{(1, 2), (2, 2), (3, 2)\}$$

forms a combinatorial line.

The minimal dimension n for which monochromatic combinatorial lines appear in every r-coloring of $[k]^n$ is called the *Hales–Jewett number* and is denoted by $HJ(k; r)$.

Computing Hales–Jewett numbers is notoriously difficult, as the associated search space grows exponentially. In this work, we focus on the cases

$$HJ(4; 2) \quad \text{and} \quad HJ(3; 3),$$

for which the best-known lower bounds are

$$HJ(4; 2) \geq 12 \quad \text{and} \quad HJ(3; 3) \geq 14.$$

Determining $HJ(k; r)$ requires establishing both lower and upper bounds, by constructing an r-coloring of $[k]^n$ that avoids monochromatic combinatorial lines or proving that no such coloring exists. In either case, this involves reasoning over the space of all possible colorings of $[k]^n$. For $HJ(4; 2)$, this entails examining

$$2^{4^{12}} = 2^{16,777,216}$$

colorings of $[4]^{12}$ and checking 227,363,409 lines per coloring. Similarly, for $HJ(3; 3)$, one must consider

$$3^{3^{14}} = 3^{4,782,969}$$

colorings of $[3]^{14}$ and check 263,652,487 lines at each step.

This combinatorial explosion motivates the use of SAT-based methods to efficiently search for monochromatic line-avoidant colorings, as brute-force approaches are computationally infeasible.

2.5 Hales–Jewett and SAT

We construct a Boolean formula $\varphi_{HJ(k;r)>n}$ that is satisfiable if and only if there exists an r-coloring of $[k]^n$ that avoids monochromatic combinatorial lines, i.e., if and only if $HJ(k;r) > n$.

2.5.1 Encoding $HJ(4;2)$. We begin by describing the encoding of $\varphi_{HJ(4;2)>n}$. For each point $x \in [4]^n$, we introduce a Boolean variable b_x, where $b_x = 1$ indicates that x is assigned the first color (e.g., red), and $b_x = 0$ indicates the second color (e.g., blue).

For each combinatorial line

$$L = \{w(1), w(2), w(3), w(4)\} \subseteq [4]^n,$$

we add the constraints

$$\neg\mathbf{b_L} := (\neg b_{w(1)} \vee \neg b_{w(2)} \vee \neg b_{w(3)} \vee \neg b_{w(4)})$$

and

$$\mathbf{b_L} := (b_{w(1)} \vee b_{w(2)} \vee b_{w(3)} \vee b_{w(4)})$$

to the formula $\varphi_{HJ(4;2)>n}$. The first clause prevents all points in L from being assigned the first color, while the second prevents all points from being assigned the second color. Together, these clauses ensure that L is not monochromatic.

Applying this construction to all combinatorial lines yields the formula

$$\varphi_{HJ(4;2)>n} = \bigwedge_{L\in\mathcal{L}} (\neg\mathbf{b_L} \wedge \mathbf{b_L}),$$

which is satisfiable if and only if $HJ(4;2) > n$.

2.5.2 Encoding $HJ(3;3)$. Next, we outline the process by which we encode $\varphi_{HJ(3;3)>n}$. For each point $x \in [3]^n$, we introduce three Boolean variables

$$(b_{x,1}, b_{x,2}, b_{x,3}),$$

where $b_{x,i} = 1$ indicates that point x is assigned color i.

For each $x \in [3]^n$, we introduce the exactly-one constraint

$$\mathbf{x_{or}} := \left\{(b_{x,1} \vee b_{x,2} \vee b_{x,3}) \ \wedge \ \left((\neg b_{x,i} \vee \neg b_{x,j}) \text{ for all } i < j\right) \right\},$$

which ensures that exactly one color is assigned to x.

Then, for each combinatorial line $L = \{w(1), w(2), w(3)\} \subseteq [3]^n$, we introduce the constraints

$$\bigwedge_{i\in\{1,2,3\}} \neg\mathbf{b_{L,i}},$$

where

$$\neg\mathbf{b_{L,i}} := (\neg b_{w(1),i} \vee \neg b_{w(2),i} \vee \neg b_{w(3),i}).$$

Each clause $\neg\mathbf{b_{L,i}}$ prevents all points in L from being assigned color i, and together these clauses ensure that L is not monochromatic.

Applying this construction to all points and combinatorial lines yields

$$\varphi_{HJ(3;3)>n} = \left(\bigwedge_{x\in[3]^n} \mathbf{x_{or}}\right) \wedge \left(\bigwedge_{L\in\mathcal{L}} \left(\bigwedge_{i\in\{1,2,3\}} \neg\mathbf{b_{L,i}}\right)\right),$$

which is satisfiable if and only if $HJ(3;3) > n$.

For a more detailed discussion of SAT encodings for Hales–Jewett problems, we refer the reader to [9].

2.6 Related Work

SAT-based methods have been widely applied to Ramsey-type problems, improving our understanding of classical parameters such as Schur numbers [10], van der Waerden numbers [7], and Hales–Jewett numbers [9]. However, the extent to which domain-specific structure can improve SAT solver heuristics remains underexplored, especially in the context of the Hales–Jewett problem.

Heuristic fine-tuning has proven effective for SAT instances arising from combinatorics, as demonstrated by *NegVanSAT* [7], a solver tailored to van der Waerden formulas. By modifying MiniSat to use fixed negative polarity rather than the standard fixed positive polarity, Abd El-Maksoud and Abdalla solved these instances at record speed. Their approach yielded new values for van der Waerden numbers that were previously computationally infeasible.

We extend their approach in two key ways. First, we focus on Hales–Jewett instances, where heuristic fine-tuning remains largely unexplored. Second, we move beyond parameter tuning by designing a new heuristic that incorporates combinatorial structure from the Hales–Jewett domain into solver branching decisions.

3 Bias in Branching

In this section, we examine how branching heuristics affect performance on Hales–Jewett SAT instances. All experiments were conducted on the Colgate Supercomputer using Kissat, a state-of-the-art CDCL SAT solver based on CaDiCaL. We report single-run results, as the solver is deterministic under fixed settings, and impose a time limit of 700 hours.

We find that polarity-based modifications, similar to those used in NegVanSAT, do not yield consistent improvements on Hales–Jewett instances. In contrast, modifications to VSIDS produce substantial and reliable performance gains, and form the primary focus of our analysis.

3.1 Disabled Activity Bumping in VSIDS

Our primary improvement arises from disabling VSIDS activity score updates. Specifically, we initialize all activity scores uniformly,

$$\text{act}[b_i] = 1,$$

and enable `--no-bump` to keep scores fixed during the search. Variable selection then follows a static ordering determined at initialization, eliminating the adaptive, conflict-driven behavior of VSIDS.

3.2 Results

Despite the central role of VSIDS in modern CDCL solvers, disabling its activity updates leads to substantial performance improvements on Hales–Jewett SAT instances. Tables 1 and 2 show that this modification significantly reduces both solving time and conflict counts.

Table 1: Kissat Runtime (s) and Conflicts on $\varphi_{HJ(3;3)>n}$

	Baseline		No-Bump	
n	Time (s)	Conflicts	Time (s)	Conflicts
7	0.09	76	0.10	97
8	51.2	810K	0.45	258
9	364	3.55M	2.61	693
10	91.7K	568M	653	1.31M

Table 2: Kissat Runtime (s) and Conflicts on $\varphi_{HJ(4;2)>n}$

	Baseline		No-Bump	
n	Time (s)	Conflicts	Time (s)	Conflicts
7	14.5	226K	1.25	12.8K
8	78.9	721K	87.9	249K
9	1.95K	7.61M	884	1.97M

For small instances ($n \leq 6$), both configurations perform similarly, as these cases are easily solved. For larger instances, however, the effect becomes pronounced. For $\varphi_{HJ(3;3)>10}$, the baseline solver requires over 25 hours, whereas the `-no-bump` configuration completes in under 11 minutes, with over two orders of magnitude fewer conflicts. For $\varphi_{HJ(4;2)>n}$, the impact is more moderate but still significant. While runtime improvements are less consistent (for example, at $n = 8$ the `-no-bump` configuration is slightly slower), conflict counts are consistently reduced, often by substantial margins, indicating a more efficient search process.

Overall, these results show that conflict-driven branching is not well-suited to Hales–Jewett SAT instances. Although such heuristics prioritize recently relevant Boolean variables, they appear to produce a near-random ordering of points in the hypercube. The constraints induced by combinatorial lines are highly symmetric and distributed, so conflicts do not concentrate on a small subset of variables. As a result, VSIDS activity updates reflect diffuse and rapidly changing conflict information rather than identifying persistently important variables, leading to unstable and effectively uninformative variable orderings that degrade search efficiency.

In contrast, a fixed ordering enforces a more structured progression through the hypercube, reducing conflicts and improving efficiency, thereby motivating the development of structure-aware branching heuristics.

4 Structure-Aware Heuristic Design

In this section, we introduce a new branching heuristic, *Incidence-Based Decision Making* (IBDM), tailored to Hales–Jewett instances, describe its implementation, and evaluate its performance against a CaDiCaL baseline.

4.1 Incidence-Based Strategy

We begin by grounding the heuristic in the combinatorial structure of the Hales–Jewett problem. Points in $[k]^n$ lie on varying numbers of combinatorial lines, depending on how many coordinates share the same value.

Let $x = (x_1, \ldots, x_n) \in [k]^n$. For each $a \in [k]$, let m_a denote the number of coordinates of x equal to a. The number of combinatorial lines passing through x is then

$$\mathrm{inc}(x) := \sum_{a=1}^{k} (2^{m_a} - 1).$$

This follows since each combinatorial line through x arises by selecting a nonempty subset of coordinates equal to some a and allowing them to vary over $[k]$, giving $2^{m_a} - 1$ possibilities for each $a \in [k]$. We refer to this quantity as the *incidence* of x.

Intuitively, assigning a color to a point that lies on many combinatorial lines imposes stronger global constraints than assigning a color to a lower-incidence point. We therefore prioritize variables corresponding to high-incidence points.

Let b_x denote the Boolean variable corresponding to a point $x \in [k]^n$. In our approach, we set

$$\mathrm{act}[b_x] = \mathrm{inc}(x).$$

At initialization, these scores are fixed, and activity updates are disabled. As a result, variable selection is determined entirely by the incidences of points in the hypercube.

4.2 IBDM Implementation

We considered several solver frameworks for implementing IBDM. MiniSat lacks sufficient performance on Hales–Jewett SAT instances, while Kissat's tightly optimized codebase makes targeted modifications difficult. We therefore use CaDiCaL, which provides a balance of competitive performance and sufficient flexibility for controlled heuristic modifications.

To solve $\varphi_{HJ(k;r)>n}$, we compute $\mathrm{inc}(x)$ for all $x \in [k]^n$ using a Julia script [8]. These scores are then supplied to CaDiCaL to initialize variable activities. We add a file loader in `src/score.cpp` to read externally generated scores, introduce the necessary declarations in `src/internal.hpp`, and update the `init_scores` routine to assign

$$\mathrm{act}[b_x] = \mathrm{inc}(x)$$

for each variable b_x. The solver is executed with activity updates disabled (CaDiCaL flags `-bump=0`, `-stabilizeonly=1`), ensuring that branching decisions are determined entirely by the incidence-based initialization.

4.3 IBDM Results

We evaluate IBDM on the Boolean formulas $\varphi_{HJ(3;3)>n}$ and $\varphi_{HJ(4;2)>n}$ for $7 \leq n \leq 9$. We use the number of conflicts as a proxy for search efficiency, isolating the effectiveness of the branching heuristic from implementation-specific overhead. Because CaDiCaL is less optimized than Kissat, the range of solvable instances is limited. Tables 3 and 4 compare IBDM against a baseline CaDiCaL configuration with uniform activity initialization and updates disabled.

Table 3: CaDiCaL Runtime (s) and Conflicts on $\varphi_{HJ(3;3)>n}$

	Baseline		IBDM	
n	Time (s)	Conflicts	Time (s)	Conflicts
7	0.19	134	0.18	77
8	0.77	386	0.76	307
9	4.54	3.29K	4.55	1.28K

Table 4: CaDiCaL Runtime (s) and Conflicts on $\varphi_{HJ(4;2)>n}$

	Baseline		IBDM	
n	Time (s)	Conflicts	Time (s)	Conflicts
7	150	666K	5.57	21.1K
8	8.53K	28.2M	2.58K	3.32M
9	Unknown	Unknown	Unknown	Unknown

For $\varphi_{HJ(3;3)>n}$, IBDM reduces the number of conflicts across all instances, with the difference increasing as n grows. Although this does not translate into runtime improvements, the reduction in conflicts indicates a more efficient search process. We attribute the lack of runtime gains to the overhead of computing and loading incidence scores relative to the short solve times.

The performance improvements of IBDM on $\varphi_{HJ(4;2)>n}$ are substantial, yielding reductions in both runtime and conflicts. In particular, we observe an order-of-magnitude speedup at $n = 7$ and similarly large gains at $n = 8$. Due to limitations of our CaDiCaL-based implementation, instances with $n = 9$ were not tractable.

5 Discussion, Limitations, and Future Work

In this section, we summarize our findings, outline limitations of our approach, and suggest directions for future work.

5.1 Discussion

We first identify a limitation of the seminal conflict-driven branching heuristic VSIDS. We show that VSIDS is poorly suited to Hales–Jewett SAT instances and that disabling activity updates yields substantial performance gains. This indicates that fixed variable orderings are more effective for these instances.

We then design a structure-aware branching heuristic, Incidence-Based Decision Making (IBDM). Building on the above observation, IBDM orders variables for branching using a combinatorial notion of incidence. In this approach, variables corresponding to high-incidence points in $[k]^n$ are prioritized in the ordering, effectively assigning colors to the most structurally central points first.

We implement IBDM in CaDiCaL and evaluate it against a baseline configuration with activity updates disabled. IBDM consistently reduces conflict counts on Hales–Jewett instances and, in some cases, yields substantial runtime improvements. These gains become more pronounced as n increases, suggesting that the benefits of IBDM grow with instance size.

More broadly, these findings demonstrate that incorporating domain-specific knowledge into SAT heuristics can significantly improve performance on structured instances. Unlike general-purpose heuristics such as VSIDS, which are agnostic to problem geometry, structure-aware approaches like IBDM exploit this information to guide the solver more effectively. Together, these results highlight structure-informed heuristic design as a promising direction for SAT-based approaches to combinatorial problems.

5.2 Limitations

Our evaluation is constrained by the computational difficulty of the Hales–Jewett problem, limiting experiments to a small range of instances. While the observed trends are consistent across all tested cases, extending the analysis to larger instances would provide stronger evidence of scalability, particularly for IBDM.

In addition, our implementation of IBDM is based on CaDiCaL, and it remains unclear how readily these modifications can be incorporated into highly optimized solvers such as Kissat without affecting overall performance. Such integration is likely necessary to fully realize the potential of our approach for improving Hales–Jewett lower bounds in practice.

5.3 Future Work

This work opens several directions for future research, centered on improving the practical performance of IBDM and extending the role of structure-aware heuristics in SAT solving.

A natural next step is to integrate IBDM into Kissat. While our CaDiCaL-based implementation demonstrates strong performance gains, implementing IBDM in a highly optimized solver is likely necessary to translate these improvements into stronger lower bounds for Hales–Jewett numbers.

Second, many opportunities remain to incorporate Hales–Jewett structure into SAT solving. Algorithms based on alternative measures, such as symmetry classes induced by coordinate permutations, as well as extensions to other SAT heuristics, may yield further improvements.

Third, the approach developed in this work may extend beyond the Hales–Jewett setting. Investigating structure-aware heuristic design for other classes of Boolean formulas—such as those arising from graph theory or network analysis, where notions of combinatorial incidence naturally arise—is a promising direction.

Finally, hardware-accelerated approaches, particularly FPGA-based SAT solving combined with structure-aware heuristics, may offer a path toward scaling these methods to substantially larger instances.

6 Acknowledgements

This work was completed under the supervision of Professor David Perkins, whose guidance was invaluable throughout the project. This work was supported in part by the Colgate University Research Computing and Data Services and the Colgate Supercomputer (partially supported by NSF grant OAC-2346664).

References

[1] Armin Biere. Cadical simplified satisfiability solver. https://github.com/arminbiere/cadical, 2018.

[2] Armin Biere, Mathias Fleury, and Maximilian Heisinger. Cadical, kissat, paracooba, plingeling and treengeling entering the sat competition 2020. In Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proceedings of SAT Competition 2020 – Solver and Benchmark Descriptions*,

pages 50–53. Department of Computer Science, University of Helsinki, 2020. URL: http://fmv.jku.at/kissat/.

[3] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*. IOS Press, 2009.

[4] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. Bounded model checking using satisfiability solving. *Formal Methods in System Design*, 19(1):7–34, 2001.

[5] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158. ACM, 1971. doi:10.1145/800157.805047.

[6] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962. doi:10.1145/368273.368557.

[7] Munira A. Abd El-Maksoud. A novel sat solver for the van der waerden numbers. *Journal of the Egyptian Mathematical Society*, 27(1):1–10, 2019. doi:10.1186/s42787-019-0021-1.

[8] Nayda Farnsworth. CDCL-based Solver. URL: https://github.com/njfarnsworth/CDCL-Experimentation.

[9] Nayda Farnsworth. A computational approach to improving bounds on the hales–jewett numbers. https://njfarnsworth.github.io/Farnsworth%20Math%20Thesis.pdf, 2025. Undergraduate thesis, Colgate University.

[10] Marijn J. H. Heule. Schur number five. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1), 2018.

[11] Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 9710 of *Lecture Notes in Computer Science*, pages 228–245. Springer, 2016.

[12] Frédéric Lafitte et al. Cryptosat: A tool for sat-based cryptanalysis. In *Proceedings of the 2018 IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE, 2018.

[13] Alexander Semenov, Alexander Zaikin, Igor Otpuschennikov, and Ilia Gribanov. Translation of algorithmic descriptions of discrete functions to sat with applications to cryptanalysis problems. *Logical Methods in Computer Science*, 16(1), 2020.