Smart Cubing for Graph Search: A Comparative Study

Markus Kirchweger
Algorithms and Complexity Group, TU Wien, Austria

Tomáš Peitl
Algorithms and Complexity Group, TU Wien, Austria

Stefan Szeider
Algorithms and Complexity Group, TU Wien, Austria

Hai Xia[1]
Algorithms and Complexity Group, TU Wien, Austria

Abstract

Parallel solving via cube-and-conquer is a key method for solving hard instances with SAT. While cube-and-conquer has proven successful for pure SAT problems, notably the Pythagorean triples conjecture, its application to SAT solvers augmented with propagators presents unique challenges as propagators learn constraints dynamically during the search.

We study this problem using SAT Modulo Symmetries (SMS) as our primary test case. In our setting, the SMS symmetry-breaking propagator is an ordinary IPASIR-UP propagator; the techniques below do not rely on properties specific to symmetry breaking, except in the benchmark instantiations. Through extensive experimentation comprising over 20,000 CPU hours, we systematically evaluate different cube-and-conquer variants on three well-studied combinatorial problems. Our methodology combines prerun phases to collect learned constraints, various cubing strategies, and parameter tuning via algorithm configuration.

The comprehensive empirical evaluation provides new insights into effective cubing strategies for propagator-based SAT solving. Our best method reduces total solving time by factors of 2-10x from improved cubing, and reduces the time for the hardest cubes by factors of 2–50x.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Mathematics of computing → Graph enumeration; Mathematics of computing → Extremal graph theory

Keywords and phrases cube and conquer, graph search, algorithm configuration, SAT solving

Digital Object Identifier 10.4230/LIPIcs.CP.2026.33

Related Version *Preliminary Version*: `https://arxiv.org/abs/2501.17201v1`

Supplementary Material *Software*: `https://doi.org/10.5281/zenodo.14746542`

Funding The authors acknowledge the support by the European Union's Horizon 2020 research and innovation programme under the Maria Skłodowska-Curie grant agreement No. 101034440, and the support from the Austrian Science Fund (FWF), project 10.55776/P36688, 10.55776/P36420 and 10.55776/COE12.

1 Introduction

Conflict-driven clause learning (CDCL) SAT solvers can solve instances with millions of variables and clauses [13]. However, for hard combinatorial problems, parallelization becomes essential. The *cube-and-conquer* technique is highly effective for such problems, famously solving the Pythagorean triples conjecture [21].

[1] Corresponding author

In cube-and-conquer, a look-ahead solver first partitions the search space into disjoint subproblems via cubes (partial assignments), which are then solved independently by CDCL solvers, enabling efficient parallelization.

Symmetries are a central challenge in constraint solving, and constraint programming has developed a broad range of techniques for detecting, exploiting, and breaking them [14]. Combinatorial SAT encodings, especially of graphs, often have highly symmetric search spaces. Many isomorphic graphs satisfy the same constraints, but only one *canonical* representative per class needs to be checked. CDCL solvers alone cannot exploit such symmetries, and static symmetry-breaking methods are limited [5].

SAT Modulo Symmetries (SMS) [28, 29] overcomes this by dynamically learning symmetry-breaking constraints via a custom propagator during search. SMS has been successfully applied to challenging problems in extremal combinatorics about graphs, hypergraphs, matroids, and clause sets [11, 22, 24, 25, 26, 27, 44, 45].

While standard cube-and-conquer can be applied to any SAT system with propagators, it often performs poorly. The key issue is that propagators dynamically introduce essential constraints that are not present in the initial formula. For SMS, cubing based only on the original formula partitions the space of all graphs, rather than canonical ones. This is one instance of a more general issue for propagator-based systems: the cuber may not see constraints that are generated only during search.

This paper presents a systematic study of cube-and-conquer tailored to propagator-augmented SAT solving, using SMS as a concrete and practically relevant instantiation. Our pipeline includes a *prerun* to extract learned and propagator-generated clauses, *look-ahead-based cubing* tested with multiple heuristics, and a final *solving* phase. Since cubes from the same instance often share structure, we use structured *algorithm configuration* [32] to optimize the solver for the resulting subproblems. We also explore new look-ahead scoring functions to guide cubing. Our experiments total over 20,000 CPU hours across several benchmarks.

Our *results* evaluate several components of the pipeline rather than isolating a single dominant factor. The main test-instance results, summarized in Table 5 and Figure 3, show that the full pipeline substantially improves over the previous CDCL-based cubing baseline. The gains come from combining prerun-informed cubing, March_cu-based look-ahead cubing, and configuration of the conquer solver.

The rest of the paper is organized as follows. Section 2 introduces background and notation. Sections 3 and 4 describe SMS and cube-and-conquer. In Section 5, we detail our adapted cubing pipeline. Section 6 introduces the benchmark problems. Section 7 presents experimental results. We conclude in Section 8.

2 Preliminaries

For a positive integer n, we write $[n] = \{1, 2, \ldots, n\}$. Below we review basics from propositional logic and graph theory.

2.1 SAT

A *literal* is a propositional variable x or its negation $\overline{x}$. A *clause* (resp., *cube*) is a disjunction (conjunction) of literals; a *CNF* (*DNF*) formula is a conjunction (disjunction) of clauses (cubes). We treat clauses and cubes as sets of literals and formulas as sets of clauses or cubes. For a clause or cube C, we write $\overline{C} := \{\,\overline{x} \mid x \in C\,\}$. Let $\mathrm{var}(x) := \mathrm{var}(\overline{x}) := x$ denote the variable of a literal, $\mathrm{var}(C) := \{\,\mathrm{var}(x) \mid x \in C\,\}$, $\mathrm{var}(F) := \bigcup_{C \in F} \mathrm{var}(C)$. A

partial assignment τ is a map from a set $\mathrm{dom}(\tau)$ of variables to $\{0,1\}$; it is *total* on V if $\mathrm{dom}(\tau) = V$. It extends to literals with variables in $\mathrm{dom}(\tau)$ via $\tau(\overline{x}) := 1 - \tau(x)$. A total assignment satisfies a clause if it maps at least one literal to 1, and satisfies a formula if it satisfies all its clauses. A satisfying total assignment is also called a *model*; the set of models of F is denoted $\mathrm{mod}(F)$. A formula F is *satisfiable* if $\mathrm{mod}(F) \neq \emptyset$, *unsatisfiable* if $\mathrm{mod}(F) = \emptyset$, and a *tautology* if $\mathrm{mod}(F) = \{0,1\}^{\mathrm{var}(F)}$. F *entails* G, written $F \models G$, if $\mathrm{mod}(F) \subseteq \mathrm{mod}(G)$. Two formulas F, G are *equisatisfiable* if both are or neither is satisfiable; they are *logically equivalent* if $\mathrm{mod}(F) = \mathrm{mod}(G)$. An assignment τ can be identified with the set of literals it maps to 1, and conversely we use any consistent set of literals as a partial assignment. Given an assignment τ and a formula F, the simplified formula $F[\tau]$ is obtained by removing satisfied clauses and false literals. For singleton assignments we omit braces, writing for instance $F[x]$ instead of $F[\{x\}]$. A *unit clause* contains a single literal. If a unit clause $\{\ell\} \in F$, then ℓ must be true in all models, and we may simplify F to $F[\ell]$. Repeating this until no unit clause remains or a contradiction arises is called *unit propagation*.

2.2 Graphs

We consider simple, undirected, and unweighted graphs without parallel edges or self-loops. A *graph* G consists of a set $V(G)$ of vertices and a set $E(G) \subseteq \binom{V}{2}$ of edges; we denote the edge between vertices $u, v \in V(G)$ by uv or equivalently vu. We write $\mathcal{G}_n$ to denote the class of all graphs with $V(G) = [n]$. The *adjacency matrix* of a graph $G \in \mathcal{G}_n$, denoted by A_G, is the $n \times n$ matrix where the element at row v and column u, denoted by $A_G(v,u)$, is 1 if $vu \in E$ and 0 otherwise. For a permutation $\pi : [n] \to [n]$, $\pi(G)$ denotes the graph obtained from $G \in \mathcal{G}_n$ by the permutation π, where $V(\pi(G)) = V(G) = [n]$ and $E(\pi(G)) = \{\, \pi(u)\pi(v) \mid uv \in E(G) \,\}$. Two graphs $G_1, G_2 \in \mathcal{G}_n$ are *isomorphic* if there is a permutation $\pi : [n] \to [n]$ such that $\pi(G_1) = G_2$; in this case G_2 is an *isomorphic copy* of G_1.

3 SAT Modulo Symmetries (SMS)

Modern SAT solvers are primarily based on *conflict-driven clause learning (CDCL)* [13, 35], a search procedure for CNF formulas. In each iteration, the solver performs unit propagation until fixpoint, chooses a branching variable, and learns clauses from conflicts (when both x and $\neg x$ are implied). When a model is found, the solver returns it; if it learns the empty clause, the formula is unsatisfiable.

The *SAT Modulo Symmetries (SMS)* framework augments a CDCL solver[2] with a custom propagator that reasons about graph isomorphisms, allowing search *modulo symmetry* for graphs with n vertices satisfying a CNF-encoded constraint.

The SMS propagator checks whether the current partial assignment defines a *canonical graph*, i.e., the lexicographically minimal adjacency matrix (row-wise) in its isomorphism class. This *minimality check* runs after unit propagation and may introduce a conflict, leading to the learning of a *symmetry-breaking clause* that excludes non-canonical graphs.

The minimality check is one of several SMS *external propagators*, which substitute for constraints that are hard to encode in CNF. For example, non-3-colorability cannot be expressed with polynomial-size formulas unless NP = coNP. Similarly, deciding lexicographic minimality among isomorphic graphs is also coNP-complete [6].

[2] Our implementation uses CaDiCaL [1], a state-of-the-art CDCL SAT solver, via the IPASIR-UP interface [12].

A key property relevant for this paper is common to SAT solvers with external propagators: some clauses are learned dynamically during solving and are not present in the initial formula. In SMS these include symmetry-breaking clauses and, depending on the benchmark, clauses from additional domain-specific propagators. This makes existing SAT parallelization approaches like cube-and-conquer difficult to apply effectively, as they assume the entire formula is known up front. In Section 5, we show how to adapt cubing to handle these dynamically added clauses; first, we review cube-and-conquer.

4 Cube-and-Conquer

Assume that a hard CNF formula F is to be solved by a SAT solver, and let x be a variable of F. Consider the formulas $F[x]$ and $F[\overline{x}]$ where x is assigned true or false, respectively. Clearly, F is satisfiable if and only if at least one of $F[x]$ and $F[\overline{x}]$ is satisfiable. Therefore, instead of solving F, one can solve $F[x]$ and $F[\overline{x}]$, the point being that these *sub-problems* can be solved in parallel, and will hopefully be easier to solve than the original formula F. This reasoning applies equally for enumeration problems: in order to enumerate the models of F, one can separately enumerate the models of $F[x]$ and $F[\overline{x}]$, and take the union of the two.

This basic splitting idea can, of course, be applied recursively: by re-splitting the sub-problem $F[\overline{x}]$ we obtain the collection of sub-problems $F[x]$, $F[\overline{x}, y]$, and $F[\overline{x}, \overline{y}]$. In general, given an input formula F, and a DNF tautology $\Gamma = C_1 \vee \cdots \vee C_r$ over (some of) the variables of F, the set of models of F is obtained as $\mathrm{mod}(F) = \bigcup_{i=1}^{r} \mathrm{mod}\left(F[C_i]\right)$. The C_i are called *cubes*, and the method that solves F by designing a suitable cube set Γ to divide into sub-problems and solving each sub-problem separately (typically in parallel) is called *cube-and-conquer* [21, 20].

Note that Γ does not have to be a tautology. For satisfiability checking in ordinary SAT (no SMS or other special propagators), it is sufficient that $F \wedge \Gamma$ is equisatisfiable to F. In our context, we add symmetry-breaking clauses on the fly, and we enumerate models instead of just checking satisfiability. We thus must require preservation of all models, i.e., that $F \models \Gamma$, or, equivalently, that $F \wedge \neg\Gamma$ is unsatisfiable.

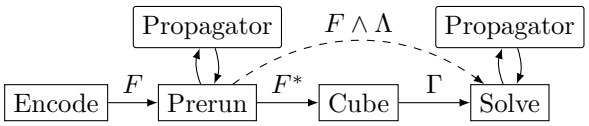
A fundamental optimization problem emerges: how to find a good splitter Γ in order to optimize solving (conquer) performance? The standard answer to this question is *look-ahead* solving. A look-ahead solver constructs a branching tree whose nodes are variables. At every node, the solver tries assigning each currently unassigned variable in both possible ways and performs the associated formula reduction (repeated removal of falsified literals and satisfied clauses followed by unit propagation). It then collects information about the reduced formula in each branch into a numerical value and combines both values into a variable score using a *scoring function*. The goal of the scoring function is to produce a single number on which all variables can be compared, which reflects both the total amount of reduction resulting from the two assignments as well as how balanced the reduction is between the two branches. The variable that scores highest on this aggregate score is picked as the next branching variable, and two subproblems are created on which the solver can be called recursively until a cutoff threshold is reached (typically specified as the number of variables to be assigned in total).

This basic method could be directly applied to a propagator-augmented SAT solver, but it has an important drawback. Not all information about the problem is encoded upfront in the formula F. Part of the encoding is "learned" dynamically by propagators; in SMS, this includes symmetry-breaking clauses as well as clauses added by other propagators. In Section 7, we confirm experimentally that this naive method of cubing, which disregards the

unknown parts of the encoding, indeed performs suboptimally. In Section 5, we propose an adaptation of cube-and-conquer that takes into account custom propagators and remedies this issue.

5 Cubing with SMS

The main contribution of this paper is the design and experimental evaluation of various cube-and-conquer pipelines for SAT solving with propagators, instantiated here with SMS and depicted in this diagram.



We start with an encoder that produces a propositional CNF F. In traditional cube-and-conquer, one would cube F directly, but in our scenario, this performs poorly (see Table 2). Instead, we propose a *prerun* phase, in which F is enriched by clauses from propagators to obtain a formula F^*.

During the prerun, we run CaDiCaL with the relevant attached propagators for a bounded amount of time. The point of this prerun phase is to improve the performance of the cubing phase by collecting three sets: a set Σ of symmetry-breaking clauses, which are one kind of propagator-generated clause in SMS, a set Π of clauses from other propagators, and a set Λ of important learned clauses from CDCL. The most important learned clauses for Λ are unit clauses, but we also store and reuse all learned clauses up to a fixed size. We may find models during prerun; we simply store them and block them in F^*. The enriched formula F^* is obtained as $F \wedge \Sigma \wedge \Pi \wedge \Lambda$.

After the prerun, we save the enriched CNF F^* and pass it on to the cuber. The cuber produces a set Γ of cubes to be solved by the final-phase solver.

The final solver is the same one as the one used for the prerun, up to the configuration of parameters.

Within this general framework, we evaluate a number of possible choices for each of the components and for the ways the components interact.

5.1 Prerun

In our experiments, the prerun uses CaDiCaL with the SMS minimality-check propagator, and with a suitable domain-specific propagator when required by the benchmark problem (see Section 6 for details on the benchmark problems and propagators). In principle, when the prerun phase is followed by a cubing phase that uses CaDiCaL, the solver could simply switch from prerun to cubing mode and continue to run, preserving all learned clauses and any other acquired state. Instead, we opted for a uniform approach over all cubers. After the defined prerun time is exceeded, the solver dumps Σ, Π, and Λ into a file and stops.

5.2 Cuber

For the cubing component, we evaluate several methods:

5.2.1 CDCL with a cutoff

In this method, whenever the number of assigned edge variables reaches a predefined threshold, we collect all assigned edge literals, output them as a cube, and add the negation as an irredundant learned clause. Thus, the generation of cubes blindly follows whatever path the CDCL solver takes. This method has been used successfully in previous applications of SMS [24]. In principle, we could run the propagator-augmented solver only once: after the amount of time given for prerun has elapsed, the solver simply switches to cubing mode and continues to run. In this case the resulting cubes depend strongly on the solver state at the end of prerun, meaning that different runs can result in very different cubes.

5.2.2 CaDiCaL with look-ahead

Here, we simulate a look-ahead solver within CaDiCaL (inside SMS). We use CaDiCaL's API methods `propagate` and `get_entrailed_literals` to track propagated literals. It is also crucial to turn on ILB (incremental lazy backtracking), which preserves the trail between calls, allowing reuse of previous computation.

At each decision point, we try to assign each candidate variable in both ways, counting the number of propagations a and b triggered in the two branches. If either assignment results in a conflict, we assign the variable to the opposite truth value and start the look-ahead again. Otherwise, we backtrack one decision level and test another variable. If all variables have been tested, we pick the variable for which the total number of propagations is as high and as balanced as possible. For this we use a *scoring function* σ, which balances these two aspects, and is a parameter of the method. The default scoring function is $\sigma(a, b) = \min(a, b)+\varepsilon(a+b)$, where $\varepsilon = 10^{-9}$ is a small positive number. The set of candidate variables can be either all unassigned variables or all unassigned edge variables. The rationale for the latter choice is that it is cheaper to look ahead on a smaller set of variables. Similar to counting the number of assigned edge variables, we stop further branching when the number of assigned variables plus the number of eliminated variables due to preprocessing exceeds a certain threshold.

5.2.3 March_cu

March_cu is a genuine look-ahead solver with a long history of evolution [17, 18, 19]. It works similarly to the previous method, except it takes into account more characteristics of the formula than just the number of propagations in each branch. Another difference is that the cubes it produces contain only branching literals. Therefore, the cubes do not contain information about all assigned variables including variables assigned by unit clauses. The scoring function used by March_cu is $\sigma(a, b) = a + b + ab$, where a and b are the numbers of propagations triggered in the two branches.

Preliminary results showed that March_cu performs best. Therefore, we additionally configure the scoring function σ within March_cu (see Section 7). To do this we prompt a large language model to offer suggestions for scoring functions, which we then evaluate. This way we take advantage of the creative potential of LLMs and avoid having to specify a parameterized search space of scoring functions. There is no risk coming from poor LLM output: we can manually check whether each suggestion makes at least superficial sense.

5.2.4 Assignment cutoff

All cubers need a threshold at which splitting stops and a cube is emitted (otherwise the cubing solver would continue until it solves the formula). This threshold is expressed in terms of assigned variables: as soon as a predetermined number of variables are assigned, the branch terminates and a cube is emitted. We refer to this threshold as the *assignment cutoff*.

We want to be able to control the final number of cubes, but this is difficult to predict because the branching tree of the cuber is not necessarily a balanced binary tree and because we also count assignments by propagation. We aim for around 5000 cubes: slightly more than the number of CPUs for load balancing, but not so many that overhead dominates. We handle this by making an informed guess for the assignment cutoff that gives us the desired number of cubes.

We follow a process similar to galloping search. We start with the blind estimate $\xi_1 = \frac{|\mathrm{var}(F^*)|}{12}$, producing γ_1 cubes. The base case $\xi_0 = 0$ yields $\gamma_0 = 1$ (empty) cube. Assuming exponential dependence of the number of cubes on the number of assigned variables $\gamma = cb^{\xi}$, for constants $c > 0$, $b > 1$, we update the estimate as:

$$\xi_{i+1} = \xi_i + (\xi_i - \xi_0) \cdot \frac{\ln(\gamma^*/\gamma_i)}{\ln(\gamma_i/\gamma_0)},$$

where $\gamma^* = 5000$ is the target. The rationale for this formula is that if the dependence is indeed $\gamma = cb^{\xi}$, then $\gamma_{i+1} = cb^{\xi_{i+1}} = \gamma^*$, i.e., the formula solves for the correct assignment cutoff in one step. If the true dependence is approximately $\gamma = cb^{\xi}$, then ξ_{i+1} should be a good estimate.

In each iteration we generate and count the number γ_i of cubes obtained under the cutoff ξ_i. If an estimate ξ_{i+1} exceeds a running upper bound ξ^*, we replace $\xi_{i+1} = \sqrt{\xi_i \xi^*}$. The upper bound ξ^* starts as the total number of variables and is reduced to $\xi^* := \xi_i - 1$ when $\gamma_i > 2\gamma^*$ cubes are produced. The process stops with ξ_i when $0.95\gamma^* \leq \gamma_i \leq 2\gamma^*$.

We perform this search whenever we generate a set of cubes anywhere in our pipeline. The number of iterations is typically very small, as the estimate ξ_i quickly converges on a value that produces an acceptable number of cubes. For each iteration, we start with the same enriched formula F^* rather than repeating the prerun each time.

5.2.5 Correctness of cubes

We independently verify the correctness of the generated cubes against the original formula F. Specifically, we ensure that no models are omitted. This validation step is crucial because the solvers we use are designed for satisfiability checking, and some modern SAT solver techniques are not necessarily model-preserving. We highlight that by verifying against the original formula F instead of the extended formula F^*, we ensure correctness of the whole process independently of any potential errors in the prerun phase.

Formally, let F be a propositional formula and Γ a set of cubes. Recall from Section 4 that the cubing is valid if $F \wedge \neg\Gamma$ is unsatisfiable. We verify this unsatisfiability using CaDiCaL. When the task is to enumerate all models, it is possible that models were encountered during the prerun. We can treat these models as cubes, i.e., for the verification, we additionally block all models found so far.

This step is computationally cheap, and in all cases it confirmed the validity of the cubes. In fact, even if the cubes were not valid, meaning there were models left under $F \wedge \neg\Gamma$, we would be able to enumerate and recover these lost models in the course of this validation check, so the step serves as both validation and error correction.

5.3 Conquering solver

For the final solver we take CaDiCaL again, but this time, we configure its parameters with the state-of-the-art automatic algorithm configuration tool SMAC [32], which has delivered performance boosts in both online [40] and offline [43] tuning for SAT-based combinatorial optimization problems. The final solver is thus the same one as used for prerun, up to possibly changed search strategies. For details of the automatic configuration process, see Section 7.

5.4 Related work

Wilson et al. [41] recently investigated cubing methods for distributed solving of Satisfiability Modulo Theories (SMT); they mostly focus on how to select good propositional atoms and other SMT-specific aspects. Wu et al. [42] proposed to use cubing as a way of facilitating the tuning of a solver; again, while they work with similar concepts, their objective is quite different. Our setup is unique in that we run the propagator for a bounded prerun in order to obtain clauses that better inform the cuber; this applies independently of whether the propagator performs symmetry breaking or domain-specific reasoning.

6 Benchmark Problems and Encoding

In this section, we introduce the graph search problems and present encodings on which we evaluate our methods. For most problems, we only sketch the encoding and refer to the original source, as our main focus is on comparing the solving approaches. See the supplementary material for generator scripts and details of the formulas.

Our benchmark set consists of three different problems. In the first two cases, the problems cannot be efficiently encoded into propositional logic, and external propagators (already implemented and available in the SMS library) are required. For these two problems, we describe what the propagator does. The last problem (Subsection 6.3) is encoded fully in propositional logic.

In addition to the individual encoding (and propagators if necessary) for each problem, we also use incomplete static symmetry breaking constraints [5]. These constraints are compatible with SMS in the sense that they are satisfied by the lexicographically minimal graph, but they are incomplete because they are satisfied by non-minimal graphs as well.

6.1 Coloring Triangle-Free Graphs (TF)

A *proper k-coloring* of a graph G is a mapping $c : V(G) \to \{1, \dots, k\}$ such that if $uv \in E(G)$, then $c(u) \neq c(v)$. The *chromatic number* of a graph G is the smallest integer k for which a proper k-coloring exists. If a graph contains a complete subgraph on k vertices, then its chromatic number must clearly be at least k. The opposite is not true: Mycielski constructed triangle-free graphs (without triangle subgraphs) with unbounded chromatic number [37]. Erdős asked about the values $f(k)$ [8], which denote the smallest number of vertices in a triangle-free non-$(k-1)$-colorable graph. Mycielski's construction provides upper bounds on $f(k)$, tight up to $k = 4$; for $k = 5$, minimal graphs are also known [15]. The cases $k \geq 6$ are open.

Task: Compute a triangle-free graph with n vertices and chromatic number at least k.

Encoding: We enumerate all triples of vertices to ensure triangle-freeness. Without loss of generality, we further restrict the search to *maximal triangle-free* graphs (triangle-free and adding any edge creates a triangle).

Propagator: When a model is found, the propagator checks whether it is k-colorable. If so, a *coloring clause* is learned, requiring future candidate graphs not to be colorable by this coloring [24].

6.2 Kochen-Specker Graphs (KS)

Kochen-Specker (KS) vector systems are special sets of vectors in at least 3-dimensional space that form the basis of the Bell-Kochen-Specker Theorem [2]. Kochen and Specker originally came up with a 3D KS vector system of size 117 [30]. The smallest known system (in 3D) has 31 vectors [38], while the best lower bound is 24 [24, 31], obtained with a computer search for *KS candidate graphs*: *non-010-colorable* graphs with additional restrictions. A graph is 010-colorable if its vertices can be colored red and blue such that no two adjacent vertices are both red and no triangle is all blue.

Task: Enumerate all KS candidates with n vertices.

Encoding: For the full list of constraints and the encoding, see [24].

Propagator: The propagator checks whether it is 010-colorable, and if so learns a *coloring clause*, which ensures that at least one edge or triangle must be present to invalidate this particular coloring.

6.3 Diameter-2-Critical Graphs (MS)

The *diameter* of a graph G is the largest distance between any pair of vertices, where distance is the length of a shortest path. A graph is *diameter-d-critical* if its diameter is d, and removing any edge increases it.

The study of these graphs was initiated by Erdős and Rényi [9] and remains an active area of research, especially for $d = 2$ [4, 16, 33]. The *Murty–Simon Conjecture* [3] posits that any diameter-2-critical graph with n vertices has at most $\lfloor n^2/4 \rfloor$ edges, with equality only for the complete bipartite graph $K_{\lceil n/2 \rceil, \lfloor n/2 \rfloor}$ (see also Mantel's Theorem [34] for triangle-free graphs). The enumeration of all diameter-2-critical graphs up to 10 vertices was done by [39] using Nauty [36]; later, 11 vertices were reached with a targeted search [7]; and finally, 13 vertices were reached using SMS [29].

We use the SAT encoding $D_2(n, m)$ from [29], which is satisfiable iff a diameter-2-critical graph with n vertices and m edges exists. It is known that the Murty–Simon bound holds for all $n \leq 24$ [10]. We set $m = \lfloor n^2/4 \rfloor$ to generate extremal graphs.

Task: Enumerate diameter-2-critical graphs with n vertices and $m = \lfloor n^2/4 \rfloor$ edges.

Encoding: We use variables $c_{i,j,k}$ to encode common neighbors: $c_{i,j,k} \leftrightarrow (e_{i,k} \wedge e_{j,k})$. This allows efficient SAT encoding of diameter-2-criticality; see [29] for details.

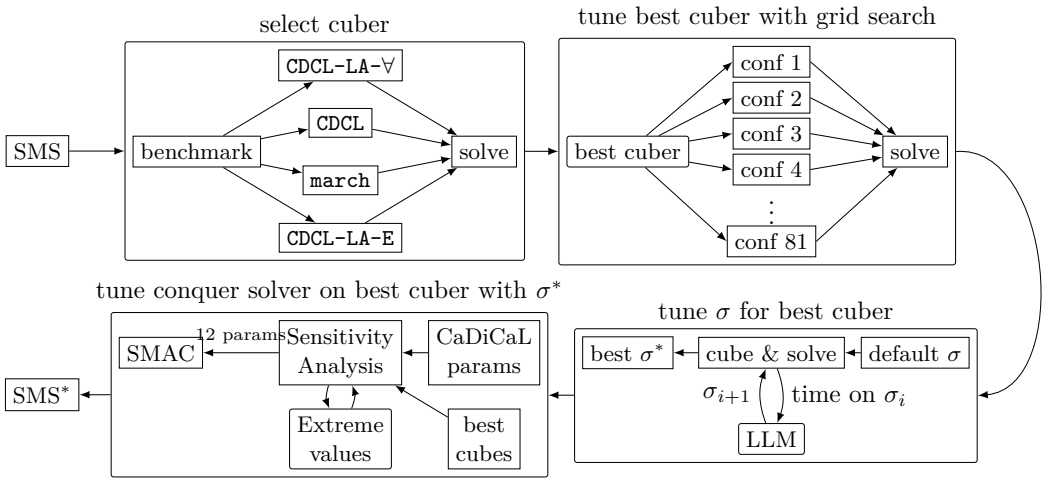


Figure 1 An overview of the whole configuration process. We start with unconfigured SMS. A "solve" node solves all cubes passed to it from an incoming arc. "cube & solve" generates fresh cubes for each suggestion of σ_i. A fixed set of cubes produced with the best cuber $+\sigma^*$ is used for tuning the conquer solver.

7 Results

In this section we present the technical details and results of our experiments. The goal is to minimize the total runtime of the entire pipeline, from encoding to obtaining solutions. However, given the complexity of the setup and the numerous parameters involved, we decided to simplify certain aspects and tune the individual stages separately.

For each benchmark problem, we create both a training and a test instance. The training instance is an easier variant of the problem with fewer vertices, solvable in approximately one CPU day. The test instance is a harder variant, with one additional vertex. While this might not sound like a big step, the search space generally grows exponentially in the number of edges, so roughly like $\exp(\Omega(n^2))$. For our selected combinatorial problems, increasing the number of vertices from n to $n+1$ typically makes the problem orders of magnitude harder. For Kochen-Specker and triangle-free graphs, we train on 21 and test on 22 vertices; for the Murty–Simon conjecture, we train on 16 and test on 17 vertices. Details on how to produce the encodings and cubes can be found in the supplementary material.

We tune the parameters solely on the training instances and only at the end assess their impact on the test instances. The training itself is split into three phases:

1. We evaluate all cuber versions with default parameters using the default SMS solver for the conquering phase.
2. Based on these results, we select the most promising cuber and tune its parameters, including the scoring function, using large language models (LLMs).
3. Lastly, we tune the parameters of the conquer solver using the cubes produced by the best cuber configuration.

We then evaluate the speedup achieved by the tuned version against the previous standard cubing approach used in the SMS context (CDCL with a cutoff). The whole process is illustrated in Figure 1.

7.1 Technical parameters

We use CaDiCaL v2.1.3 with minor adaptations that allow us to traverse learned clauses and check which variables remain after simplification (are "active"). We use a forked version of March_cu[3]. We run the experiments on a cluster with 16× Intel Xeon E5-2640 v4, 2.40GHz 10-core processors and 160GB of RAM, except for algorithm configuration of SMS using SMAC (see Section 7.4), which we ran on a cluster equipped with 3× AMD EPYC 7402, 2.80GHz 24-core processors and 1024GB of RAM, running Ubuntu 18.04.

7.2 Evaluation of Cubers

We evaluated all four cubing configurations mentioned in Section 5.2 (see Table 1).

Table 1 Summary of our cubing configurations. `LA` stands for look-ahead. We tested CaDiCaL in SMS with look-ahead on either all variables (`CDCL-LA-`$\forall$) or only the edge variables (`CDCL-LA-E`). The values "any" mean that any σ can be used by these cubers in principle.

cuber	`CDCL`	`CDCL-LA-`$\forall$	`CDCL-LA-E`	`march`
LA scope	NA	full	edge	full
σ	NA	any	any	any

We ran all training instances without a prerun and with a prerun time of 10 minutes using all four cubers, aiming for 5000 cubes (with the method described in Section 5.2.4). In this phase we use the default scoring function for each solver (if applicable).

Table 2 shows the solving times under various configurations and illustrates the impact of prerun. The clearest benefit of prerun appears in combination with `march`, suggesting a useful synergy between the clauses collected during prerun and the subsequent look-ahead cubing. For `march`, prerun reduces the time for the hardest cube on all three instances. The total solving time also improves for KS and MS, substantially so for MS, while it remains roughly comparable for TF. For the CDCL-based cubers the picture is more mixed. Since cubes are typically solved in parallel, the solving time for the hardest cube is an important additional indicator of quality (e.g., it lower bounds total wall-clock solving time).

The results indicate that `march` with prerun is the most suitable cubing setup for our problem set. `march` achieves a 6× speedup on the Murty–Simon training instance and a 1.5× speedup on the triangle-free training instance, compared to the default cuber, `CDCL`. Although `march` is not the top performer on Kochen-Specker, it remains competitive with the other cubers. Therefore, we proceed with tuning `march` in the following phases.

The advantage of `march` appears more pronounced for harder instances: those with higher overall solving times. A potential explanation is that a simple and cheap look-ahead metric counting only the number of propagated literals still performs fine for easier instances, but more complex look-ahead pays off for harder instances.

7.3 Tuning March_cu

March_cu is tuned by exhaustive grid search over $3^4 = 81$ combinations of four key parameters; the best-performing configuration is selected as March_cu*.

[3] We ran into memory problems when attempting to run the version of March_cu from `https://github.com/marijnheule/CnC`. We use the fork from `https://github.com/BrianLi009/MathCheck/tree/main/gen_cubes/march_cu`.

Table 2 Total solve time (sum) and time to solve the hardest cube (max) in CPU hours for cubes produced for training instances by various cubers (rows), with and without prerun, and solved by default SMS. t_{cube} gives the time for prerun (if applicable) and cubing, $\#_{cube}$ the number of cubes. `LA-E` and `LA-∀` abbreviate `CDCL-LA-E` and `CDCL-LA-∀`.

		KS			MS			TF		
prerun	cuber	sum (max)	t_{cube}	$\#_{cube}$	sum (max)	t_{cube}	$\#_{cube}$	sum (max)	t_{cube}	$\#_{cube}$
no	`CDCL`	42.8 (5.9)	0.00	1518	124.6 (3.4)	0.00	7385	42.5 (2.1)	0.00	8284
600s	`CDCL`	15.0 (0.2)	0.17	7191	213.8 (2.6)	0.17	5952	45.6 (0.5)	0.17	6526
no	`LA-E`	16.7 (0.2)	0.03	5430	64.2 (2.2)	0.13	7008	**27.5** (0.3)	0.05	9847
600s	`LA-E`	**13.4** (0.0)	0.18	5469	109.5 (0.9)	0.20	4886	36.2 (0.2)	0.27	5133
no	`LA-∀`	16.0 (0.3)	0.37	7036	69.6 (0.8)	3.40	5888	31.3 (**0.2**)	8.36	6009
600s	`LA-∀`	14.5 (0.0)	0.33	5867	65.4 (1.1)	7.09	5191	43.8 (0.3)	4.02	5240
no	`march`	29.0 (0.7)	0.06	9290	107.5 (4.6)	1.09	5085	30.3 (1.0)	1.05	5318
600s	`march`	15.4 (**0.0**)	0.43	8773	**33.9** (**0.7**)	1.22	5058	31.6 (0.2)	1.13	4870

We proceed with tuning March_cu, considering the following four parameters and testing three values for each (the default is given in boldface):

1. the weight of binary clauses: `bin` $\in \{15, \mathbf{25}, 35\}$;
2. the rate of exponential decay: `dec` $\in \{0.40, \mathbf{0.50}, 0.60\}$;
3. the minimum heuristic value: `min` $\in \{0, \mathbf{8}, 16\}$; and
4. the maximum heuristic value: `max` $\in \{\mathbf{550}, 1000, 1000000\}$.

Other parameters, such as single and double look-ahead iterations, are omitted, as increasing these values significantly extends cubing time. We try all 81 combinations of the above parameters: Table 3 provides for each training instance the best configuration and the running time of the default configuration as a reference. We observe that the best configuration is problem-dependent.

Having optimized `march_cu`'s parameters, we proceed to the scoring function.

Table 3 Best configurations for each training instance, with results (sum and max) for the best configuration and the running time of the default configuration (sum and max) as a reference.

	KS	MS	TF
cuber	sum (max)	sum (max)	sum (max)
`default`	15.39 (0.034)	35.24 (0.54)	29.53 (0.169)
`best`	13.51 (0.032)	34.34 (0.74)	27.19 (0.168)

Configuration of the scoring function σ

Starting from the default σ, an LLM proposes alternative scoring functions; these are evaluated by cubing and the measured runtimes are then used as feedback for another round of suggestions.

We used OpenAI GPT-o1 as a brainstorming aid for this step, but it turned out to have only limited impact on the final performance. The prompt described the benchmark problems, cube-and-conquer, the role of the scoring function, and the default function $\sigma(a, b) = a + b + ab$, where a and b are the numbers of propagated literals obtained by setting

the branching literal to true and false, respectively. After each round, we generated cubes with the suggested scoring functions, measured the total solving time with default SMS, and reported the best times back to the LLM before asking for new candidates. Overall, we tried 20 suggested scoring functions and selected, for each benchmark problem, the one with the shortest total solving time. The resulting improvements are modest compared with the main gains from prerun-based cubing and solver configuration. Table 4 provides the best scoring function and the speedup over the default scoring function for each benchmark problem.

Table 4 Scoring functions $\sigma(a, b)$ and speedup over the default scoring function for all training instances. a and b are the respective numbers of propagations of the left and right branches in a node of the search tree.

Problem	$\sigma(a, b)$	Speedup
KS	$\min(a, b) \; + \; \epsilon \cdot (a + b)$	3%
MS	$a + b \; + \; a * b \; + \; 0.5 \cdot \lvert a - b \rvert$	2.9%
TF	$a + b \; + \; a * b \; + \; 0.5 \cdot \lvert a - b \rvert$	7.9%

7.4 Tuning the Conquer Solver: SMS Configuration

A sensitivity analysis first identifies the 12 most impactful CaDiCaL and SMS parameters by toggling each to its extreme value; these are then optimized with SMAC to produce the tuned solver SMS*.

After selecting the most promising cubing setup, namely March_cu with prerun and the tuned scoring function, the final training step is to optimize the search strategies of the conquering solver. Thus, the following conquer-solver tuning is conditional on this fixed cuber choice; we do not tune a separate conquer solver for every cuber considered above. We first list all parameters of CaDiCaL and pick those that are relevant to solver performance. We run a sensitivity analysis for each parameter. This means that we take the default configuration and, for each parameter, toggle it to a different value. In the case of numerical parameters, we change the value to the high end of the admissible range. We evaluate each resulting configuration on the set of cubes produced by this selected March_cu-based cubing setup on each of the three training instances. With this, we identify a set of 10 parameters that seem promising for further automated tuning with SMAC: the parameters whose change yields the highest speedups. Along with the other two parameters for SMS itself ("`frequency`" and "`cutoff`"), the 12 parameters in total are listed in the supplementary material.

Figure 2 lists CaDiCaL's parameters ranked by impact on performance among sets of problems (cubes) with different default solving times. Each colored line indicates the ranking of one parameter on different sets of instances with given default solving time ranges. The three highest-ranked parameters rarely leave the top group. For example, the parameter "`BRD`" is not only the most influential parameter when solving the cubes with a default running time between 1 and 16 minutes but also the second most decisive parameter among the cubes with longer default solving times.

The parameters in Figure 2 can be divided into three groups. `SSO`, `STF`, and `SMI` control "stabilization phases:" these are phases in CDCL search when restarts are turned off. `BMA` and `BMI` give limits on clause sizes that enter blocked clause elimination [23], an important pre- and inprocessing technique. `BRD` and `FPH` control decision heuristics: `BRD` influences which variable scores are increased during clause learning, and `FPH` influences the phases of branching variables (whether they are set to true or false). From these observations it appears that restart-free "stable" phases are good for our benchmark problems.

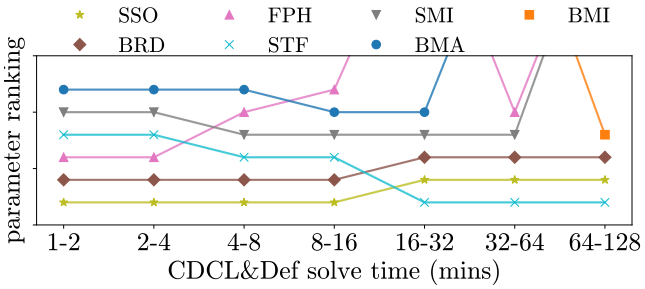


Figure 2 Parameter ranking for CaDiCaL when solving KS cubes, grouped by solving time under CDCL&Def. The parameters are ranked by impact, with the most impactful parameters at the bottom; lines leaving the plotting area indicate low impact. Parameter acronyms: stabilizeonly (SSO), bumpreasondepth (BRD), forcephase (FPH), stabilizefactor (STF), stabilizemaxint (SMI), blockminclslim (BMA), blockmaxclslim (BMI).

We run SMAC for each benchmark problem, tuning the parameters of SMS specifically for that problem and for the fixed set of cubes produced by the selected cuber. Training is restricted to cubes that the default solver solves within 5 minutes but not under 1 minute. The rationale behind this is to avoid training on trivial cubes, while also preventing excessive costs from training on extremely hard cubes.

Below, we refer to this optimized configuration as `Tun`, and to the default configuration as `Def`. The default remains the same for all benchmark problems, whereas "tuned" is specific to each problem and to the selected March_cu-based cubing setup. The first part of Table 5 summarizes the running times on the training instances.

7.5 Test performance

Lastly, we compare the performance of different pipelines on the test instances to assess the impact of tuning on smaller instances. This information is also summarized in Table 5, which shows the results for all cubers, both `Def` and `Tun`. The bottom part of Table 5 contains the main end-to-end comparison on test instances. It shows that the full pipeline, combining March_cu-based cubing and the tuned conquer solver, gives substantial speedups over the previous CDCL-based cubing baseline on all three benchmark problems.

Interestingly, while using March_cu as the cuber for the Kochen-Specker training instance resulted in no speedup, we observed a speedup factor of nearly 2 on the test instance. This outcome leaves us optimistic that, for even harder instances, we may benefit from this approach even more.

The results suggest that both the choice of cuber and the tuning of the conquer solver are crucial for optimizing pipeline efficiency. However, tuning March_cu resulted in only a modest improvement in running times.

In Figure 3 we show more detailed information about the distribution of running times over individual cubes. For each benchmark problem we show four plots: the rows differ in the cuber, and the columns differ in the conquering solver. We compare the default and best cuber, and also the default and best configuration of CaDiCaL. The plots show the total running time for cubes, grouped by individual running time. The total area of each plot corresponds to total solving time. As we move right within the group of four plots corresponding to one benchmark problem, the area shrinks and shifts to the left. Thus, the total solving time is reduced and concentrated more on easier cubes, making parallelization more effective.

Table 5 Total solving time (sum) and time for the hardest cube (max) in CPU hours for cubes produced for **training** and **test** instances by different cubers and conquer solvers (rows). `march*` refers to the tuned March_cu version. t_{cube} gives the time for prerun and cubing, $\#_{cube}$ the number of cubes.

	KS			MS			TF		
C&C	sum (max)	t_{cube}	$\#_{cube}$	sum (max)	t_{cube}	$\#_{cube}$	sum (max)	t_{cube}	$\#_{cube}$
Training									
`CDCL&Def`	15.0 (0.2)	0.17	7191	213.8 (2.6)	0.17	5952	45.6 (0.5)	0.17	6526
`march&Def`	15.4 (0.0)	0.43	8773	33.9 (0.7)	1.22	5058	31.6 (0.2)	1.13	4870
`march*&Def`	14.1 (0.0)	0.51	8893	35.2 (0.7)	0.96	5095	29.6 (0.2)	3.86	5941
`march*&Tun`	**11.8** (**0.0**)	0.51	8893	**29.9** (**0.4**)	0.96	5095	**19.9** (**0.1**)	3.86	5941
Test									
`CDCL&Def`	621.2 (58.0)	0.17	5521	5812.7 (643.6)	0.17	3861	783.0 (10.1)	0.17	8986
`march&Def`	371.4 (2.8)	1.93	6878	1095.7 (29.9)	3.14	6663	597.0 (10.8)	0.64	4880
`march*&Def`	353.6 (2.1)	1.41	7879	999.6 (25.3)	1.19	5469	625.9 (6.3)	1.20	4906
`march*&Tun`	**233.9** (**1.0**)	1.41	7879	**548.9** (**11.2**)	1.19	5469	**359.1** (**5.4**)	1.20	4906

Overall, the bottom part of Table 5 and Figure 3 provide the clearest summary of the final outcome. The full pipeline improves the total solving time on all test instances and also shifts the cube-runtime distribution toward easier cubes. Thus, while the individual ablation results above show a mixed picture across cubers and tuning choices, the combined configuration gives the strongest test-instance performance.

8 Conclusion

Our experimental evaluation considered several components of cube-and-conquer pipelines for propagator-augmented SAT solving: prerun-informed cubing, the choice of cuber, March_cu parameter choices, scoring functions, and configuration of the conquer solver. The clearest final takeaway is the end-to-end speedup on test instances shown in Table 5 and Figure 3: the full pipeline consistently improves over the previous CDCL-based cubing baseline. Among the individual components, March_cu with prerun was the most effective cubing setup in our experiments, while scoring-function tuning gave only modest additional gains.

Several promising directions emerge from these findings. While March_cu generates well-balanced subproblems, our analysis suggests potential for specialized splitting strategies that account for propagator behavior. Finally, our success with automatic configuration raises interesting questions about parameter space structure, as the preliminary analysis indicates cube clusters that might benefit from targeted approaches. Our methodology demonstrates how carefully handling dynamically learned propagator clauses can substantially improve parallel SAT-solving performance beyond the particular case of symmetry propagation.

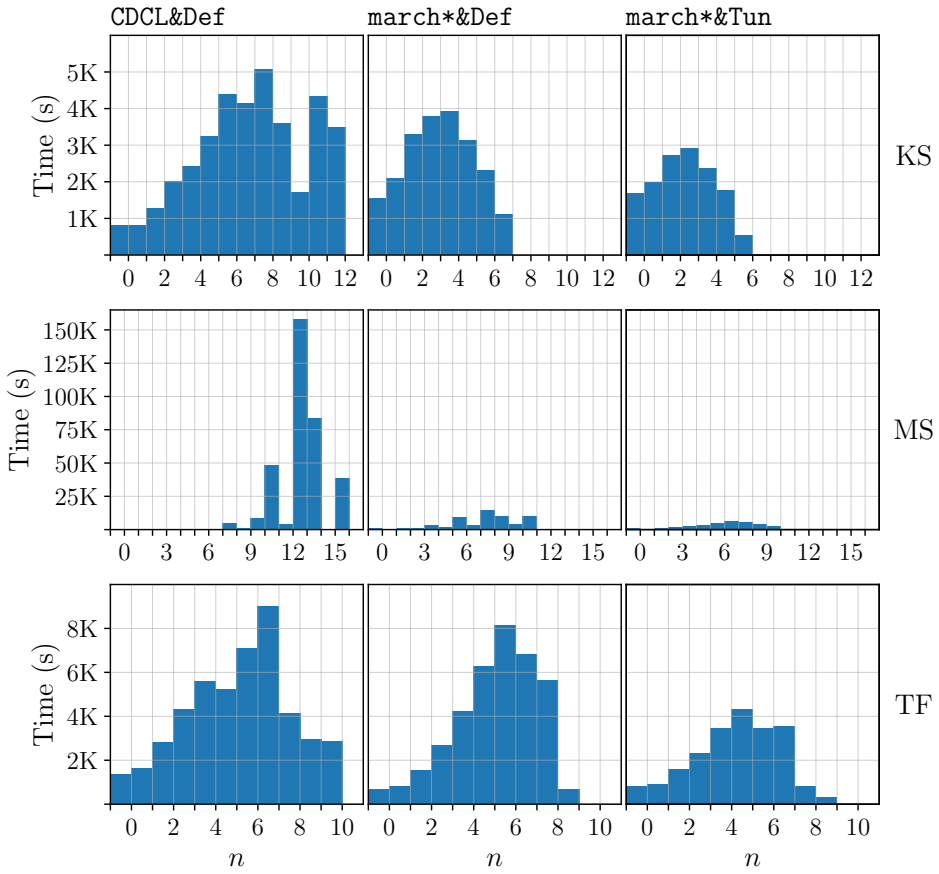


Figure 3 Total time to solve all cubes grouped by cube hardness. The height of a bar between x_0 and x_1 is the total amount of time needed to solve the cubes that took between 2^{x_0} and 2^{x_1} minutes. The total area of each chart is proportional to the total solving time reported in Table 5; all three charts for one problem use the same unit of area and are thus pairwise comparable.

References

1 Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL 2.0. In Arie Gurfinkel and Vijay Ganesh, editors, *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I*, volume 14681 of *Lecture Notes in Computer Science*, pages 133–152. Springer, 2024. doi:10.1007/978-3-031-65627-9_7.

2 Costantino Budroni, Adán Cabello, Otfried Gühne, Matthias Kleinmann, and Jan-Åke Larsson. Kochen-Specker contextuality. *Rev. Mod. Phys.*, 94:045007, 2022. doi:10.1103/RevModPhys.94.045007.

3 Louis Caccetta and Roland Häggkvist. On diameter critical graphs. *Discrete Math.*, 28(3):223–229, 1979. doi:10.1016/0012-365X(79)90129-8.

4 Ya-Chen Chen and Zoltán Füredi. Minimum vertex-diameter-2-critical graphs. *J. Graph Theory*, 50(4):293–315, 2005. doi:10.1002/jgt.20111.

5 Michael Codish, Alice Miller, Patrick Prosser, and Peter J. Stuckey. Constraints for symmetry breaking in graph representation. *Constraints*, 24(1):1–24, 2019. doi:10.1007/s10601-018-9294-5.

6 James M. Crawford, Matthew L. Ginsberg, Eugene M. Luks, and Amitabha Roy. Symmetry-breaking predicates for search problems. In Stuart C. Shapiro Luigia Carlucci Aiello, Jon Doyle, editor, *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR'96), Cambridge, Massachusetts, USA, November 5-8, 1996*, pages 148–159. Morgan Kaufmann, 1996.

7 Antoine Dailly, Florent Foucaud, and Adriana Hansberg. Strengthening the Murty-Simon conjecture on diameter 2 critical graphs. *Discrete Math.*, 342(11):3142–3159, 2019. doi:10.1016/j.disc.2019.06.023.

8 P. Erdős. Some remarks on chromatic graphs. *Colloq. Math.*, 16:253–256, 1967. doi:10.4064/cm-16-1-253-256.

9 P. Erdős and A. Rényi. On a problem in the theory of graphs. *A Magyar Tudományos Akadémia Matematikai Kutató Intézetének Közleményei*, 7(4):623–641 (1963), 1962.

10 Genghua Fan. On diameter 2-critical graphs. *Discret. Math.*, 67(3):235–240, 1987. doi:10.1016/0012-365X(87)90174-9.

11 Katalin Fazekas, Aina Niemetz, Mathias Preiner, Markus Kirchweger, Stefan Szeider, and Armin Biere. IPASIR-UP: User propagators for CDCL. In Meena Mahajan and Friedrich Slivovsky, editors, *The 26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023), July 04-08, 2023, Alghero, Italy*, LIPIcs. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.SAT.2023.8.

12 Katalin Fazekas, Aina Niemetz, Mathias Preiner, Markus Kirchweger, Stefan Szeider, and Armin Biere. Satisfiability modulo user propagators. *J. Artif. Intell. Res.*, 81:989–1017, 2024. doi:10.1613/JAIR.1.16163.

13 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, June 2023. doi:10.1145/3560469.

14 Ian P. Gent, Karen E. Petrie, and Jean-François Puget. Symmetry in constraint programming. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 329–376. Elsevier, 2006. doi:10.1016/S1574-6526(06)80014-3.

15 Jan Goedgebeur. On minimal triangle-free 6-chromatic graphs. *J. Graph Theory*, 93(1):34–48, 2020. doi:10.1002/jgt.22467.

16 Teresa W. Haynes, Michael A. Henning, Lucas C. van der Merwe, and Anders Yeo. Progress on the Murty-Simon Conjecture on diameter-2 critical graphs: a survey. *J. Comb. Optim.*, 30(3):579–595, 2015. doi:10.1007/s10878-013-9651-7.

17 Marijn Heule, Mark Dufour, Joris E. van Zwieten, and Hans van Maaren. March_eq: Implementing additional reasoning into an efficient look-ahead SAT solver. In Holger H. Hoos and David G. Mitchell, editors, *Theory and Applications of Satisfiability Testing, 7th International Conference, SAT 2004, Vancouver, BC, Canada, May 10-13, 2004, Revised Selected Papers*, volume 3542 of *Lecture Notes in Computer Science*, pages 345–359. Springer, 2004. doi:10.1007/11527695_26.

18 Marijn Heule and Hans van Maaren. March_dl: Adding adaptive heuristics and a new branching strategy. *J. Satisf. Boolean Model. Comput.*, 2(1-4):47–59, 2006. doi:10.3233/SAT190016.

19 Marijn Heule and Hans van Maaren. Look-ahead based SAT solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 155–184. IOS Press, 2009. doi:10.3233/978-1-58603-929-5-155.

20 Marijn J. H. Heule, Oliver Kullmann, and Armin Biere. Cube-and-conquer for satisfiability. In Youssef Hamadi and Lakhdar Sais, editors, *Handbook of Parallel Constraint Reasoning*, pages 31–59. Springer, 2018. doi:10.1007/978-3-319-63516-3_2.

21 Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th International Conference, Bordeaux, France, July 5-8, 2016, Proceedings*, volume 9710 of *Lecture Notes in Computer Science*, pages 228–245. Springer Verlag, 2016. doi:10.1007/978-3-319-40970-2_15.

22 Mikoláš Janota, Markus Kirchweger, Tomáš Peitl, and Stefan Szeider. Breaking symmetries in quantified graph search: A comparative study. In *AAAI 2025: The 39th Annual AAAI Conference on Artificial Intelligence*, 2025.

23 Matti Järvisalo, Armin Biere, and Marijn Heule. Blocked clause elimination. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6015 of *Lecture Notes in Computer Science*, pages 129–144. Springer, 2010. doi:10.1007/978-3-642-12002-2_10.

24 Markus Kirchweger, Tomás Peitl, and Stefan Szeider. Co-certificate learning with SAT modulo symmetries. In *Proceedings of the Thirty-Second International Joint Conference on Artificial Intelligence, IJCAI 2023, 19th-25th August 2023, Macao, SAR, China*, pages 1944–1953. ijcai.org, 2023. doi:10.24963/IJCAI.2023/216.

25 Markus Kirchweger, Tomás Peitl, and Stefan Szeider. A SAT solver's opinion on the Erdős-Faber-Lovász conjecture. In *26th International Conference on Theory and Applications of Satisfiability Testing, SAT 2023, July 4-8, 2023, Alghero, Italy*, volume 271 of *LIPIcs*, pages 13:1–13:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.SAT.2023.13.

26 Markus Kirchweger, Manfred Scheucher, and Stefan Szeider. A SAT attack on Rota's Basis Conjecture. In *Theory and Applications of Satisfiability Testing - SAT 2022 - 25th International Conference, Haifa, Israel, August 2-5, 2022, Proceedings*, 2022. doi:10.4230/LIPIcs.SAT.2022.4.

27 Markus Kirchweger, Manfred Scheucher, and Stefan Szeider. SAT-based generation of planar graphs. In Meena Mahajan and Friedrich Slivovsky, editors, *The 26th International Conference on Theory and Applications of Satisfiability Testing (SAT 2023), July 04-08, 2023, Alghero, Italy*, LIPIcs. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.SAT.2023.14.

28 Markus Kirchweger and Stefan Szeider. SAT modulo symmetries for graph generation. In *27th International Conference on Principles and Practice of Constraint Programming (CP 2021)*, LIPIcs, pages 39:1–39:17. Dagstuhl, 2021. doi:10.4230/LIPIcs.CP.2021.34.

29 Markus Kirchweger and Stefan Szeider. SAT modulo symmetries for graph generation and enumeration. *ACM Trans. Comput. Log.*, 25(3):1–30, 2024. doi:10.1145/3670405.

30 Simon Kochen and Ernst Specker. The problem of hidden variables in quantum mechanics. *J. Math. Mech.*, 17(1):59–87, 1967.

31 Zhengyu Li, Curtis Bright, and Vijay Ganesh. A SAT solver + computer algebra attack on the minimum Kochen–Specker problem. In Kate Larson, editor, *Proceedings of the Thirty-Third International Joint Conference on Artificial Intelligence, IJCAI-24*, pages 1898–1906. International Joint Conferences on Artificial Intelligence Organization, August 2024. Main Track. doi:10.24963/ijcai.2024/210.

32 Marius Lindauer, Katharina Eggensperger, Matthias Feurer, André Biedenkapp, Difan Deng, Carolin Benjamins, Tim Ruhkopf, René Sass, and Frank Hutter. Smac3: A versatile bayesian optimization package for hyperparameter optimization. *Journal of Machine Learning Research*, 23(54):1–9, 2022. URL: http://jmlr.org/papers/v23/21-0888.html.

33 Po-Shen Loh and Jie Ma. Diameter critical graphs. *J. Combin. Theory Ser. B*, 117:34–58, 2016. doi:10.1016/j.jctb.2015.11.004.

34 W. Mantel. Problem 28. *Wiskundige Opgaven*, 10:60–61, 1907.

35 João Marques-Silva, Inês Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability - Second Edition*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 133–182. IOS Press, 2021. doi:10.3233/FAIA200987.

36 Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, II. *J. Symbolic Comput.*, 60:94–112, 2014. doi:10.1016/j.jsc.2013.09.003.

37 Jan Mycielski. Sur le coloriage des graphs. *Colloquium Mathematicae*, 3(2):161–162, 1955.

38 A Peres. Two simple proofs of the Kochen-Specker theorem. *J. Phys. A Math. Theor.*, 24(4):L175, February 1991. doi:10.1088/0305-4470/24/4/003.

39 Jovan Radosavljević and Miodrag Živković. The list of diameter-2-critical graphs with at most 10 nodes. *IPSI Transactions on Advanced Research*, 16(1):1–5, January 2020. URL: http://ipsitransactions.org/journals/papers/tar/2020jan/p9.pdf.

40 Vaidyanathan Peruvemba Ramaswamy, Stefan Szeider, and Hai Xia. The power of collaboration: Learning large bayesian networks at scale. In *2024 IEEE 36th International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 371–378, 2024. doi:10.1109/ICTAI62512.2024.00061.

41 Amalee Wilson, Andres Noetzli, Andrew Reynolds, Byron Cook, Cesare Tinelli, and Clark Barrett. Partitioning strategies for distributed SMT solving. In *2023 Formal Methods in Computer-Aided Design (FMCAD)*, pages 199–208. IEEE, 2023.

42 Haoze Wu, Clark Barrett, and Nina Narodytska. Cubing for tuning. *arXiv preprint arXiv:2504.19039*, 2025. doi:10.48550/arXiv.2504.19039.

43 Hai Xia and Stefan Szeider. SAT-Based tree decomposition with iterative cascading policy selection. *Proceedings of the AAAI Conference on Artificial Intelligence*, 38(8):8191–8199, March 2024. doi:10.1609/aaai.v38i8.28659.

44 Tianwei Zhang, Tomás Peitl, and Stefan Szeider. Small unsatisfiable k-cnfs with bounded literal occurrence. In Supratik Chakraborty and Jie-Hong Roland Jiang, editors, *27th International Conference on Theory and Applications of Satisfiability Testing, SAT 2024, August 21-24, 2024, Pune, India*, volume 305 of *LIPIcs*, pages 31:1–31:22. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024. doi:10.4230/LIPIcs.SAT.2024.31.

45 Tianwei Zhang and Stefan Szeider. Searching for smallest universal graphs and tournaments with SAT. In *29th International Conference on Principles and Practice of Constraint Programming, CP 2023, August 27-31, 2023, Toronto, Canada*, volume 280 of *LIPIcs*, pages 39:1–39:20. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023. doi:10.4230/LIPIcs.CP.2023.39.