



NORTHWESTERN UNIVERSITY

Computer Science Department

Technical Report
Number: NU-CS-2026-26

May, 2026

Learning Better Heuristics for SAT Solving using Machine Learning

Aryaman Chawla

Abstract

Modern SAT solvers are pre-programmed with a fixed set of heuristic parameters. While these defaults work well on most problems, they underperform on certain types of SAT formulas. This dissertation investigates whether a machine-learning model can use characteristics of a SAT problem to predict a modification to MiniSat's heuristics that results in faster solving as compared to the default. The system we developed reads a SAT formula, extracts 43 structural features, and uses these features as input to a pre-trained model that selects a heuristic configuration from a portfolio of 301 configurations covering MiniSat's eight-dimensional heuristic parameter space. The configuration is called directly from MiniSat through a `--use-ml` command-line flag, and no external computation beyond the prediction is required at runtime. When testing on 189 unseen SAT formulas drawn from SAT Competition problem databases, the system achieves a 1.27x geometric-mean speedup. On a subset of 32 problems where neither configuration timed out and the runtime exceeded two seconds, it achieved a 1.88x geometric-mean speedup, with 20 wins, 11 losses and 1 tie. The largest individual speedup exceeded 178x, confirming that the model delivers substantial wall-clock time improvements.

Keywords

SAT solving, Machine Learning

NORTHWESTERN UNIVERSITY

**Learning Better Heuristics for SAT Solving using
Machine Learning**

A DISSERTATION

SUBMITTED TO THE GRADUATE SCHOOL
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

for the degree

MASTER OF SCIENCE

Computer Science

By

Aryaman Chawla

EVANSTON, ILLINOIS

May 2026

Abstract

Modern SAT solvers are pre-programmed with a fixed set of heuristic parameters. While these defaults work well on most problems, they underperform on certain types of SAT formulas. This dissertation investigates whether a machine-learning model can use characteristics of a SAT problem to predict a modification to MiniSat’s heuristics that results in faster solving as compared to the default. The system we developed reads a SAT formula, extracts 43 structural features, and uses these features as input to a pre-trained model that selects a heuristic configuration from a portfolio of 301 configurations covering MiniSat’s eight-dimensional heuristic parameter space. The configuration is called directly from MiniSat through a `--use-ml` command-line flag, and no external computation beyond the prediction is required at runtime. When testing on 189 unseen SAT formulas drawn from SAT Competition problem databases, the system achieves a $1.27\times$ geometric-mean speedup. On a subset of 32 problems where neither configuration timed out and the runtime exceeded two seconds, it achieved a $1.88\times$ geometric-mean speedup, with 20 wins, 11 losses and 1 tie. The largest individual speedup exceeded $178\times$, confirming that the model delivers substantial wall-clock time improvements.

Acknowledgments

I am immensely thankful to my supervisor, Prof. Ian Horswill, who supported me throughout this journey and without whom this work would not have been possible. I am extremely grateful for his consistent academic and personal support. I also thank my friends and family for their patience dealing with my frequent absences and constant distractions.

Contents

Abstract	2
Acknowledgments	3
1 Introduction	7
1.1 Background and Motivation	7
1.2 Problem	7
1.3 Goal	8
1.4 Research Questions	8
1.5 Hypotheses	9
1.6 Contributions	9
1.7 Audience	9
1.8 Dissertation Outline	10
2 Literature Review	11
2.1 Formula Structure and Solver Difficulty	11
2.2 Early SAT Algorithms	12
2.3 Conflict-Driven Clause Learning	12
2.4 Restart Strategies and Clause Database Management	14
2.5 Preprocessing and Inprocessing	14
2.6 Parallel and Portfolio SAT Solving	15
2.7 Stochastic Local Search	15
2.8 Algorithm Configuration	16
2.9 Algorithm Selection	16
2.10 Deep Learning and Neural SAT Solvers	16
2.11 Limitations of Current Approaches	17

	5
2.12 Why MiniSat Remains an Important Baseline	18
2.13 Summary	18
3 System Design	19
3.1 Overall Pipeline	19
3.2 Bias in the Pipeline	20
3.3 MiniSat Parameters	20
3.4 Features	21
3.5 Model	23
4 Methodology	24
4.1 Experimental Setup	24
4.2 Generating Training Data	24
4.3 Model Training	24
4.4 Baselines and Evaluation	25
5 Implementation	27
5.1 Codebase Architecture	27
5.2 Tools Used	27
5.3 Reproducibility	27
6 Results	28
6.1 Data Collection	28
6.2 Cover Construction	28
6.3 Cross-Validated Model Performance	28
6.4 Benchmark Evaluation	29
6.5 Portfolio Coverage	30
6.6 Predictor Overhead	31

	6
7 Conclusion	33
7.1 Validity	33
7.2 Summary	33
7.3 Answers to the Research Questions	34
7.4 When ML Configuration Helps and When It Does Not	35
7.5 Comparison to Prior Work	35
7.6 Future Work	35
A Appendix: Heuristic Parameter Ranges and Defaults	41
B Appendix: Full Feature List	41
C Appendix: Reproduction Recipe	43

1 Introduction

1.1 Background and Motivation

The Boolean satisfiability problem, more commonly known as SAT, asks whether a given propositional formula can be satisfied (i.e., made true) by some assignment of its variables. SAT was the first problem proved NP-complete [1], meaning that any candidate solution can be verified in polynomial time, but no known algorithm can find one efficiently. Despite this theoretical barrier, modern SAT solvers routinely solve industrial formulas containing millions of variables and clauses and are deployed in diverse fields such as cryptanalysis and software-package management [2].

Modern state-of-the-art complete SAT solvers are based on the Conflict-Driven Clause Learning (CDCL) algorithm. A CDCL solver makes numerous heuristic decisions at runtime, such as how often to restart, how aggressively to delete learned clauses, and whether to cache variable polarities. MiniSat [3] packaged these ideas into a compact and modular open-source solver that became the foundation for dozens of award-winning SAT Competition entries and remains a widely used reference implementation to this day. Its heuristic behavior is governed by a set of parameters whose default values were empirically chosen by the authors.

1.2 Problem

The default heuristics perform well on average, but they are far from optimal on many realistic workloads. For example, random 3-SAT formulas near the phase transition, large industrial circuits, and hand-crafted combinatorial puzzles have very different structures [4], and a heuristic that works well on one problem may not work well on another, even within the same family of problems. Moreover, tuning the restart policy, variable-activity decay, or clause-deletion threshold for a specific input can change the total runtime by several orders

of magnitude, and doing this manually for every new problem is infeasible.

1.3 Goal

This dissertation aims to build a system that automatically selects a MiniSat heuristic configuration for each input formula and to measure whether doing so outperforms the default parameters. The system must:

1. Read a SAT formula and compute a fixed set of inexpensive structural features.
2. Use a trained machine learning (ML) model to predict a configuration that is likely to solve the problem faster than the default.
3. Apply the predicted configuration inside MiniSat and run the solver.
4. Achieve a runtime improvement over default MiniSat on unseen Boolean formulas.

1.4 Research Questions

This dissertation answers three questions:

1. Can a model that predicts a heuristic configuration beat the MiniSat defaults on a realistic mix of SAT benchmark problems?
2. Which structural features of a SAT formula carry useful information for choosing a configuration, and why?
3. What is the runtime overhead of feature extraction and model prediction, and does this overhead negate the predicted speedup in practice?

1.5 Hypotheses

We hypothesize that a heuristic configuration chosen by an ML model will yield an average speedup over the defaults on difficult SAT problems, but that speedup will diminish on easier formulas. A secondary hypothesis is that the most informative features will be the interactions between features and parameters.

1.6 Contributions

This work makes the following contributions:

1. A feature extractor that computes 43 structural features from a Boolean formula.
2. A portfolio of 301 MiniSat configurations consisting of the default plus a set of configurations sampled via Latin Hypercube Sampling across MiniSat’s eight-dimensional heuristic parameter space.
3. A data-collection pipeline that times MiniSat under every heuristic configuration in the portfolio on a variety of SAT formulas.
4. An ML model that scores each heuristic configuration on a given SAT problem and selects the best one, embedded within MiniSat.
5. An evaluation suite that runs MiniSat with default heuristics and MiniSat with the ML-chosen heuristic configuration, and compares the results.

1.7 Audience

This work is most relevant to three audiences. For researchers and SAT solver developers, it provides a compact, reproducible module that equips a single classical SAT solver with the ability to optimize itself for each problem. For students and educators, the small size of

MiniSat, the readability of the predictor, and the simplicity of the model training loop make this work a useful case study in augmenting an algorithmic system with classical machine learning.

1.8 Dissertation Outline

Section 2 reviews related work in SAT solving, algorithm configuration, and portfolio-based solvers. Section 3 describes the system design. Sections 4 and 5 detail the methodology and implementation. Section 6 reports experimental results, and Section 7 concludes with a discussion of when machine learning helps, when it does not, and potential directions for future work.

2 Literature Review

Just as different algebraic representations can express the same concept, different Boolean structures can represent the same SAT formula, each with distinct computational advantages. For example, if the goal is to directly read off a satisfying assignment without performing any complex analysis, Disjunctive Normal Form (DNF) is ideal because one only needs to find a single satisfying clause. On the other hand, Boolean Circuits and Negation Normal Form (NNF) are used to avoid the memory blowup that can occur when converting a formula into Conjunctive Normal Form (CNF).

Within the realm of SAT solving, SAT formulas are most commonly expressed in CNF, where the formula is a conjunction (logical AND) of clauses, each clause is a disjunction (logical OR) of literals, and each literal is either a variable or its negation (logical NOT). This uniform "AND of ORs" structure mirrors real-world constraint checklists, in which every rule must be satisfied. Standardizing the format to CNF has simplified algorithm design, as any SAT problem can be efficiently converted into CNF without an exponential blowup using the Tseitin transformation.

Barring any major complexity-theoretic breakthrough [1], no algorithm today can solve every SAT formula quickly. However, modern SAT solvers routinely solve industrial formulas containing millions of variables and clauses, and this gap between theory and practice motivates ongoing research into improving solver configuration and selection.

2.1 Formula Structure and Solver Difficulty

A key reason SAT is tractable in practice is the structure of different formula classes. Cheeseman, Kanefsky, and Taylor first identified a sharp phase transition in the difficulty of NP-complete problems, as some parameter crosses a critical value, noting that the hardest problems cluster at the boundary between under- and over-constrained regions [5]. Mitchell,

Selman, and Levesque refined this for SAT, showing that for random 3-SAT formulas the hardest clause-to-variable ratio is about $m/n \approx 4.25$ [4], later narrowed to ≈ 4.267 . In contrast, industrial SAT problems exhibit a community structure, where variables cluster into groups that occur across many clauses [6], and this modularity correlates strongly with CDCL solve time [7]. Horn formulas are SAT formulas in CNF where every clause contains at most one positive literal. They form another special case and are solvable in linear time [8]. These structural differences explain why a single default heuristic cannot perform equally well across all formula types.

2.2 Early SAT Algorithms

Davis and Putnam introduced the DP procedure, a resolution-based algorithm that eliminates variables one by one: for each variable, it replaces every clause pair containing opposite-polarity occurrences with their resolvent and removes the variable [9]. While theoretically sound, the procedure was impractical for large formulas because the cross-multiplication in the resolution step incurred an exponential memory cost. Davis, Logemann, and Loveland soon replaced variable elimination with recursive branching and chronological backtracking, creating the DPLL algorithm [10]. DPLL maintains a partial assignment, propagates unit clauses, recursively branches on an unassigned variable, and backtracks upon conflict. This assign-propagate-backtrack pattern remains the conceptual backbone of modern SAT solving.

2.3 Conflict-Driven Clause Learning

Modern complete SAT solving relies on the CDCL framework. The GRASP solver [11] first implemented CDCL. It analyzed each conflict and non-chronologically backtracked to the second-highest decision level implicated in the learned clause. On the hardest benchmarks of

the time, GRASP achieved orders-of-magnitude speedups over prior DPLL implementations. Conflict-driven clause learning allowed the solver to incrementally refine the formula during search, rather than blindly exploring the entire tree.

Chaff showed that careful engineering of CDCL could yield further speedups [12]. Its key innovation was a two-watched-literal Boolean Constraint Propagation (BCP) scheme, where only two unassigned literals per clause are monitored, rather than every literal. In addition, the VSIDS branching heuristic prioritized variables involved in recent conflicts using an array lookup, replacing a full conflict scan. Together, these allowed Chaff to handle problems an order of magnitude faster than GRASP.

MiniSat built directly on this foundation, with readability as its hallmark. Its entire CDCL loop comprised about six hundred lines of C++ code and was simple enough to be adapted by students and researchers [3], making it the reference implementation for SAT research, teaching, and solver extensions. A later version added conflict-clause minimization, in which literals are recursively removed from a learned clause to reduce memory usage [13].

A next-generation solver, Glucose, later introduced the Literal Block Distance (LBD) metric. This metric represents the number of distinct decision levels among its literals, and clauses with $lbd \leq 2$ (glue clauses) are treated as long-lived because they connect blocks of propagation that the solver has already committed to [14]. Glucose used LBD both to guide aggressive clause-database reduction and to trigger restarts when the moving average of recent LBDs was large. MapleSAT later combined LBD-based retention with a Learning-Rate Branching (LRB) heuristic that modeled each variable as a multi-armed bandit, updating its score via an exponentially recency-weighted average of the rate at which selecting that variable produced learned clauses [15].

Recent solvers push this engineering further. CaDiCaL was redesigned from scratch around a clean separation of search, inprocessing, and proof recording, with a stable propagator interface that lets external programs plug in [16]. Kissat, by contrast, is optimized for

raw speed at the cost of flexibility. CaDiCaL and Kissat represent the state of the art for sequential CDCL solving and serve as useful baselines for evaluating new systems.

2.4 Restart Strategies and Clause Database Management

A weakness of CDCL solvers is that they can become trapped in unproductive regions of the search space for long periods. Gomes et al. showed that the runtime distribution of complete backtracking SAT solvers is heavy-tailed [17]. Specifically, a minor change in the search trajectory, such as an early wrong decision, can cause runtime to explode from seconds to days. Random restarts counter this by abandoning the current path and exploring a new region of the search tree while retaining learned clauses. Earlier solvers used static restart schedules, and MiniSat popularized dynamic schedules [3].

Clause database management, tightly coupled with restarting, is another critical aspect of SAT solving. As CDCL solvers run, they generate learned clauses continuously, some of which become redundant after a restart. Keeping every clause slows propagation and wastes memory, but discarding useful clauses may force the solver to search the same region of the search space repeatedly, leading to an infinite loop. MiniSat used activity-based deletion, where a clause’s activity increases with each conflict it is involved in and decays multiplicatively thereafter [3]. These two mechanisms are among the most important tunable parameters, and their interaction depends heavily on the problem type, which is why the eight-parameter MiniSat heuristic configuration space targeted in this dissertation contains both `rfirst` and `gc_frac`.

2.5 Preprocessing and Inprocessing

Preprocessing became essential alongside CDCL. SatELite introduced bounded variable elimination—deleting a variable by resolving all clauses containing it and its negation—along

with self-subsumption and clause strengthening [18]. Solvers like Lingeling later integrated such simplifications periodically during search, a technique termed inprocessing [19]. Modern solvers such as Kissat continue this trend, applying aggressive inprocessing throughout the solve to continuously simplify the formula.

2.6 Parallel and Portfolio SAT Solving

Parallel SAT solving emerged from the observations that no single heuristic dominates all formulas and that SAT runtime distributions are heavy-tailed. Portfolio-based parallel solvers such as ManySAT run multiple differently configured copies of MiniSat concurrently, exchanging short learned clauses through a lock-free shared buffer [20]. Plingeling extends this idea to dozens of Lingeling workers and shares both unit clauses and equivalences [21]. Cube-and-Conquer uses a different approach. It partitions a hard formula into disjoint sub-problems (cubes), solves them independently, and combines the results [22]. These approaches demonstrate that performance can improve not only through better per-solver heuristics but also through coordination among diverse strategies.

2.7 Stochastic Local Search

Instead of systematic tree search, SLS solvers iteratively improve a candidate assignment by flipping variable values according to local criteria. GSAT performs greedy hill-climbing on the number of satisfied clauses, breaking ties uniformly at random and restarting after a fixed flip budget [23]. WalkSAT refined this by interleaving greedy and random flips [24]. This randomized strategy enables WalkSAT to solve large random satisfiable formulas extremely quickly, often outperforming CDCL on such problems. However, SLS cannot prove unsatisfiability, which is why this dissertation builds on a complete CDCL solver.

2.8 Algorithm Configuration

Algorithm configuration asks if there is a fixed parameter setting that performs well across a distribution of formulas. ParamILS searches the parameter space via iterated local search, sequentially modifying one parameter and choosing the direction that most improves solve time [25]. Irace takes a complementary statistical approach, racing candidate configurations and eliminating those worse than the current best [26].

SMAC takes a model-based approach, building a random-forest surrogate of solver performance over the (configuration, formula-feature) space and selecting the next configuration by maximizing expected improvement [27]. This dissertation follows a similar tree-ensemble philosophy but uses gradient boosting rather than random forests, and predicts a configuration per problem instead of a single configuration for a whole family of problems.

2.9 Algorithm Selection

Algorithm selection is the problem of choosing the best algorithm for a given problem based on measurable features [28]. SATzilla implemented this idea [29] by computing 48 structural features, feeding them into a linear model to predict the runtime of several independent solvers, and selecting the fastest. In such systems, feature engineering is the most important aspect, as the features must be cheap to compute but still capture enough of the problem structure for an ML model to exploit. This dissertation follows the same philosophy but predicts a configuration of a single solver rather than choosing among several different solvers and maintaining different code-bases.

2.10 Deep Learning and Neural SAT Solvers

Over the past decade, deep learning has entered SAT research. NeuroSAT showed that a graph neural network (GNN) can learn search-like behavior [30]. Although it was not

competitive with algorithmic solvers in raw runtime, it demonstrated that learned graph representations can capture meaningful SAT formula structure. Neural approaches, however, impose substantial computational overhead, requiring GPUs and extremely large training datasets. In contrast, tree-based models offer fast CPU inference and integrate naturally into a lightweight solver. This trade-off motivated our decision to use classical hand-engineered structural features with gradient-boosted trees instead.

2.11 Limitations of Current Approaches

Several well-documented weaknesses of existing approaches influenced this dissertation.

First, many portfolio solvers rely on multiple independent solvers (MiniSat, Glucose, Lingeling, etc.). While this improves benchmark scores, it results in systems that are difficult to maintain and deploy, as solvers differ in parameter interfaces, preprocessing pipelines, and clause-sharing mechanisms. This dissertation instead focuses on a single configurable base solver, MiniSat, which keeps the implementation compact and makes comparisons against a fixed baseline simple.

Second, prediction overhead can dominate runtime on small formulas. Portfolio approaches were designed for SAT Competition problems where runtimes last minutes or hours, making a few seconds of feature extraction acceptable. However, on easy problems, such overhead may exceed any runtime savings.

Finally, many ML-guided SAT systems are hard to reproduce. Solver implementations change rapidly, datasets are often unavailable, and small preprocessing differences can significantly alter results. One motivation for this dissertation is to build a lightweight, reproducible ML pipeline.

2.12 Why MiniSat Remains an Important Baseline

Despite the availability of more powerful solvers, MiniSat remains a valuable baseline [3]. It is small enough to understand and modify entirely, but still complete enough to reflect the core ideas of modern CDCL solving. Its simple structure, few inprocessing layers, and direct mapping from parameters to observable behavior make it a clean platform for testing the system.

2.13 Summary

The literature traces a progression from early complete search to modern learning-based, randomized methods. DPLL established the framework and CDCL made SAT solving practical. MiniSat made CDCL compact and reusable, and Kissat pushed pure search performance further. Portfolio and configuration methods demonstrated that the optimal behavior often depends on the specific problem. Learning-based approaches showed that machine learning can be applied directly inside SAT solvers, albeit often at a computational cost.

This dissertation builds on that literature, asking whether machine learning can predict a good MiniSat configuration well enough to yield real runtime gains without sacrificing MiniSat’s simplicity. The system combines feature-based methods from SATzilla with the configuration-prediction ideas of SMAC, applying them to a single compact solver rather than a heterogeneous portfolio of SAT solvers.

3 System Design

3.1 Overall Pipeline

The system comprises of an "offline" training phase and an "online" prediction phase. The training phase produces a single binary file that the MiniSat prediction module uses to evaluate MiniSat heuristic configurations.

The training phase runs before runtime and consists of four steps. First, a portfolio of MiniSat heuristic configurations is constructed - one is the MiniSat default, and the rest are generated using Latin Hypercube Sampling (LHS) over the eight tunable parameter space. LHS divides each parameter axis into N equal-probability bins, draws exactly one sample per bin per axis, and then permutes the bin assignments across axes, ensuring complete coverage without the clustering typical of uniform random sampling. Second, for each SAT problem, a 43-dimensional feature vector is extracted by reading the file and computing structural statistics. Third, every portfolio configuration is timed on every training problem, producing an empirical performance matrix (EPM) of wall-clock runtimes or timeout markers. Finally, this matrix is used to train a model that scores each configuration for an unseen formula on predicted penalized-runtime, and the configuration with the lowest score is selected.

The prediction phase adds a single `--use-m1` flag to MiniSat. When called with this flag, MiniSat reads the input CNF, extracts the feature vector, runs the trained model, selects the predicted-best configuration, and then runs the solver with that configuration. In this way, the core CDCL code remains unchanged, and any speedup or slowdown stems solely from the chosen parameter settings.

3.2 Bias in the Pipeline

The system implements two safety mechanisms to reduce the risk of the model hurting performance.

The first mechanism is a "simple formula" gate. Before committing to a predicted configuration, the gate estimates the default MiniSat runtime on the problem and overrides the model's choice if that estimate falls below a certain threshold of two seconds. On simple formulas, MiniSat solves the problem quickly regardless of configuration, so the overhead of prediction plus the lack of room for improvement can cause a slowdown.

The second mechanism is the choice of training label. The training target is the PAR-2 score. Penalized Average Runtime (PAR) with penalty multiplier k records the actual wall-clock time on solved formulas and $k \times$ wall limit on unsolved ones. PAR-2 uses $k = 2$, so a timeout is recorded as twice the time budget. This discourages the model from selecting configurations that occasionally finish quickly but often time out. Moreover, if every portfolio configuration times out on a formula, that problem is excluded because it provides no information on which configuration provides an improvement. The ranker is trained on relative PAR-2 and weighted by problem difficulty, so that hard problems dominate the loss.

3.3 MiniSat Parameters

The portfolio varies eight MiniSat parameters, listed in Table 1. These eight parameters were chosen because they govern the key aspects of CDCL search: variable selection, restarts, clause learning, and memory management.

The two decay parameters control how quickly the solver "forgets" old conflicts. With a low decay, activity scores fade slowly, so the solver retains many past conflicts and explores broadly.

The restart parameters control how often the solver abandons its current search path

and restarts while retaining learned clauses. Restarting too rarely can waste time on dead ends but restarting too often may drive the solver into sparser tree regions and risk infinite looping.

Phase saving remembers the last assigned truth value for each variable and reuses it on subsequent decisions, aiding navigation when the solver revisits previously explored parts of the search space.

Conflict-clause minimization controls how hard the solver works to shorten each learned clause after a conflict. Shorter clauses propagate more effectively but take more time to produce.

Finally, the garbage collection parameter controls how aggressively learned clauses that are no longer useful are deleted to free memory. On large problems, this can determine whether MiniSat stays within the memory limit at all.

Parameter	Values	Controls
<code>vsids_decay</code>	[0.70, 0.9995]	Variable activity decay
<code>clause_decay</code>	[0.85, 0.9999]	Learned clause activity decay
<code>restart_type</code>	{Luby, geometric}	Restart sequence
<code>restart_factor</code>	[1.01, 4.00]	Restart growth ratio
<code>rfirst</code>	[10, 1000]	First restart length
<code>phase_saving</code>	{0, 1, 2}	Polarity caching
<code>ccmin_mode</code>	{0, 1, 2}	Conflict-clause minimization mode
<code>gc_frac</code>	[0.05, 0.50]	Learned clause GC threshold

Table 1: The eight MiniSat parameters varied in the portfolio.

3.4 Features

Each Boolean formula is described by a 43-dimensional feature vector, grouped into seven categories as shown in Table 2. All features can be computed in a single pass over the formula and require no solver runs.

Raw size features are the most basic and typically correlate with memory-management

Group	Examples
Raw size	<code>n_vars</code> , <code>n_clauses</code> , <code>literal_count</code>
Clause-length	<code>clause_len_mean</code> , <code>binary_clause_frac</code> , <code>long_clause_frac</code>
Horn clauses	<code>horn_frac</code> , <code>dual_horn_frac</code> , <code>strict_horn_frac</code>
Variable-degree moments	<code>var_degree_mean</code> , <code>var_freq_skew</code> , <code>top10pct_var_degree_share</code>
Polarity	<code>pos_neg_ratio</code> , <code>polarity_balance_mean</code> , <code>polarity_balance_var</code>
Hub contribution	<code>var_contrib_mean</code> , <code>var_contrib_max</code>
VIG	<code>vig_mean_degree</code> , <code>vig_var_degree</code> , <code>vig_max_degree</code>

Table 2: Feature groups (43 features in total).

demands. Clause-length statistics indicate the structural type of the formula. Horn clause fractions measure the proportion of clauses with at most one positive literal, as problems with a high Horn fraction are structurally simpler and known to be easier to solve [8] using forward chaining and unit propagation.

Variable-degree moments and hub-contribution features capture how unevenly variables appear across clauses. The four moments computed are mean, standard deviation, skew, and kurtosis. Industrial problems, for example, often have a small number of hub variables, which can guide the solver to branch on them.

Polarity features measure how often each variable appears positively versus negatively. A strong polarity imbalance interacts with phase saving: if a variable is nearly always positive, caching that polarity is more likely to lead quickly to a solution.

VIG-degree features describe the Variable Incidence Graph (VIG), where nodes represent variables and edges connect variables that co-occur in at least one clause. VIG degree statistics capture how tightly coupled variables are—a strong correlate of solver difficulty. For very large problems, exact VIG computation is expensive, so the extractor randomly samples clauses and caps the total work, ensuring the resulting statistics are within sampling noise of the exact values.

3.5 Model

The model takes as input the 43-dimensional feature vector and a 13-dimensional encoding of the candidate configuration (the eight raw parameter values plus five derived terms). It also evaluates eight feature $\times$ parameter interaction terms, chosen because classical algorithm-configuration studies and our own empirical tests identified them as the most informative parameter-structure pairs.

The scoring head is a bagged ensemble of 40 XGBoost rankers, each an 800-tree ensemble with maximum depth 10, trained with a pairwise ranking objective so that each booster outputs a relative ordering score over candidate configurations, not a calibrated runtime estimate. Bagging here means that each of the 40 boosters trains on the same data but with a different random seed for tree construction, and the outputs are normalized and averaged.

At inference time, the bag is queried once per configuration (40 evaluations per problem), and the configuration with the lowest composite score is selected.

4 Methodology

4.1 Experimental Setup

All experiments are run on a single 24-core x86_64 workstation under Linux with 32 GB of RAM. The MiniSat binary is built once with the upstream Makefile, with no changes to the solver code. Random seeds are fixed throughout, ensuring the pipeline is reproducible.

4.2 Generating Training Data

The training data take the form of an Empirical Performance Matrix (EPM). This is a table with rows for SAT problems and columns for heuristic configurations, where each entry records the wall-clock runtime of MiniSat under that configuration on that problem (or a TIMEOUT/ERROR marker). Every (problem, configuration) cell is timed exactly once, with a 3600-second wall limit and a 4 GB memory limit.

All 301 configurations are timed in parallel on the same problem, yielding a 206×301 EPM containing 62006 (problem, configuration) cells.

4.3 Model Training

Training proceeds in two steps. First, the EPM is loaded and relative PAR-2 scores are computed, weighted by problem difficulty. The weight is proportional to $\log(1 + \text{PAR2}_{\min})$, so hard problems contribute more to the loss. The same EPM is used to construct the 40-member greedy oracle cover of heuristic configurations (cf. §6.2).

Second, XGBoost hyperparameters are tuned using Bayesian optimization. The search runs for 25 iterations over the joint search space:

- `max_depth` $\in [3, 10]$,
- `learning_rate` $\in [0.01, 0.30]$,

- `subsample` $\in [0.5, 1.0]$,
- `colsample_bytree` $\in [0.5, 1.0]$,
- `min_child_weight` $\in [1, 10]$,
- `reg_lambda` $\in [10^{-3}, 10]$,
- `reg_alpha` $\in [10^{-4}, 1]$,
- `gamma` $\in [0, 5]$,
- `num_round` $\in [200, 1200]$ trees.

The validation score at each step is the geometric-mean PAR-2 across the held-out folds of a 5-fold cross-validation. The problem set is shuffled randomly and split into five equal partitions, the rankers are trained on four partitions, and then evaluated on the fifth. Tuning runs separately for the ranker objective and for the regression objective so that each head receives hyperparameters tailored to its loss. Finally, the tuned model is serialized, to be used by MiniSat at runtime.

4.4 Baselines and Evaluation

The baseline is unmodified MiniSat, invoked with only wall-clock and memory limits.

The system is evaluated on 189 unseen SAT problems. For each problem, the test suite extracts features once, selects a configuration, and then runs both the default configuration and the ML-chosen configuration.

The primary metric is the geometric mean of PAR-2 speedup ratios. The geometric mean of n positive numbers $x_1, \dots, x_n$ is $(\prod_i x_i)^{1/n}$. It is the appropriate average for multiplicative quantities such as speedups because it gives equal weight to a $2\times$ speedup and a $2\times$ slowdown, whereas the arithmetic mean would report a $1.25\times$ speedup. For each problem,

the speedup is computed as $\text{PAR2}_i^{\text{default}}/\text{PAR2}_i^{\text{ML}}$, where $\text{PAR2}_i = t_i$ on solved problems and $\text{PAR2}_i = 2 \times \text{wall time}$ on unsolved ones. The test suite also reports win/loss/tie counts using a $\pm 5\%$ threshold - a win if the speedup exceeds $1.05\times$, a loss if below $0.95\times$.

5 Implementation

5.1 Codebase Architecture

The repository is organized into two top-level directories: `minisat/` and `ml-config/` (the Python pipeline we built for data generation, model training, and evaluation).

5.2 Tools Used

The training pipeline depends on five external libraries. At solve time, the modified MiniSat has no runtime dependencies.

5.3 Reproducibility

All stochastic components of the pipeline are seeded, but any step involving MiniSat remains inherently unpredictable due to MiniSat’s internal randomness.

6 Results

6.1 Data Collection

The data-collection sweep ran with a 3600 s wall-clock limit and a 4 GB memory cap per (problem, configuration) pair. The resulting EPM contains 62006 cells (206 problems $\times$ 301 configurations). Of these, 20746 finished SAT, 12213 finished UNSAT, and the remaining 29047 timed out or exceeded memory. This 53% timeout rate reflects the heavy-tailed nature of SAT solving. Of the 206 formulas, 144 have at least one configuration that solved within the wall budget and thus contribute to the relative-PAR-2 target, and 143 of those 144 contain at least one configuration strictly faster than the default, thus forming the training data for the ranker.

6.2 Cover Construction

A greedy oracle cover is built from the EPM by iteratively adding the configuration that yields the largest marginal PAR-2 improvement over the currently selected set. Formally, given a partial cover $\mathcal{C}_k$ after k rounds, the next configuration is chosen to minimize the sum over training problems of the minimal PAR-2 of the cover plus that configuration. The procedure stops when the cover reaches $K = \min(40, \lfloor 0.7N \rfloor)$ configurations, where N is the number of scorable training problems. The remaining 261 portfolio configurations are still used during training, but at runtime only the 40 cover members are evaluated, so there is a small, fixed inference wall-clock cost.

6.3 Cross-Validated Model Performance

The cross-validated results are summarized in Table 3. The system achieves a geometric mean PAR-2 speedup of 19% over the default, with 91 wins, 33 losses, and 20 ties at the

$\pm 5\%$ threshold. The mean PAR-2 across the 144 problems drops from 1749 s to 1340 s. While substantial headroom remains, these results demonstrate competitive performance for a single-configuration method on this distribution.

Approach	geo-speedup	mean PAR-2 (s)	wins	losses	ties
Default	1.000×	1749.5	—	—	—
ML chosen config	1.186×	1339.6	91	33	20

Table 3: 5-fold cross-validated PAR-2 speedup of ML-enhanced MiniSat heuristics over default MiniSat heuristics, 3600 s wall limit, "simple-formula" gate at 50 s.

6.4 Benchmark Evaluation

We assessed the system on a previously unseen set of 189 SAT problems drawn from a database of SAT Competition problems, using a 600 s wall limit per run. On 105 of the 189 formulas, "simple-formula" gate predicted that the default configuration would not finish within 50 seconds, and the model's choice was retained. Across all 189 problems, the selected configurations were dominated by four cover members: 293 (73 instances), 280 (72), 287 (16), and 267 (15), with a long tail (configurations 298, 284, 289, 0, 1, 273) accounting for the remainder (cf. §6.5).

The raw aggregate over all 189 problems is dominated by dual timeouts and by trivially short solves, where wall-time differences are due to process startup time. Consequently, the unfiltered geometric-mean PAR-2 speedup across all 189 problems is only 1.27×. To focus on cases where the configuration choice is actionable, we filter out (i) problems on which either the default or the ML-chosen configuration timed out, and (ii) problems where the default configuration solved in under two seconds. After filtering, 32 problems remain on which both configurations completed and the default did real solver work.

On this filtered set, the geometric-mean PAR-2 speedup is 1.88×, and the aggregate ratio of total solve times is 1.83×. At the $\pm 5\%$ threshold, the win/loss/tie counts are

20/11/1. The largest individual win is a $178.6\times$ speedup and the largest loss is a $0.008\times$ speedup (i.e., a $125\times$ slowdown), which occurred when the model selected a configuration with an unfavorable restart schedule. These results are summarized in Table 4 and visualized in Figures 1–2.

Table 4: Evaluation results (no timeouts, default ≥ 2 s; $N = 32$).

Metric	Value
Geometric-mean PAR-2 speedup	$1.876\times$
Aggregate PAR-2 speedup ($\sum t_{\text{def}} / \sum t_{\text{ml}}$)	$1.827\times$
Median PAR-2 speedup	$1.133\times$
Wins / losses / ties (at $\pm 5\%$)	20 / 11 / 1
Largest single win / loss	$178.57\times$ / $0.008\times$
Default total solve time	3501.7 s
ML total solve time	1916.4 s

Thus, on problems where the configuration choice actually matters, the model delivers a substantial speedup. The unfiltered results are dominated by timeouts and by short solves where millisecond-scale startup overhead and noise obscure any configuration-driven improvements.

6.5 Portfolio Coverage

On the 189 unseen problems, the model’s selections concentrate on a narrow subset of the 40 cover configurations. The most frequently chosen are 293 (73 instances), 280 (72), 287 (16), and 267 (15), with the remaining 13 spread over six other configurations. Among the 105 problems where the easy-gate did not fire (so the model’s pick was used), configuration 280 dominates (56 instances, 53%), followed by 293 (17), 267 (15), and 287 (9). This pattern aligns with cross-validation winners and confirms that no single configuration is universally optimal, while also showing that the model is selective rather than uniform across the cover members.

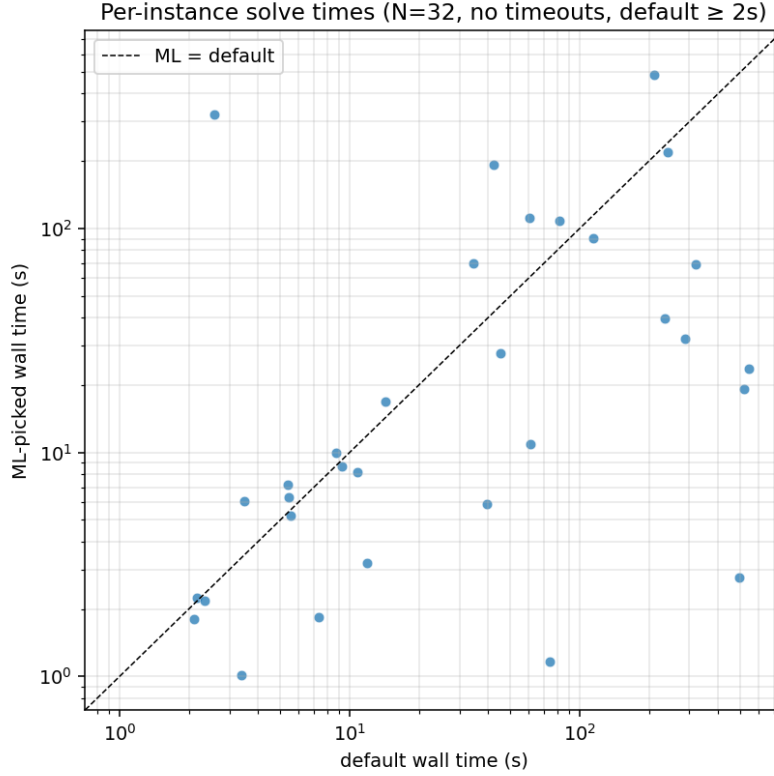


Figure 1: Solve times on the filtered unseen problems ($N = 32$). Points below the dashed identity line are wins for the ML-picked configuration.

6.6 Predictor Overhead

When MiniSat is invoked with `--use-ml`, it parses the CNF, extracts the 43 features, evaluates the easy-gate, and, if the gate does not fire, traverses the 32000 trees of the model to score all 40 cover configurations. The dominant cost is feature extraction, which scales linearly with formula size. Model loading is amortized over the process lifetime, and the tree evaluation per booster is roughly $100\ \mu\text{s}$ and independent of input size.

On the largest benchmark problems, predictor overhead is comfortably under one second, representing a negligible fraction of total solving time on all problems where the easy-gate did not redirect to the default.

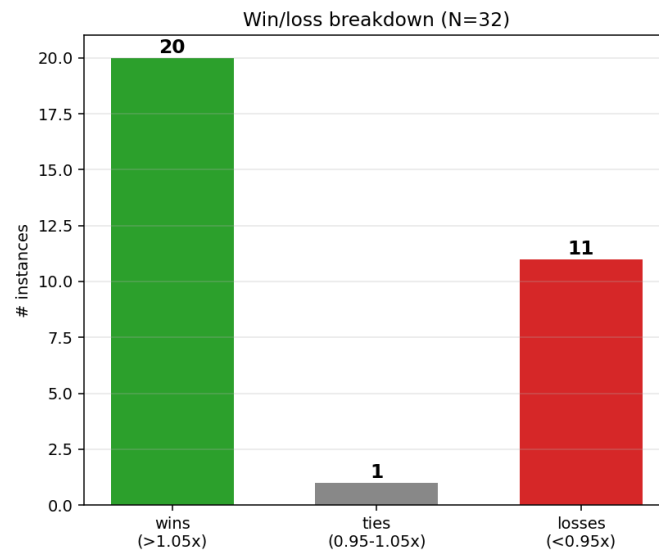


Figure 2: Win/loss/tie breakdown on the filtered unseen subset using a $\pm 5\%$ threshold.

7 Conclusion

7.1 Validity

Two points affect the interpretation of these results. First, the formula pool is not a uniform random sample of all SAT problems. It is drawn from SAT Competition benchmarks, which contain an unknown mix of random and industrial formulas. Generalization to other distributions (e.g., hand-crafted puzzles) therefore remains an open question.

Second, the 3600 s wall-clock limit in the offline data-collection sweep truncates the long tail of solver runtimes. A configuration that would solve a hard problem in, say, 4000 s appears indistinguishable from one that would run forever, even though the former is far more useful. The PAR-2 penalty only partially captures this distinction, and the test suite inherits the same issue. Tightening the test wall time to 600 s amplifies this effect relative to the training sweep, but was necessary to keep testing tractable on a single workstation.

7.2 Summary

This dissertation built an end-to-end heuristic configuration system for MiniSat. The system reads a Boolean formula, computes a 43-dimensional feature vector, and scores the 40 cover configurations using a 40-seed bag of XGBoost pairwise rankers. The configuration with the lowest composite score is selected and passed to the solver. The user-facing entry point is the `--use-ml` flag added to MiniSat. The data-collection script times every portfolio configuration on every training problem. The training script tunes XGBoost hyperparameters via a 25-iteration, 5-fold Bayesian search, fits the model, and produces a flat binary file that MiniSat consumes at runtime with no external dependencies.

7.3 Answers to the Research Questions

(1) Can a model that predicts a heuristic configuration beat the MiniSat defaults on a realistic mix of SAT benchmark problems? Yes. On the filtered benchmark set of 32 difficult formulas, the model achieves a $1.88\times$ geometric-mean speedup with 20 wins, 11 losses, and 1 tie.

(2) Which structural features of a CNF formula carry useful information for choosing a configuration, and why? Three interaction terms proved particularly informative: `horn_frac` $\times$ `clause_decay`, `vig_mean_degree` $\times$ `rfirst`, and `n_clauses` $\times$ `gc_frac`. Each captures a problem-parameter coupling that the model cannot easily reconstruct from raw features alone.

The horn fraction roughly indicates how propagation-dominated a formula is - optimal clause activity decay differs sharply between propagation-dominated and conflict-dominated formulas, so the interaction supplies information that neither feature alone carries.

The mean VIG degree proxies community structure - densely connected formulas benefit from earlier restarts, making the interaction with the first restart interval informative.

The product of clause count and the garbage-collection fraction captures the memory pressure on the learned-clause database. Oversized databases waste memory on hard problems, while undersized ones discard useful clauses on easy problems. The joint term helps the model decide when aggressive garbage collection is beneficial.

(3) What is the runtime overhead of feature extraction and model prediction, and does this overhead ever negate the predicted speedup? Feature extraction and model prediction have a negligible overhead relative to MiniSat solve times - it was comfortably under one second, and never impacted the speedup significantly.

7.4 When ML Configuration Helps and When It Does Not

ML-based configuration is beneficial when (1) the solve time is long enough that feature extraction and inference are negligible, (2) the test formula falls within the structural distribution of the training data, and (3) at least one non-default configuration is faster than the default.

7.5 Comparison to Prior Work

Compared to SATzilla [29], our system is simpler and more deployable. It ships a single solver binary instead of managing a portfolio of independent solvers, and it predicts a configuration for one solver rather than selecting among several. Compared to SMAC [27], our system selects a configuration at solve time rather than searching for one fixed configuration across an entire problem distribution. Compared to graph-neural-network approaches [30], we use hand-engineered features instead of learned representations, enabling training on hundreds rather than hundreds of thousands of instances and eliminating the need for heavy GPU computation at runtime.

7.6 Future Work

Two natural extensions follow. First, the static portfolio of 301 configurations could be replaced by an actively constructed portfolio, e.g., using SMAC to target parameter regions where the model is most uncertain. This would likely yield a smaller, more effective cover. Second, the training EPM is drawn from a single benchmark database and expanding the training distribution would likely improve generalization to a wider variety of unseen formulas.

References

- [1] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing (STOC)*, pages 151–158. ACM, 1971. doi: 10.1145/800157.805047.
- [2] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2021.
- [3] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 2919 of *LNCS*, pages 502–518. Springer, 2004. doi: 10.1007/978-3-540-24605-3_37.
- [4] David Mitchell, Bart Selman, and Hector J. Levesque. Hard and easy distributions of SAT problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI)*, pages 459–465, 1992.
- [5] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the really hard problems are. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 331–337, 1991.
- [6] Carlos Ansótegui, Jesús Giráldez-Cru, and Jordi Levy. The community structure of SAT formulas. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 7317 of *LNCS*, pages 410–423. Springer, 2012. doi: 10.1007/978-3-642-31612-8_31.
- [7] Carlos Ansótegui, Maria Luisa Bonet, Jesús Giráldez-Cru, Jordi Levy, and Laurent Simon. Community structure in industrial SAT instances. *Journal of Artificial Intelligence Research*, 66:443–472, 2019. doi: 10.1613/jair.1.11741.

- [8] William F. Dowling and Jean H. Gallier. Linear-time algorithms for testing the satisfiability of propositional horn formulae. *Journal of Logic Programming*, 1(3):267–284, 1984. doi: 10.1016/0743-1066(84)90014-1.
- [9] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960. doi: 10.1145/321033.321034.
- [10] 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.
- [11] João P. Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999. doi: 10.1109/12.769433.
- [12] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th Design Automation Conference*, pages 530–535, 2001. doi: 10.1145/378239.379017.
- [13] Niklas Sörensson and Niklas Eén. A SAT solver with conflict-clause minimization. In *SAT Competition 2005: Solver Descriptions*, 2005. MiniSat v1.13.
- [14] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, pages 399–404, 2009.
- [15] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki. Learning rate based branching heuristic for sat solvers. In *Theory and Applications of Satisfiability Testing – SAT 2016*, pages 123–140. Springer, 2016.

- [16] Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL 2.0. In *Computer Aided Verification*, LNCS. Springer, 2024.
- [17] Carla P. Gomes, Bart Selman, Neil Crato, and Henry Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24(1):67–100, 2000.
- [18] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 3569 of *LNCS*, pages 61–75. Springer, 2005. doi: 10.1007/11499107_5.
- [19] Armin Biere. Lingeling essentials, a tutorial on design and implementation aspects of the sat solver lingeling. In *POS-14. Fifth Pragmatics of SAT workshop*, volume 27 of *EPiC Series in Computing*, page 88, 2014. doi: 10.29007/jhd7.
- [20] Youssef Hamadi, Said Jabbour, and Lakhdar Sais. ManySAT: A parallel SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 6(4):245–262, 2009. doi: 10.3233/SAT190070.
- [21] Armin Biere. CaDiCaL, Lingeling, Plingeling, Treengeling, YalSAT entering the SAT competition 2017. In Tomáš Balyo, Marijn Heule, and Matti Järvisalo, editors, *Proc. of SAT Competition 2017 – Solver and Benchmark Descriptions*, volume B-2017-1 of *Department of Computer Science Series of Publications B*. University of Helsinki, 2017.
- [22] Marijn J. H. Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In *Hardware and Software: Verification and Testing – 7th International Haifa Verification Conference (HVC 2011)*, volume 7261 of *LNCS*, pages 50–65. Springer, 2012. doi: 10.1007/978-3-642-34188-5_8.
- [23] Bart Selman, Hector J. Levesque, and David Mitchell. A new method for solving

- hard satisfiability problems. In *Proceedings of the National Conference on Artificial Intelligence*, 1992.
- [24] Bart Selman, Henry Kautz, and Bram Cohen. Noise strategies for improving local search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence*, 1994.
- [25] Frank Hutter, Holger H. Hoos, Thomas Stützle, and Kevin Leyton-Brown. ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306, 2009. doi: 10.1613/jair.2861.
- [26] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle. The irace package: Iterated racing for automatic algorithm configuration. *Operations Research Perspectives*, 3:43–58, 2016. doi: 10.1016/j.orp.2016.09.002.
- [27] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Learning and Intelligent Optimization (LION)*, volume 6683 of *LNCS*, pages 507–523. Springer, 2011. doi: 10.1007/978-3-642-25566-3_40.
- [28] John R. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976. doi: 10.1016/S0065-2458(08)60520-3.
- [29] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, 2008. doi: 10.1613/jair.2490.
- [30] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and

David L. Dill. Learning a SAT solver from single-bit supervision. In *International Conference on Learning Representations (ICLR)*, 2019.

A Appendix: Heuristic Parameter Ranges and Defaults

Parameter	Type	Range / values	Default	MiniSat flag
<code>vsids_decay</code>	continuous	[0.70, 0.9995]	0.95	<code>-var-decay</code>
<code>clause_decay</code>	continuous	[0.85, 0.9999]	0.999	<code>-cla-decay</code>
<code>restart_type</code>	categorical	{Luby, geometric}	Luby	<code>-luby/-no-luby</code>
<code>restart_factor</code>	continuous	[1.01, 4.00]	2.0	<code>-rinc</code>
<code>rfirst</code>	integer	[10, 1000]	100	<code>-rfirst</code>
<code>phase_saving</code>	categorical	{0, 1, 2}	2	<code>-phase-saving</code>
<code>ccmin_mode</code>	categorical	{0, 1, 2}	2	<code>-ccmin-mode</code>
<code>gc_frac</code>	continuous	[0.05, 0.50]	0.20	<code>-gc-frac</code>

Table 5: Portfolio parameters, ranges, defaults, and rendered MiniSat flags.

B Appendix: Full Feature List

The 43 features extracted from each CNF, in the order returned by `FEATURE_NAMES` in `ml-config/data.py`:

1. `n_vars`: number of distinct variables.
2. `n_clauses`: number of clauses.
3. `literal_count`: total number of literal occurrences.
4. `clause_var_ratio`: $n_{\text{clauses}}/n_{\text{vars}}$.
5. `clause_len_mean`: mean clause length.
6. `clause_len_std`: standard deviation of clause length.

7. `binary_clause_frac`: fraction of length-2 clauses.
8. `ternary_clause_frac`: fraction of length-3 clauses.
9. `binary_count`: number of length-2 clauses.
10. `ternary_count`: number of length-3 clauses.
11. `unit_clause_frac`: fraction of unit clauses.
12. `long_clause_frac`: fraction of length ≥ 4 clauses.
13. `binary_to_long_ratio`: $\# \text{binary} / \# \text{long}$.
14. `horn_frac`: fraction of clauses with ≤ 1 positive literal.
15. `dual_horn_frac`: fraction with ≤ 1 negative literal.
16. `strict_horn_frac`: fraction with exactly 1 positive literal.
17. `var_degree_mean`: mean variable occurrence.
18. `var_degree_std`: standard deviation of variable occurrence.
19. `var_entropy`: Shannon entropy of the variable-occurrence distribution.
20. `var_freq_skew`: skew of variable occurrence frequencies.
21. `var_freq_kurtosis`: kurtosis (frequency outliers).
22. `var_freq_powerlaw_ratio`: median variable frequency.
23. `top10pct_var_degree_share`: contribution of the top-10% most-frequent variables.
24. `vcg_min_degree`: minimum variable-clause graph degree.
25. `vcg_max_degree`: maximum VCG degree.
26. `vcg_mean_degree`: mean VCG degree.
27. `clause_degree_min`: minimum clause length.
28. `clause_degree_max`: maximum clause length.
29. `clause_degree_mean`: mean clause length.

- 30. `clause_degree_entropy`: entropy of clause-length distribution.
- 31. `pos_neg_ratio`: global positive/negative literal ratio.
- 32. `pos_neg_ratio_mean`: mean per-variable pos/neg ratio.
- 33. `pos_neg_ratio_var`: variance of per-variable pos/neg ratio.
- 34. `polarity_balance_mean`: mean of $\min(p, n) / \max(p, n)$.
- 35. `polarity_balance_var`: variance of same.
- 36. `polarity_imbalance_mean`: mean of $1 - \min(p, n) / \max(p, n)$.
- 37. `polarity_imbalance_var`: variance of same.
- 38. `var_contrib_mean`: average variable contribution, $\sum_c 1/|c|$.
- 39. `var_contrib_std`: standard deviation of variable contributions.
- 40. `var_contrib_max`: maximum variable contribution.
- 41. `vig_mean_degree`: mean VIG degree (subsampled if needed).
- 42. `vig_var_degree`: variance of VIG degree.
- 43. `vig_max_degree`: maximum VIG degree.

C Appendix: Reproduction Recipe

To reproduce the system end-to-end:

1. Build MiniSat using the Makefile.
2. Create the Python environment and install dependencies.
3. Generate the portfolio (or a new one).
4. Run the data-collection sweep. This is resumable but will take several CPU-days of wall time. We recommend using an HPC cluster.

5. Train the model.
6. Solve a new CNF using the ML-picked configuration via `--use-ml`.
7. Run the end-to-end Python-level evaluation to verify the system.