Feature-Driven SAT Instance Generation

Benchmarking Model Counting Solvers Using Horn-Clause Variations

Vuk Jurišić[1]

Supervisor & Responsible Professor: Dr. Anna L. D. Latour

[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
January 26, 2025

Name of the student: Vuk Jurišić
Final project course: CSE3000 Research Project
Thesis committee: Dr. Anna L. D. Latour, Dr. Martin Skrodzki

Abstract

Model counting (#SAT) is a fundamental problem in theoretical computer science with applications in probabilistic reasoning, reliability analysis, and verification tasks. Despite advancements in solvers and #SAT instance generation, existing benchmarks fail to fully capture the diversity of structural features that influence solver performance. This paper introduces a feature-driven #SAT instance generator that systematically varies the fraction of Horn clauses across the full range (0% to 100%), enabling a rigorous evaluation of solver performance. Results reveal a "U-shaped" correlation between solve times and Horn-clause fractions and a strong relationship with model counts, exposing computational bottlenecks. Our contributions include the generator design, experimental validation across multiple solvers, and insights into improving solvers for challenging structural configurations, advancing #SAT research.

1 Introduction

Counting solutions to logical formulas is more than a theoretical curiosity—it underpins critical real-world applications. Model counting (MC), often referred to as #SAT, determines the number of satisfying assignments for a given propositional formula [Gomes *et al.*, 2009]. As a canonical #P-complete problem, it is central to theoretical computer science and vital for applications such as probabilistic reasoning [Littman *et al.*, 2001, Bacchus *et al.*, 2003], reliability analysis [Valiant, 1979], and verification [Baluta *et al.*, 2019]. A variety of model counting solvers have been developed to tackle #SAT, employing diverse strategies to address its computational challenges.

Solvers are algorithms designed to find the count of #SAT instances, while generators create these problem instances. Systematic evaluation and comparison of model counting solvers has been driven in part by the Model Counting Competition [Fichte and Hecher, 2021]. Although existing benchmark suites have advanced the field, our analysis of existing model counting generators reveals that they do not fully explore the structural diversity of #SAT instances. This shows the need for refined instance generators capable of systematically varying structural features to better understand their impact on solver performance. One such feature, the fraction of Horn clauses, will be the focus of our research.

The #SAT problem extends SAT by counting the total number of satisfying assignments, rather than just determining satisfiability. Instances are typically represented in Conjunctive Normal Form (CNF), a structure composed of clauses, each a disjunction of literals. A literal represents a variable or its negation. Structural features of #SAT instances include metrics such as clause-to-variable ratios, clause polarity, and the fraction of Horn clauses, all of which can influence solver performance. A Horn clause, in particular, is a special type of clause containing at most one positive literal.

This research addresses the question: *How can we design #SAT instance generators focusing on the fraction of Horn clauses to evaluate and benchmark model counting solvers, and how does the fraction of Horn clauses influence solver performance?* To explore this, we develop a feature-driven generator capable of producing similar #SAT instances with controlled Horn-clause fractions. Using these instances, we benchmark solvers from the 2024 Model Counting Competition under time constraints, gaining insights into solver efficiency and correctness across different structural configurations. Our analysis aims to provide valuable recommendations for improving solvers, particularly for challenging configurations.

The remainder of this paper is organized as follows. Section 2 reviews prior research that inspired and informed our work. Section 3 introduces foundational concepts of #SAT instances. Section 4 presents the problem statement and subquestions we aim to answer. Section 5 outlines our methodology for generating and evaluating feature-driven #SAT instances. Section 6 details the implementation of the generator. Section 7 discusses the experimental setup and results, while Section 8 addresses responsible research principles. Finally, Section 9 examines broader implications, and Section 10 concludes with directions for future work.

2 Related Work

This section reviews foundational and recent studies that informed and guided the development of our work. Horn clauses have been studied in both SAT and #SAT due to their unique structural properties and computational advantages. The Unit Propagation algorithm [Zhang and Stickel, 1996] solves instances of SAT problems composed entirely of Horn clauses in linear time. This approach simplifies satisfiability checking and laid the groundwork for modern SAT solvers. In model counting, [Dubray *et al.*, 2023] extended these ideas. They proposed an efficient projected weighted MC solver for Horn clause instances, demonstrating how their structured nature supports scalable probabilistic inference. These studies highlight the theoretical and practical significance of Horn clauses, motivating our exploration of their potential in model counting.

A notable contribution to this area is *SharpVelvet* [Latour and Soos, 2024], a modular tool for fuzzing propositional model counters. Building on the work of [Biere, 2009] and [Brummayer, 2009, Brummayer *et al.*, 2010], *SharpVelvet* generates #SAT instances using the CNFuzzDD and FuzzSAT generators. In this project, we have implemented one more generator, PairSAT [Jurišić, 2025b], and added it to SharpVelvet framework to help our research. However, our findings indicate that these instances lack structural diversity and proximity to the Horn formula. This limitation reinforces the need for a generator that systematically explores the fraction of Horn clauses.

Recent work has focused on designing benchmarks to challenge solvers by manipulating structural properties of instances. [Escamocher and O'Sullivan, 2022] proposed generators for small, yet difficult model counting instances. [Giráldez-Cru and Levy, 2016] introduced SAT instance generation techniques that preserve community structure. Although targeting SAT, this approach showed how structural modifications have been studied.

Research into structural features of #SAT instances has been pivotal in model counting field. Simple metrics, like clause-to-variable ratios, have proven insufficient to fully capture problem hardness. [Nudelman *et al.*, 2004] found that metrics such as weighted clause graph clustering coefficients are also useful indicators of instance difficulty. These findings have shaped automated algorithm selection frameworks, such as [Shavit and Hoos, 2024], which use diverse instance features to develop adaptive, portfolio-based solver strategies.

Building on these insights, benchmarks tailored to structural characteristics, such as varying Horn clause proportions, represent a promising research direction. We leverage post-processing techniques inspired by [Li *et al.*, 2023] to refine the fraction of Horn clauses in generated instances. Additionally, we integrate concepts from the generators proposed by [Escamocher and O'Sullivan, 2022] to target instances specifically for model counting and adopt their instance sizing strategies to guide our testing.

3 Preliminaries

This chapter introduces the foundational concepts necessary to understand this work, focusing on propositional model counting and related structures. Model counting (#SAT) is the problem of determining the number of satisfying assignments for a given propositional formula, typically expressed in Conjunctive Normal Form (CNF). A *solver* is an algorithm designed to find or count solutions for such #SAT instances, while a *generator* creates these problem instances to evaluate and benchmark solver performance. Structural properties of such formulas, such as the ratio between number of variables and clauses, are important for development #SAT generators.

3.1 CNF Formula

A *Conjunctive Normal Form (CNF)* formula is a conjunction of clauses, where each clause is a disjunction of literals. A *literal* is either a propositional variable or its negation.

For example, the formula $(A \vee \neg B) \wedge (C \vee D \vee \neg E) \wedge (\neg F)$ is in CNF. This formula consists of three clauses, each with a defined *arity*, which is the number of literals it contains. In this example, the arities of the clauses are 2, 3, and 1, respectively.

3.2 SATZilla Features

SATZilla is a framework designed to predict the best #SAT solver for a given instance by analyzing its structural features [Shavit and Hoos, 2024]. These features describe properties such as the size of the formula, graph connectivity, and clause-to-variable ratios. By using these metrics, SATZilla identifies the solver best suited for a specific instance, enhancing efficiency and performance. In this research, we used SATZilla's software to calculate feature values for #SAT instances.

3.3 Horn Clauses

Definition 1 (Horn Clause)**.** A *Horn clause* is a disjunction of literals in propositional logic that contains at most one positive literal. Formally, a clause $C = L_1 \vee L_2 \vee \cdots \vee L_n$ is a Horn clause if $|\{L_i \mid L_i \text{ is positive}\}| \leq 1$, where L_i represents a literal.

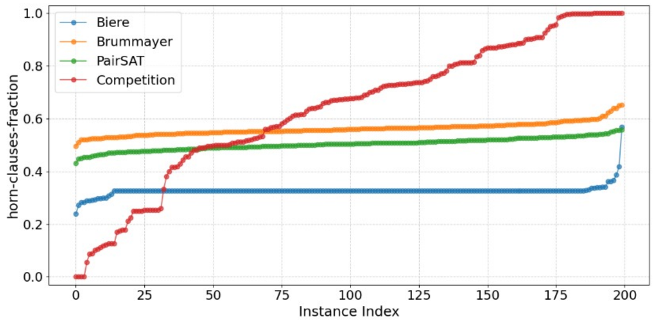

Figure 1: Horn-clause percentages generated by existing generators. The data is sorted per generator by the measured metric.

Definition 2 (Horn-Clause Fraction)**.** The *Horn-clause fraction* quantifies the proportion of clauses in a #SAT instance that are Horn clauses. If an instance has n total clauses and h of those are Horn clauses, the Horn-clause fraction is defined as:

$$\text{Horn-clause fraction} = \frac{h}{n}.$$

Horn clauses are significant due to their unique structural properties, which *can* influence solver efficiency and complexity. Analyzing the Horn-clause fraction helps in understanding solver behavior and instance difficulty [Shavit and Hoos, 2024].

4 Problem Description

Building on the concepts outlined in the preliminaries, this section describes the problem addressed in this research. Advancing solver performance is essential due to the broad applications of model counting. Achieving progress requires access to diverse #SAT instances that capture a wide range of structural properties, including the fraction of Horn clauses.

4.1 Problem Statement

We want to find out if the fraction of Horn clauses is influencing #SAT instance complexity and solver performance. However, data from 200 instances generated using existing tools available in *SharpVelvet* indicate that Horn-clause percentages are typically concentrated between 25% and 65% (see Figure 1). This limited range leaves much of the feature space unexplored, reducing the effectiveness of benchmarks in identifying solver limitations.

Additionally, Figure 1 includes, in red, a set of instances that do not conform to the pattern mentioned above. These instances are drawn from the publicly available Track 1 dataset of the Model Counting Competition [Fichte *et al.*, 2024] and originate from a variety of research groups rather than a single generative tool. Hence, designing a unified, publicly accessible generator capable of replicating such diversity in Horn-clause distributions remains an open challenge.

Motivated by the limited range of available generators and the recent interest in Horn clauses [Dubray *et al.*, 2023], this work aims to systematically vary the Horn-clause fraction across its full range (0% to 100%). Such variation should enable the creation of diverse and challenging #SAT instances that stress-test solvers. By doing so, we aim to identify solver inefficiencies, errors, and weaknesses.

4.2 Relation to Research Questions

Building on the research question introduced in the Section 1, this study is guided by the following research questions:

1. **RQ1:** How can we design a #SAT instance generator that systematically varies the fraction of Horn clauses while keeping values of other features stable?
2. **RQ2:** How can analysing solver performance on instances produced by our generator reveal solver strengths, weaknesses, and opportunities for improvement?

Addressing these questions involves testing solvers under diverse and extreme conditions, exposing vulnerabilities such as timeouts, memory inefficiencies, or incorrect model counts. This rigorous evaluation aims to identify areas for improvement in solvers.

5 Methodology

The primary objective of this research is to explore the full feature space of the "horn-clauses-fraction" feature while minimizing the impact on other key features of #SAT instances. While this goal is theoretically straightforward, in practice, certain features are tightly coupled. To address this, we developed an approach to systematically controls other features while varying the horn-clauses-fraction.

5.1 Feature Selection and Monitoring

To ensure meaningful and balanced #SAT instance generation, we selected 8 additional SATZilla features spanning diverse structural and statistical properties to monitor and stabilize during the generation process:

1. **vars-clauses-ratio**: The ratio of variables to clauses, representing problem size.
2. **VCG-VAR-mean**: The mean variable node degree in the Variable Clause Graph.
3. **VCG-CLAUSE-mean**: The mean clause node degree in the Variable Clause Graph.
4. **cluster-coeff-mean**: The mean weighted clustering coefficient in the Variable Clause Graph.
5. **reducedClauses**: The number of clauses remaining after preprocessing the SAT formula.
6. **reducedVars**: The number of variables remaining after preprocessing the SAT formula.
7. **BINARY+**: The fraction of clauses with 2 or more literals.
8. **TRINARY+**: The fraction of clauses with 3 or more literals.

Feature Name	**NCV Value**	
	CNFuzzDD	**Competition**
horn-clauses-fraction	0.06118	0.35889
BINARY+	0.23235	0.68741
VCG-VAR-mean	0.13415	0.22757
VCG-CLAUSE-mean	0.13410	0.23705
cluster-coeff-mean	0.08751	1.50337
vars-clauses-ratio	0.04495	0.50339
reducedClauses	0.00729	0.29252
reducedVars	0.00677	0.41309
TRINARY+	0.06654	0.36689

Table 1: Normalized Coefficient of Variation (NCV) values for selected features across 200 instances generated with CNFuzzDD generator and Track 1 of 2024 MC Competition.

These features were chosen from 56 calculated by SATZilla software. Many were excluded for being tightly coupled with horn-clause fraction. Some measured computation times, making them too difficult to stabilize. SATZilla also includes derived features like variance and higher-order values. These are helpful as data for machine learning but less relevant for our application. We manually analysed all features with help of a Pearson Correlation Matrix (found in section **??**) and picked the most meaningful ones. The final set is distinct, loosely coupled, and important for hardness.

Stabilizing these features ensures that horn-clause variations are isolated. This ensures that behaviour of `horn-clauses-fraction` values can be independently examined.

5.2 Challenge

Maintaining the eight selected features' values constant across many instances is inherently challenging. Interdependencies among features complicate varying the Horn-clause fraction from 0% to 100%. For instance, changing the horn-clauses-fraction directly influences the ratio of positive to negative literals, which in turn affects structural features like `cluster-coeff-mean`, `VCG-VAR-mean` and `VCG-CLAUSE-mean`. Additionally, features like `reducedVars` and `reducedClauses` are influenced by preprocessing heuristics that depend on the initial distribution of literals and clauses. Likewise, arity of clauses should also be kept same, and although not influenced by polarity of literals, is also structurally important. Arity affects `vars-clauses-ratio`, `BINARY+` and `TRINARY+`. This indicates that our instances should be constructed well so that preprocessing doesn't significantly simplify them.

5.3 Normalized Coefficient of Variation (NCV)

To evaluate the performance of the generators, we designed a metric called the *Normalized Coefficient of Variation (NCV)*. The NCV measures the variability of a feature across its theoretical range, adjusted by the coefficient of variation (CV). The CV is calculated as:

$$\text{Coefficient of Variation (CV)} = \frac{\sigma}{\mu},$$

where σ is the standard deviation of the feature values across generated instances, and μ is the mean value of the feature. This measures the relative dispersion of the feature.

The NCV is obtained by multiplying the CV with an adjustment factor that accounts for the observed range of the feature relative to its theoretical range. Specifically:

$$\text{Adjustment Factor} = \frac{\text{ObsMax} - \text{ObsMin}}{\text{TheorMax} - \text{TheorMin}},$$

where ObsMax and ObsMin are the maximum and minimum observed values for the feature across instances, and TheorMax and TheorMin are the theoretical maximum and minimum values.

To calculate the theoretical range, we conducted a thorough examination of all features and their calculation methods. Additionally, we analysed feature values produced by all generators seen in Figure 1. This analysis validated and supported our calculations.

The NCV is then calculated as:

$$\text{NCV} = \text{CV} \cdot \text{Adjustment Factor}.$$

This NCV metric captures both the variability of the feature and its dispersion across its theoretical range. High NCV values indicate greater variability and feature exploration, while low NCV values suggest limited variability. In this research the threshold of lower variability has been chosen as 0.1.

5.4 Visualization of Results

Table 1 summarizes the NCV values for selected features, providing insights into the variability achieved during instance generation. The values reflect the generator's ability vary feature of values across generated instances. Features with higher NCV values, such as `cluster-coeff-mean` in case of Competition instances, demonstrate effective exploration, whereas lower values indicate limited variability.

6 Implementation

In this section, we describe the design of our custom #SAT instance generator [Jurišić, 2025a]. The generator ensures controlled Horn-clause fractions while maintaining stability in other selected features. By adapting post-processing and solution-fitting techniques from prior works, it achieves both diversity and consistency, meeting the requirements outlined in Section 4.

6.1 Objective

The generator's primary objective is to produce #SAT instances with Horn-clause fractions ranging from 0% to 100%. We also aim to preserve stability in other selected features. Specifically, we targeted an NCV below 0.1 for all monitored features, which we estimated as sufficient to test Horn-clause fractions independently. This goal addresses the limitations of existing generators in *SharpVelvet*, which fail to explore the full Horn-clause fraction feature space.

Algorithm 1 Horn-Generator Pseudocode

Input: F - CNF instance with v variables and c clauses, n - amount of instance to generate
Output: Set of n CNF instances with varying horn clause counts

1 $step \leftarrow c/n$
2 $instances \leftarrow \emptyset$
3 **for** $i \leftarrow 0$ **to** 100 **do**
4     $F_{temp} \leftarrow F$
5     $target \leftarrow i \times step$
6     $count \leftarrow$ horn_clauses(F_{temp})
7   **if** $count < target$ **then**
8   **for** *clause in* F_{temp} **do**
9   **if** *clause is not Horn and count* $< target$ **then**
10   Flip positive literals to negatives
11   **end**
12   **end**
13   **end**
14   **if** $count > target$ **then**
15   **for** *clause in* F_{temp} **do**
16   **if** *clause is Horn and count* $> target$ **then**
17   Flip negative literals to positives;
18   **end**
19   **end**
20   **end**
21   **if** $rand() < 0.75$ **then**
22   Fit a solution to F_{temp} for satisfiability
23   **end**
24     $instances \leftarrow instances \cup F_{temp}$
25 **end**
26 **return** $instances$

6.2 Algorithm Design

To answer **RQ1**, the algorithm employs a post-processing technique inspired by [Crowley *et al.*, 2024,Giráldez-Cru and Levy, 2016]. It begins with an existing #SAT instance and modifies it to achieve the desired Horn-clause fraction while preserving other structural properties. The input instance can be any CNF formula, though instances without unit clauses (arity 1) are preferred since these clauses are always Horn. The generator has been validated using instances from CNFuzzDD, FuzzSAT, PairSAT, and G2SAT [You *et al.*, 2019].

The adjustment of the Horn-clause fraction is accomplished by flipping the polarity of literals, ensuring that other structural features remain stable, as described in Algorithm 1 on lines 10 and 17. On line 10, positive literals are flipped to negative to convert non-Horn clauses into Horn clauses, while on line 17, the opposite operation is performed for negative literals. The algorithm minimizes the number of flips to retain the structure of the original instance as much as possible. The choice of which literals to flip is random; however, to ensure reproducibility, the randomization process is tied to a seed.

In addition to adjusting the Horn-clause fraction, the algorithm incorporates a solution-fitting step, as shown on line 22 of Algorithm 1. The pseudocode for this step is provided in Section A.1. During this process, the algorithm traverses the clauses linearly and assigns an observed polarity to each

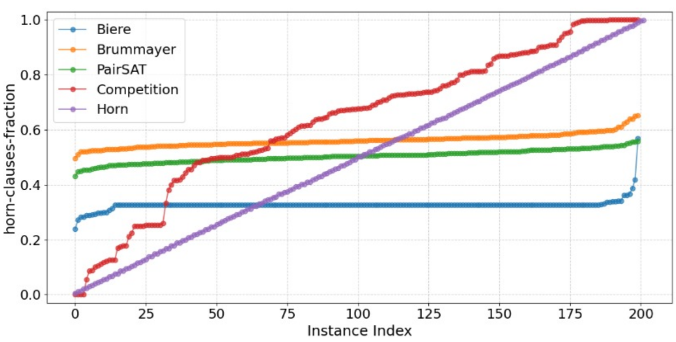

Figure 2: Horn-clause percentages generated by existing generators with our generator in purple.

literal, thereby creating a set of assignments, a solution. If any clause is not satisfied by the generated solution, the literals within that clause are flipped while preserving the overall Horn-clause fraction. This step is applied with a 75% probability to ensure that a majority of the instances remain satisfiable, while leaving some instances unsatisfiable. Unsatisfiable instances are important for benchmarking solvers. The concept of solution fitting is adapted from [Escamocher and O'Sullivan, 2022].

7 Experiments and Results

This section details the experimental evaluation of our implementation and presents the results obtained. The primary objective of this study is to address **RQ2**, investigating how variations in the Horn-clause fraction influence the performance of model counting solvers. The findings demonstrate the effectiveness of our generator in producing instances that challenge state-of-the-art solvers while maintaining control over structural features.

7.1 Implementation Results

A total of 200 instances were generated using our custom Horn generator to evaluate its ability to vary the Horn-clause fraction compared to existing generators. Figure 2 illustrates the generator's capability to produce a uniformly distributed range of Horn-clause fractions, showcasing its effectiveness in addressing the limitations of current tools.

To assess the stability of other features while varying the Horn-clause fraction, we created an additional 1010 instances. The generation was performed using 10 base instances, selected as follows: 3 from the FuzzSAT generator, 3 from PairSAT, 3 from CNFuzzDD, and 1 from G2SAT. For each base instance, 101 variants were generated using the Horn generator. The number 101 comes from each of the percentages of Horn-clause fraction value from 0% to 100%. Table 2 presents the Normalized Coefficient of Variation (NCV) values for these features, confirming that the generator achieves high variability in the Horn-clause fraction while ensuring minimal deviation in other structural features. These results validate the generator's ability to produce diverse instances with controlled feature stability, answering the objective in **RQ1**.

Feature Name	**NCV Value**
horn-clauses-fraction	0.570534
cluster-coeff-mean	0.011602
vars-clauses-ratio	0.000648
reducedVars	0.000259
reducedClauses	0.000098
VCG-VAR-mean	0.000015
VCG-CLAUSE-mean	0.000015
BINARY+	0.000014
TRINARY+	0.000000

Table 2: Normalized Coefficient of Variation (NCV) values for selected features across 1000 #SAT instances generated with Horn generator. Feature with highest value is highlighted.

7.2 Experimental Setup

All experiments were conducted on TU Delft's HPC cluster [Delft High Performance Computing Centre (DHPC), 2024] using the p2 cluster for its higher CPU frequency, needed for #SAT tasks. Each task was allocated one core and 8 GB of memory, this being the memory limit per solver. Tasks ran in parallel, with solvers operating independently without sharing memory.

The solvers tested were `d4` [Lagniez and Marquis, 2017], `ganak` [Sharma *et al.*, 2019], and `gpmc` [Hashimoto, 2023]. Identical hardware and instances were used to ensure fair performance comparisons. Each solver had 10 minutes to solve an instance. Instances not *Solved* within this limit were classified as *Unsolved*.

7.3 Evaluation Metrics

To measure solver performance, we used solving time. We also tracked key features for consistency. We ensured NCV remained below 0.1. We checked that after each generation. If NCV exceeded 0.1, we stopped. Then we refined our generator. This has however not happened once during the testing process.

7.4 Results

We measured solving time on each generated instance, with *SharpVelvet* enforcing a 10-minute time limit and 8 GB of memory per solver. This memory limit is measured solvers recorded their own memory usage however it is further ensured by the *DelftBlue*. For reliability checks, we compared solver outputs across repeated runs of identical instances.

Three large experiments were conducted using 3-CNF instances, a common focus in #SAT research. Each experiment comprised 1010 instances, aiming to solve approximately 75% within the prescribed limit. 10 base instances were used with horn generator creating 101 instances from each with 0% to 100% of horn clauses. We fixed the variable count at 400 and varied the number of clauses (90, 100, 110), guided by earlier observations that problems near a 4:1 clause-to-variable ratio pose particular difficulty [Nudelman

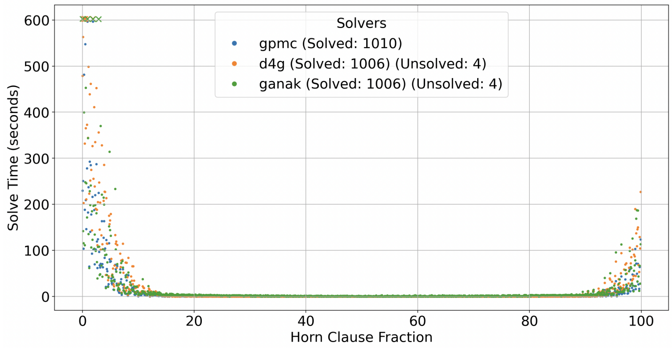

Figure 3: Solver performance on 3-CNF instances with 400 clauses and 90 variables.

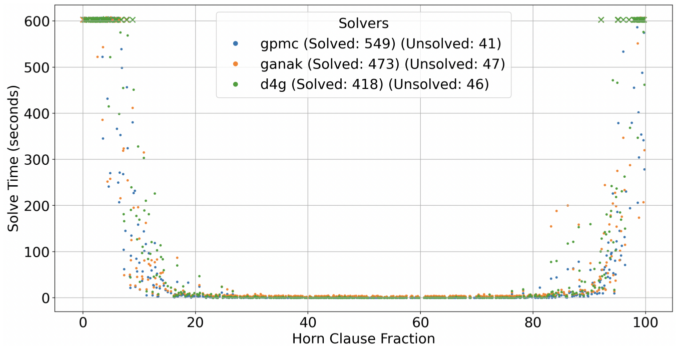

Figure 4: Solver performance on 3-CNF instances with 400 clauses and 100 variables.

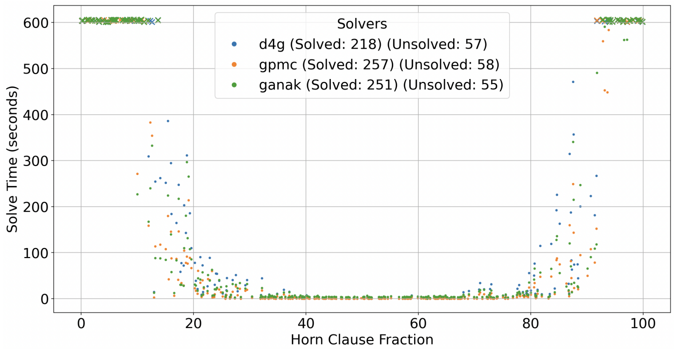

Figure 5: Solver performance on 3-CNF instances with 400 clauses and 110 variables.

et al., 2004, Escamocher and O'Sullivan, 2022]. Prior experimentation with DelftBlue hardware found that instances of about 400 clauses are suitable for our 10 minute timeout. Base instances were generated via PairSAT to ensure consistent control over arity and size.

Figure 3 presents solver performance on instances with 90 clauses over a 12-hour horizon. All instances were eventually solved, though *ganak* and *d4* timed out on four instances (fewer than 5% of Horn clauses) prior to the 10-minute cutoff. Figures 4 and 5 show results for 100 and 110 clauses, respectively, confirming a "U-shaped" performance curve tied

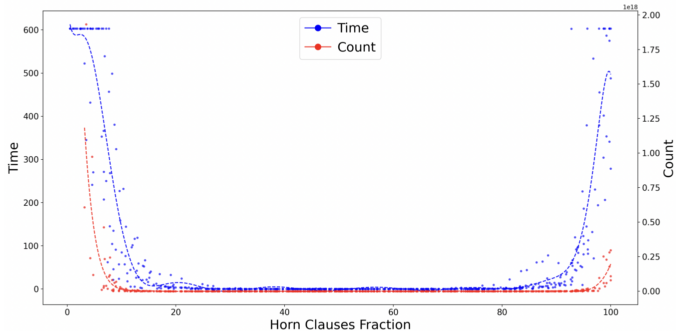

Figure 6: Correlation between *gpmc* solver runtime and model count across instances with varying Horn-clause fractions.

to varying Horn-clause fractions. For 100 clauses, about 9% of instances timed out, with *gpmc* showing stronger performance than *d4*. Increasing to 110 clauses led to a 23.5% timeout rate, aligning with the intended 75% solve threshold. Instances with Horn-clause fractions below 15% or above 90% were particularly challenging. Across these conditions, *ganak* narrowly outperformed *gpmc* in terms of successful solves, with timeouts at 21.9% and 22.5%, respectively.

7.5 Result Analysis

A notable observation from the experiments is a pronounced "U-shaped" solve time curve, where instances with either a very low or very high Horn-clause fraction require more time to solve. Initial efforts to correlate this phenomenon with other structural features were inconclusive, suggesting that the Horn-clause fraction itself might be driving the observed difficulty. Instances at the extremes of Horn-clause fraction exhibit similar polarity assignments for their variables, resulting in a large number of solutions (high model count). This lead us to investigate the relationship between solve time and model count.

To quantify this relationship, we plotted the model count against solver runtime for the dataset with 400 variables and 100 clauses, as shown in Figure 6. The Spearman rank correlation coefficient between model count and solving time was 0.972, indicating a strong monotonic association. Since model counts may span several orders of magnitude, we suspected an exponential relationship between model count and solving time. This initial analysis suggested that model count plays a critical role in influencing solver performance, particularly for instances at extreme Horn-clause fractions.

After testing various transformations, we found the fourth-root transformation (i.e., $\sqrt[4]{\text{model count}}$) to exhibit the highest Pearson correlation coefficient of 0.862 (Figure 7). This transformation improved linearity and demonstrated how the model count tightly predicts solver runtime. A similar pattern emerged in the 90-clause dataset, for which we measured a Spearman coefficient of 0.918 and a Pearson correlation of 0.842 after applying a cubic-root transformation. These findings underscore that extreme Horn-clause fractions yield large solution spaces, thereby correlating strongly with the heightened computational cost of model counting.

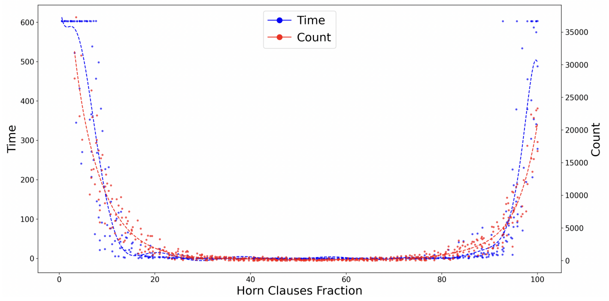

Figure 7: Correlation between *gpmc* solver runtime and model count across instances with varying Horn-clause fractions with transformation function $f(x) = \sqrt[4]{x}$ applied to model count.

8 Responsible Research

Research integrity is paramount in ensuring the credibility and reproducibility of scientific work. In alignment with the "Netherlands Code of Conduct for Research Integrity" [Netherlands Organisation for Scientific Research (NWO), 2024], we adhered to the principles of honesty, scrupulousness, transparency, independence, and responsibility throughout this study.

To ensure honesty, all reported results were obtained without manipulation or bias. The experimental setup, methodologies, and metrics used for evaluation are explicitly documented to facilitate transparency. Where applicable, state-of-the-art approaches were incorporated, and the latest advancements in #SAT instance generation and model counting solvers were considered.

To promote reproducibility, the implementation of our custom #SAT instance generator, will be made publicly available on a GitHub repository [Jurišić, 2025a]. Additionally, testing instances and evaluation scripts will be shared to enable the community to validate and extend our findings. Following recommendations for open research practices [Foster and Deardorff, 2017], we also ensured that no proprietary or personal data was used during this research.

By adhering to these principles, we aim to contribute to the advancement of reproducible and responsible research in the field of model counting.

9 Discussion

In this section, we analyze the unique contributions of our Horn-driven generator, focusing on its ability to expand the range of benchmarks, the difficulty of generated instances, solver performance, and limitations.

Our generator distinguishes itself from tools like FuzzSAT and CNFuzzDD by covering a broader range of Horn-clause fractions (0% to 100%). Unlike existing generators, which are constrained to a narrower spectrum, this range enables more diverse and rigorous benchmarking of solvers. Notably, instances generated by our approach demonstrate significantly higher complexity, as indicated by longer solve times, affirming the generator's capacity to stress-test solvers effectively.

Generated instances posed challenges beyond those of traditional benchmarks, particularly at extreme Horn-clause fractions. The observed "U-shaped" performance curve reflects the computational challenge imposed by extreme configurations, likely due to the high model count. This highlights the value of integrating such instances into benchmarking suites to evaluate solvers thoroughly.

A minor yet noteworthy observation was the occurrence of discrepancies in model counts across solvers for less than 1% of cases. These differences, often by a single count, appear linked to very high model counts, exceeding integer limits. While non-reproducible locally, these anomalies show a possible need for enhanced precision handling in solvers for extreme scenarios. This is however, subject to future research.

Solver behaviour varied considerably across the generated instances. `d4` struggled with extreme Horn-clause fractions, particularly at the higher end, while `ganak` and `gpmc` demonstrated better resilience. However, the consistent difficulty at extremes for all solvers suggests a potential area for algorithmic improvements, such as strategies for handling expansive solution spaces more efficiently.

10 Conclusions and Future Work

This research introduced a feature-driven generator for #SAT instances, capable of varying the Horn-clause fraction systematically across the full range (0% to 100%), directly addressing **RQ1**. By maintaining stability in key structural features, the generator enables precise evaluation of solver performance under varying Horn-clause configurations, filling gaps in existing tools.

Several limitations remain. The generator can't work with unit clauses in base instances, as these restrict the achievable Horn-clause fractions. Future enhancements should address this constraint and refine heuristic methods to further stabilize features. Additionally, the minor model count discrepancies observed in less than 1% of cases, likely due to numerical limitations in handling extremely high counts, require in-depth investigation.

Our results show that the generator effectively reveals solver limitations, with performance following a "U-shaped" curve tied to extreme Horn-clause fractions. This insight, addressing **RQ2**, highlights the computational challenges posed by large solution counts, found at extreme Horn-clause fractions.

For solver developers, our findings suggest focusing on optimizing algorithms for high model count instances. For example, strategies to efficiently manage large solution spaces could improve solver performance. While solvers like `ganak`, `gpmc`, and `d4` exhibited similar performance on most instances, `d4` struggled on instances with extreme values of Horn clause fraction, further emphasizing need for improvement.

The promising correlation between solve times and model counts suggests a pattern we didn't find in literature before, though the derived transformation function requires validation on larger datasets. Future work should expand on this analysis, diving deeper into relation between solve time and model count.

A Appendix

A.1 Solution Fitting Algorithm

Algorithm 2 provides the pseudocode for the solution-fitting step referenced in line 21 of Algorithm 1. This step adjusts the polarity of literals within clauses to ensure the generated instances remain satisfiable while aiming to match the target Horn clause count. The process involves flipping literals strategically, prioritizing minimal disruption to the original formula's structure. This step is critical for maintaining feature stability while systematically varying the Horn clause fraction.

Algorithm 2 Pseudocode for the Solution Fitting Algorithm.

```
Input: F – CNF Formula, target – target Horn count
Output: A modified CNF formula F that is satisfiable.
solution ← generate_solution(F)
current ← count_horn_clauses(F)
foreach clause c in F do
    if not satisfied(c, solution) then
        P ← {all positive literals in c}
        N ← {all negative literals in c}
        current ← {Count of horn clauses}
        if current ≤ target then
            if |P| > 1 then
                Flip literals in P if negative in solution
                if satisfied(c, solution) and c is Horn then
                    current ← current + 1;
                end
            else
                Flip literals in P or N so that c is satisfied
                if satisfied(c, solution) and c not Horn then
                    current ← current − 1;
                end
            end
        else
            if |P| > 1 then
                Flip literals in N if positive in solution
                if satisfied(c, solution) and c is Horn then
                    current ← current + 1;
                end
            else
                Flip literals in N if positive in solution
                if satisfied(c, solution) and c is no longer
                  Horn then
                    current ← current − 1;
                end
            end
        end
    end
end
```

References

[Bacchus *et al.*, 2003] F. Bacchus, S. Dalmao, and T. Pitassi. Algorithms and complexity results for #SAT and Bayesian inference. In *44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings.*, pages 340–351, October 2003. ISSN: 0272-5428.

[Baluta *et al.*, 2019] Teodora Baluta, Shiqi Shen, Shweta Shinde, Kuldeep S. Meel, and Prateek Saxena. Quantitative Verification of Neural Networks and Its Security Applications. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 1249–1264, New York, NY, USA, November 2019. Association for Computing Machinery.

[Biere, 2009] Armin Biere. Cnfuzzdd: A CNF instance generator. `https://fmv.jku.at/cnfuzzdd/`, 2009. Adapted in SharpVelvet as generators/cnf-fuzz-biere.c.

[Brummayer *et al.*, 2010] Robert Brummayer, Florian Lonsing, and Armin Biere. Automated Testing and Debugging of SAT and QBF Solvers. In Ofer Strichman and Stefan Szeider, editors, *Theory and Applications of Satisfiability Testing – SAT 2010*, pages 44–57, Berlin, Heidelberg, 2010. Springer.

[Brummayer, 2009] Robert Brummayer. Fuzzsat: A CNF instance generator. `https://fmv.jku.at/fuzzsat/`, 2009. Adapted in SharpVelvet as generators/cnf-fuzz-brummayer.py.

[Crowley *et al.*, 2024] Daniel Crowley, Marco Dalla, Barry O'Sullivan, and Andrea Visentin. SAT Instances Generation Using Graph Variational Autoencoders. 2024.

[Delft High Performance Computing Centre (DHPC), 2024] Delft High Performance Computing Centre (DHPC). DelftBlue Supercomputer (Phase 2). `https://www.tudelft.nl/dhpc/ark:/44463/DelftBluePhase2`, 2024.

[Dubray *et al.*, 2023] Alexandre Dubray, Pierre Schaus, and Siegfried Nijssen. Probabilistic Inference by Projected Weighted Model Counting on Horn Clauses. In Roland H. C. Yap, editor, *29th International Conference on Principles and Practice of Constraint Programming (CP 2023)*, volume 280 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 15:1–15:17, Dagstuhl, Germany, 2023. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. ISSN: 1868-8969.

[Escamocher and O'Sullivan, 2022] Guillaume Escamocher and Barry O'Sullivan. Generation and Prediction of Difficult Model Counting Instances, December 2022. arXiv:2212.02893.

[Fichte and Hecher, 2021] Johannes K. Fichte and Markus Hecher. Model Counting Competition 2021: Call for Benchmarks/Participation. Technical report, February 2021.

[Fichte *et al.*, 2024] Johannes Fichte, Markus Hecher, and Arijit Shaw. Model Counting Competition 2024: Competition Instances, November 2024.

[Foster and Deardorff, 2017] Erin D. Foster and Ariel Deardorff. Open Science Framework (OSF). *Journal of the Medical Library Association : JMLA*, 105(2):203–206, April 2017.

[Giráldez-Cru and Levy, 2016] Jesús Giráldez-Cru and Jordi Levy. Generating SAT instances with community structure. *Artificial Intelligence*, 238:119–134, September 2016.

[Gomes *et al.*, 2009] Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Model Counting. In *Handbook of Satisfiability*, pages 633–654. IOS Press, 2009.

[Hashimoto, 2023] Kenji Hashimoto. GPMC, 2023.

[Jurišić, 2025a] Vuk Jurišić. HornSAT. `https://github.com/Chevuu/HornSAT/`, 2025.

[Jurišić, 2025b] Vuk Jurišić. PairSAT. `https://github.com/Chevuu/PairSAT/`, 2025.

[Lagniez and Marquis, 2017] Jean-Marie Lagniez and Pierre Marquis. An Improved Decision-DNNF Compiler. pages 667–673, 2017.

[Latour and Soos, 2024] Anna L.D. Latour and Mate Soos. Sharpvelvet, 2024.

[Li *et al.*, 2023] Yang Li, Xinyan Chen, Wenxuan Guo, Xijun Li, Wanqian Luo, Junhua Huang, Hui-Ling Zhen, Mingxuan Yuan, and Junchi Yan. HardSATGEN: Understanding the Difficulty of Hard SAT Formula Generation and A Strong Structure-Hardness-Aware Baseline. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, KDD '23, pages 4414–4425, New York, NY, USA, August 2023. Association for Computing Machinery.

[Littman *et al.*, 2001] Michael L. Littman, Stephen M. Majercik, and Toniann Pitassi. Stochastic Boolean Satisfiability. *Journal of Automated Reasoning*, 27(3):251–296, October 2001.

[Netherlands Organisation for Scientific Research (NWO), 2024] Netherlands Organisation for Scientific Research (NWO). Netherlands Code of Conduct for Research Integrity | NWO, 2024.

[Nudelman *et al.*, 2004] Eugene Nudelman, Kevin Leyton-Brown, Holger H. Hoos, Alex Devkar, and Yoav Shoham. Understanding Random SAT: Beyond the Clauses-to-Variables Ratio. In Mark Wallace, editor, *Principles and Practice of Constraint Programming – CP 2004*, pages 438–452, Berlin, Heidelberg, 2004. Springer.

[Sharma *et al.*, 2019] Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S. Meel. GANAK: A Scalable Probabilistic Exact Model Counter. pages 1169–1176, 2019.

[Shavit and Hoos, 2024] Hadar Shavit and Holger H. Hoos. Revisiting SATZilla Features in 2024. In *27th International Conference on Theory and Applications of Satisfiability Testing (SAT 2024)*, pages 27:1–27:26. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2024.

[Valiant, 1979] Leslie G. Valiant. The Complexity of Enumeration and Reliability Problems. *SIAM Journal on Computing*, 8(3):410–421, August 1979. Publisher: Society for Industrial and Applied Mathematics.

[You *et al.*, 2019] Jiaxuan You, Haoze Wu, Clark Barrett, Raghuram Ramanujan, and Jure Leskovec. G2SAT: Learning to Generate SAT Formulas, October 2019. arXiv:1910.13445.

[Zhang and Stickel, 1996] Hnatao Zhang and Mark E. Stickel. An Efficient Algorithm for Unit Propagation, 1996.