Breaking Weighted Model Counting Solvers Using EXTREMEgen

Generating WMC instances for fuzzing

Bram Snelten[1]

Supervisor: 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: Bram Snelten, TU Delft Final project course:
CSE3000 Research Project
Thesis committee: Anna L.D. Latour, Examiner

Abstract

Weighted model counting (WMC) solvers play a key role in Bayesian inference applications, used for medical diagnosis [17] [16] and risk assessment [14]. Ongoing efforts to improve WMC solver developers aim to develop a fuzzer to identify bugs. This research is aimed at enhancing the quality of this fuzzer by developing an additional method to generate WMC instances, EXTREMEgen [21]. The functionality of this new approach relies on generating practical instances from Bayesian networks and breaking the solvers using extreme weights. Our empirical experiments show that EXTREMEgen exposes bugs in state-of-the-art WMC solvers. The generated instances are solved fast enough to be usable in fuzzing. However, generation speed needs to be optimized to become practical for fuzzing. When optimized, EXTREMEgen could become the first generator of WMC instances specifically designed for fuzzing.

1 Introduction

In domains like medical diagnosis systems and risk modeling, a subtle bug in a Weighted Model Counting (WMC) solver could lead to catastrophic miscalculations, jeopardizing health or safety! On top of that, tools to systematically expose bugs in WMC solvers do not exist yet. Critical systems depend on reliable WMC solvers, which tackle an extension of the SAT problem: determining whether a Boolean formula has at least one solution. Model counting (MC), the counting variant of SAT, goes further by asking how many unique solutions (models) exist for the formula. WMC then maps numerical weights to solutions, requiring solvers to compute the weighted sum of all solutions.

WMC solving is a state-of-the-art technology used in probabilistic reasoning [12], [13], [5], [18]. Which in turn is used in various practical fields such as medical diagnosis [17], [16], [20] and risk modeling [14].

Recently the yearly model counting competition [10] started streamlining the development of solvers. This initiative encouraged the development of a fuzzer to assist solver development, SharpVelvet [2]. A fuzzer [25] automatically generates problem instances, trying to find instances that trigger bugs in a model counter. The more diverse the problem instances are, the more likely it is that one of them will trigger a bug. SharpVelvet already contains two generators that can generate MC instances, CNFuzz and FuzzSAT [4]. SharpVelvet has limited functionality to convert these MC into WMC instances.

This research aims to enhance the quality of SharpVelvet by developing a WMC instance generator named EXTREMEgen that exposes bugs in state-of-the-art WMC solvers. These generated instances need to be solved very fast to enable high-throughput fuzzing. Lastly, generating instances should be much faster than solving them. Slow generation also limits the fuzzing throughput. Therefore this research answers the following questions:

RQ1: Does EXTREMEgen expose bugs in state-of-the-art WMC solvers?

RQ2: Does EXTREMEgen generate instances that can be solved fast enough to be usable in fuzzing?

RQ3: Does EXTREMEgen generate instances fast enough to be usable in fuzzing?

EXTREMEgen is designed to expose bugs related to precision, rounding, and handling of extreme numerical values. The generation process starts with generating Bayesian networks. After which extreme numerical weights are added to the variables of the WMC instance. EXTREMEgen successfully exposes bugs related to handling of extreme weights and determining satisfiability. The generated instances are solved fast enough for a high throughput. Fuzzing with EXTREMEgen is bottlenecked by generation speed. Generation speed needs to be optimized to achieve a high throughput.

The remainder of this paper is organized as follows. We briefly discuss notation and provide relevant definitions in Section 2. Related work is highlighted in Section 3. Section 4 outlines our approach for generating WMC instances. Section 5 describes the experimental evaluation of the generation method using SharpVelvet and state-of-the-art WMC solvers. We conclude and discuss future research in Section 6, after which the paper ends with a reflection on the responsibility of this research in Section 7.

2 Preliminaries

We briefly present the concepts of CNF formulas, model counting, and weighted model counting. We also introduce the Model Counting Competition data format and highlight the underlying floating-point format.

2.1 CNF and Model Counting

Consider Boolean formula 1 where a and b are Boolean variables.

$$f = (a \vee \neg b) \wedge (\neg a \vee b) \tag{1}$$

Boolean formulas are composed of literals and clauses. A literal is a Boolean variable or its negation (e.g., a, $\neg b$), a clause is a disjunction of one or more literals (e.g., $a \vee \neg b$).

A formula is said to be in Conjunctive Normal Form (CNF) if it is a conjunction of one or more clauses. For example, Eq. 1 is in CNF because it is a conjunction of two clauses. Since f has 2 variables, there are a total of 4 possible assignments to a and b:

$$a = \text{false},\ b = \text{false} \tag{2}$$

$$a = \text{false},\ b = \text{true} \tag{3}$$

$$a = \text{true},\ b = \text{false} \tag{4}$$

$$a = \text{true},\ b = \text{true} \tag{5}$$

Of these, assignments 2 and 5 are solutions, i.e., assignments such that f is true.

2.2 Weighted Model Counting

We next illustrate the Weighted Model Counting (WMC) problem, which extends the model counting problem by mapping numerical weights to solutions. The task is to compute the sum of the weights of all solutions of the formula.

Consider again the CNF formula 1. We convert it into a WCNF instance by assigning the following weights to the variables:

$$w(a = \text{true}) = 0.7, \quad w(a = \text{false}) = 0.3,$$

$$w(b = \text{true}) = 0.6, \quad w(b = \text{false}) = 0.4.$$

The weighted model count is computed by first calculating the weight of each solution, 2 and 5, and then summing them:

$$w(a = \text{false}) \cdot w(b = \text{false}) = 0.3 \cdot 0.4 = 0.12.$$

$$w(a = \text{true}) \cdot w(b = \text{true}) = 0.7 \cdot 0.6 = 0.42.$$

$$\text{WMC}(f) = 0.42 + 0.12 = 0.54.$$

2.3 Input Format

The WMC solver SharpVelvet aims to improve upon the performance of existing solvers that follow the Model Counting Competition Data Format [11]. The format specifies the following about weights: "The weight will be given as floating point (e.g., 0.0003) with at most 9 significant digits after the decimal point, or in 32-bit scientific floating point notation (e.g., 1.23e+4), or as fraction (e.g., 3/10) consisting of two integers separated by the symbol / [11]." In this paper we make the assumption that the competition format enforces Single-Precision floating-point format by IEEE 754 standards [1], because this format usually occupies 32 bits in computer memory. This is the same amount of bits mentioned in the competition format. Applying the same reasoning, we assume integers used in the fractional weight notation are enforced to be 32-bit integers. Solvers need to be able to handle negative weights. Therefore we assume the 32-bit integers are signed.

Single-precision floating-point format

The IEEE 754 standard specifies a single-precision floating point as having: 1 sign bit, 8 exponent bits, and 23 fraction bits. The value is calculated using the formula:

$$(-1)^{\text{sign}} \times 2^{\text{exponent}-127} \times (1.\text{fraction})$$

where:

- **sign** is the sign bit (0 for positive, 1 for negative).
- **exponent** is the unsigned integer value of the 8-bit exponent field.
- **fraction** is the binary value represented by the 23 fraction bits, interpreted as the fractional part of a binary number (e.g., 101 would mean 0.101 in binary).

Special cases include the exponent being all ones, representing $\pm\infty$ if the fraction is zero or NaN if the fraction is nonzero.

3 Related Work

EXTREMEgen is the first WMC fuzzing instance generator designed specifically to expose bugs related to the handling of weights. Unlike previous generators, which focus on generating instances for benchmarking or different model counting variants.

Initial testing of model counters using fuzzing was started by Robert Brummayer, Florian Lonsing, and Armin Biere [4]. They developed three methods of generating instances. Firstly, 3SATGen generates traditionally hard instances based on the SAT phase transition [24]. Secondly, CNFuzz was the first attempt to generate structured instances. Lastly, FuzzSAT improved upon CNFuzz by generating instances with an explicit structure: directed acyclic graphs. SharpVelvet implemented CNFuzz and FuzzSAT, while extending their functionality by making it possible to transform CNFuzz and FuzzSAT instances into WMC instances by adding weights to their variables. These weighted CNFuzz and FuzzSAT instances serve as baseline generators when evaluating EXTREMEgen.

Usman et al. built upon the generators from Biere and Brummayer to create their own test suite, TestMC [23]. Their method of classifying bug types is also used in this paper. TestMC introduces the Bounded Exhaustive generation technique on the projected model counting variant. Although their generator works well, it is not applicable to the weighted model counting variant.

Dilkas recently developed a generator for benchmarking WMC instances [7]. However, benchmark instances are unusable for fuzzing because their solve time is very long. Eschamocher et al. developed another generator for benchmarking instances [8], with the requirement that the difficulty of the instances should arise from model counting rather than satisfiability. This is interesting for our generator because this requirement would ensure that each instance targets the intended model counting functionality rather than the satisfiability functionality. But their approach is not compatible with our generator because they inject solutions which would destroy the network structure EXTREMEgen depends upon.

4 Approach

In this section, we present the approach EXTREMEgen uses to generate WMC instances for fuzzing state-of-the-art WMC solvers. As stated in the introduction, EXTREMEgen is designed with the following requirements in mind:

1 Generated instances should expose bugs in state-of-the-art WMC solvers.

2 Generated instances should not take long to solve.

3 Generating instances should not take long.

Our method follows three steps when generating instances. First, generating a Bayesian network. The second step is encoding the generated Bayesian network into a CNF instance [3]. The last step is converting the CNF to a WCNF instance by adding extreme weights to variables. Each generated instance is expected to be diverse due to the combination of random network structures, variable node counts, and the random assignment of extreme weights.

4.1 Generating Bayesian Networks

WMC is commonly performed on Bayesian networks to solve probabilistic inference tasks [12], [13], [5], [18]. A concrete example of a probabilistic inference task using a Bayesian network is medical diagnosis. [17], [16], [20]. We use

Bayesian networks to mimic these realistic instances. If an instance would expose a bug, the bug would be likely to occur in a realistic scenario. This makes it extra valuable to expose bugs. Additionally, having a structure at all should introduce a base difficulty for solvers, therefore preventing trivial instances from being generated.

Bayesian networks are directed acyclic graphs. We briefly explain 3 different Bayesian network types that are commonly used in practice:

DQMR [18] These instances are CNF encodings of Bayesian networks that simulate diagnostic networks [20]. They are two-layer bipartite networks, where the nodes in the first layer represent diseases and the nodes in the second layer represent symptoms. Each symptom is randomly linked to 4 diseases. The number of diseases and symptoms vary between 50 and 100.

GRID [18] These networks are a square grid of size $N \times N$, where each node $X_{i,j}$ for $1 \leq i, j \leq N$ has an incoming edge from $X_{i-1,j}$ and $X_{i,j-1}$, as long as the indices are greater than zero.

TREE To complement the previous existing Bayesian network implementation by Sang et al., we implemented tree-structured Bayesian networks. Bayesian networks are common in risk assessment strategies, such as Fault Tree Analysis [26]. Expanding the range of structures generated by EXTREMEgen with trees enhances diversity, which should improve exposing bugs, while maintaining relevance to practical applications. Additionally, due to their simplicity, tree structures are expected to be easy to solve.

When generating TREE instances the user specifies the number of nodes and the maximum number of children per node. The resulting Bayesian network forms a random tree structure according to previous parameters.

For each of the network types, the amount of nodes can be varied, which should impact generation time and solving time. This should make it possible for us to tune the generator to fulfill requirements 2 and 3.

4.2 Extreme Weights

After encoding the generated network into CNF [3], we convert the instance to WCNF by adding weights to each variable. This subsection describes how the extreme weights for EXTREMEgen were chosen. And how the weights are assigned.

In this context, "extreme weights" refer to the largest allowed finite positive and negative weights, as well as the smallest weights. These extreme values are intended to trigger overflow or precision-related errors during weight multiplication or summation. Overflow errors are expected to occur when very large values exceed the representable range, resulting in the value being rounded to infinity. Precision loss bugs should arise when large numbers approach the limits of floating-point precision, leading to discrepancies in the results. Furthermore, multiplying small weights can also induce precision loss, as values close to zero could be unjustly rounded to zero. This produces unexpected outcomes when multiplying or dividing.

Recall the 3 weight notations in Subsection 2.3. EXTREMEgen operates with 2 sets of extreme weights. Set 1 contains weights that are in the domain [0,1], mimicking realistic probabilities, attempting to cause precision errors. Set 2 contains weights confined to the competition data format.

Set 1 contains extreme weights $\in$ [0,1], testing realistic probabilities. For the floating point formats, these extremes are:

$$1.000000000, 0.000000000, 0.999999999, 0.000000001$$

And their scientific notation counterparts. The maximum signed 32-bit integer is 2147483647. Making the fractional notation extremes:

$$2147483647/2147483647, \quad 0/2147483647$$
$$2147483646/2147483647, \quad 1/2147483647$$

Set 2 additionally contains the largest allowed finite positive and negative weights, attempting to cause overflow errors. In floating point representation, these maximum weights are achieved with exponents of 254. In scientific notation this is noted as $\pm$3.402823466e+38. The boundary weights for the fractional weight notation are $\pm$2147483647/1.

Lastly, each set is complemented by fractional weights made up of negative integers. The weights in this group were created by creating every combination of positive/negative numerator and denominator for the fractional weight already introduced. The created weight is added to Set 2 only if the weight $\in$ [0,1]. The full list of weights can be found in the implementation [21].

When generating with Set 1, for each literal l, both w_l and $w_{\neg l}$ get assigned a random weight uniformly from Set 1. When generating using Set 2, each literal also gets assigned a random weight uniformly from Set 2 while following $w_l + w_{\neg l} = 1$. The strategy in assigning weights, in this case randomly, should impact the amount of bugs found positively. The primary goal of this strategy is to explore the full range of extreme values. By generating many different networks with numerous nodes, this strategy covers a wide spectrum of possible weight configurations. This should lead to a higher amount of exposed bugs. The simplicity of the random weight assignment strategy also contributes to the efficiency of instance generation.

5 Experiments

This section is aimed at answering the following research questions:

RQ1: Does EXTREMEgen expose bugs in state-of-the-art WMC solvers?

RQ2: Does EXTREMEgen generate instances that can be solved fast enough to be usable in fuzzing?

RQ3: Does EXTREMEgen generate instances fast enough to be usable in fuzzing?

5.1 Experimental Setup

To address RQ1, we compare the EXTREMEgen generator with its three network types (DQMR, GRID, and TREE) against the baseline WCNF generators CNFuzz and FuzzSAT

implemented using support in SharpVelvet. For each of the 5 generation techniques, we generate 100 instances using SharpVelvet. We additionally generated both set 1 and set 2 instances from EXTREMEgen. Then we solved all generated instances using SharpVelvet with the solvers described in Subsubsection 5.1.

To evaluate how fast generated instances are solved, we record the time required to solve the instances for each generator and solver combination. This allows us to quantify the performance impact of different generation techniques on solver runtime. By comparing these times to the baseline generators, we determine whether a generation technique produces instances that are fast or slow to solve.

To investigate the efficiency of the instance generation techniques themselves, we measured the time taken to generate 1,000 instances for each of the 5 generation techniques. We measured CPU time using the time function from Bash. Again, by comparing these times to the baseline generators, we decide if a generation technique performs well.

All experiments were conducted on an AMD Ryzen 5 5500U, 2100 Mhz, 6 Core(s), 12 Logical Processor(s), 16 GB ram.

WMC Solvers

We tested EXTREMEgen with the solvers from the official model counting competition [10]. The solvers submitted to the 2024 competition are publicly available to download [9]. In the competition, WMC has 2 tracks: normal WMC and the bonus WMC track with negative weights. We tested the solvers submitted to the bonus WMC track. This choice should result in fewer bugs being exposed, because the bonus track solvers should be able to handle more diverse weight types. 9 different solvers were submitted to the bonus WMC track; however only 4 solvers were able to run in the SharpVelvet environment. These were GPMC [22], Ganak [19], SharpSAT-TD [15], and SharpSATTD-CH [6]. The fact that only 4 out of 9 solvers were successfully executed should not influence the results, because we include the top 3 scoring solvers: Ganak, SharpSAT-TD, and SharpSAT-TD-CH, ensuring sufficient coverage of state-of-the-art solver implementations.

Generation parameters

The baseline generators were run using standard parameters found in the example configuration from SharpVelvet. EXTREMEgen parameters were chosen to mimic the literals amount generated by the baseline generators. The Ganak solver was submitted to the competition with 2 different configurations; we tested both. The other solvers had only 1 configuration that worked. Specific parameters configurations used to run the experiments are described in the provided GitHub repository [21]. The specific instances used for research question 1 can also be found there.

5.2 Experimental Results

RQ1: We exposed 3 different bug types. The first bug type, "Timeout" is the case where a solver takes longer than the standard timeout (10 seconds) to return an answer. A timeout bug could indicate the solver is unable to solve the instance in a reasonable time or at all. The second bug type, called "Wsat," indicates that a solver states a wrong satisfiability. For each instance, if the solvers disagree on whether the instance is satisfiable, the result from the solver with minority results is considered incorrect. This approach is inspired by differential testing also used by Usman et al. in TestMC [23]. Lastly, if the final weighted sum of all satisfying assignments is wrong, we call it a "Wsum" bug. A Wsum bug occurs when a solver outputs an infinite value for a satisfiable instance, as solvers should produce a real, albeit extreme, value for satisfiable cases.

Tables 1, 2, and 3 show the number of bug-exposing instances for each generator.

Algorithm	Timeout	Wsat	Wsum
gpmc	0	0	0
ganak-conf-1	0	0	0
ganak-conf-2	0	0	0

Table 1: Bugs exposed through SharpVelvet-CNFuzz and SharpVelvet-FuzzSAT instances. 200 total instances.

Algorithm	Timeout	Wsat	Wsum
gpmc	91	183	183
ganak-conf-1	0	0	183
ganak-conf-2	0	0	183
SharpSAT-TD-weighted	0	0	183
SharpSATTD-CH-weighted	0	0	183

Table 2: Bugs exposed through EXTREMEgen DQMR, GRID, and TREE instances using weights from set 1. 300 total instances.

Algorithm	Timeout	Wsat	Wsum
gpmc	101	174	174
ganak-conf-1	0	0	174
ganak-conf-2	0	0	174
SharpSAT-TD-weighted	0	0	174
SharpSATTD-CH-weighted	0	0	174

Table 3: Bugs exposed through EXTREMEgen DQMR, GRID, and TREE instances using weights from set 1. 300 total instances.

In response to RQ1, EXTREMEgen successfully exposed bugs in all tested solvers, whereas the baseline generators did not produce any bugs, as shown in Table 1[1]. Conversely, EXTREMEgen instances exposed bugs in all tested solvers, as detailed in Tables 2 and 3. GPMC encountered timeouts and produced incorrect satisfiability and weighted sum results in over half of the instances. Furthermore, the consistency of Wsum bugs across all solvers for EXTREMEgen instances highlights the effectiveness of the extreme weights to expose this bug type.

Comparing the results from Tables 2 and 3, we notice no meaningful difference in the number of bugs exposed between Set 1 and Set 2 instances. Set 2's inclusion of ex-

[1]Running instances from the baseline generators on each of the SharpSATTD variations was not possible. Therefore, Table 1 shows only 3 rows.

treme positive and negative values did not result in additional exposed bugs, suggesting that solvers handle numerical extremes more reliably than precision challenges. This suggests that solvers are more susceptible to precision issues within the [0, 1] range from set 1, making this an area solvers could improve on.

RQ2: Figure 1 shows the average time taken by various solvers to solve instances generated by EXTREMEgen and the baseline generators. GRID instances tend to take as long or longer to solve compared to the baseline generators. DQMR instances are solved fast unless you solve with GPMC. TREE instances are the fastest to solve, generally being solved faster than the baseline instances. Overall, solving times for EXTREMEgen instances are sufficiently small to not hinder the fuzzing process, when compared to the baseline instances.

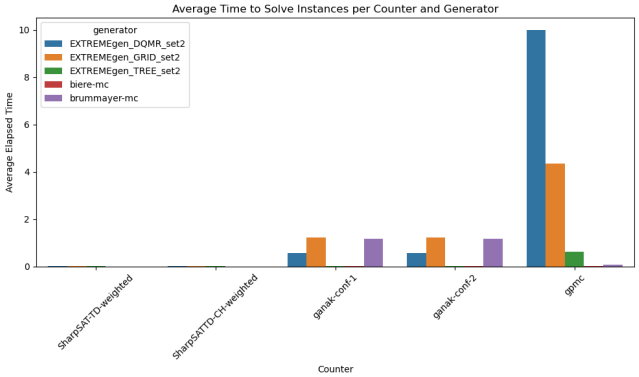

Figure 1: Average time taken by various solvers to solve instances generated by EXTREMEgen and the baseline generators.

RQ3: Figure 1 also illustrates the significant difference in generation times between EXTREMEgen and the baseline generators. Generating 1000 instances with EXTREMEgen takes about 67 times longer compared to the baseline generators. Although thought has been put into making the generation process faster, EXTREMEgen has not been optimized. From these results we conclude that for practical use in fuzzing, the generation process needs to be faster.

6 Conclusion

In this paper, we developed a generator that creates WMC instances for fuzzing state-of-the-art WMC solvers. These instances are aimed at exposing bugs related to handling extreme numbers. EXTREMEgen starts by generating a Bayesian network of type DQMR, GRID, or TREE. These types were chosen to imitate real-life WMC instances, ensuring practical relevance. After converting the Bayesian network to CNF, EXTREMEgen adds extreme weights to these networks. Weights were chosen to expose faulty multiplication and summation of extremely small and extremely large numbers while staying in the allowed WMC format. Our empirical evaluation demonstrates that EXTREMEgen achieves its main goal of exposing bugs in state-of-the-art WMC solvers. The generated instances are solved fast enough to

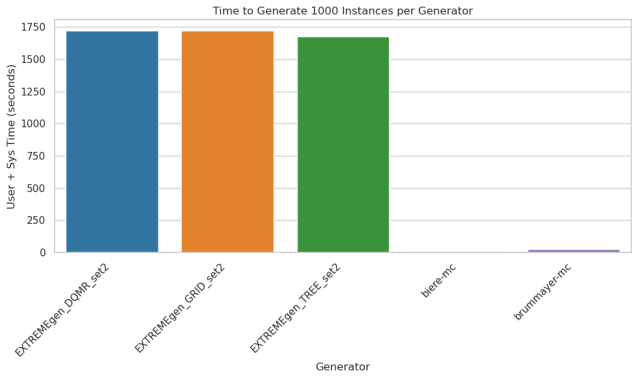

Figure 2: Time taken to generate 1000 instances for each generation technique.

be usable in fuzzing. However, generation speed needs to be optimized to be able to be a practical fuzzing generator. It currently takes longer to generate instances than it takes to solve them. When optimized, EXTREMEgen could become a practical addition to the WMC solver fuzzing toolkit.

6.1 Future Work

Optimization of Instance Generation One of the key limitations of EXTREMEgen is the long time required to generate instances. Future work should primarily focus on optimizing the generation process, particularly the network generation. This would make EXTREMEgen more suitable for real-world fuzzing applications.

Collaboration with fuzzing researchers Currently, research is being done on developing a delta-debugger, software that simplifies bug-triggering instances while still triggering the bug. Future work could integrate this delta-debugger with textsfEXTREMEgen instances. This could make it very clear which weights expose bugs. Additionally, a new tool for analyzing the similarity of generators is being developed. This tool allows researchers to identify differences between generators that may have an impact on triggering bugs. It would be interesting to test the similarity between the different network types from EXTREMEgen. This could help us understand how the structure of the networks impacts bug discovery.

7 Responsible Research

This section discusses our efforts to uphold the principles of responsible research.

All software tools and libraries used in this research are compliant with their respective licenses. The following measures have been taken to improve the reproducibility of this research. Although part of the code in EXTREMEgen cannot be publicized yet, the author of relevant code is currently in the process of making the code open source. The public part of the code, the analysis scripts, and the data used in the experiments have been published in a public GitHub repository [21]. In addition to this, the software parameters used in the experiments are described completely in section 5.1. The writing of this report has been assisted by the large

language model ChatGPT; queries and answers used are provided in the same repository. Lastly, while this research is intended to advance the field of model counting. We recognize that EXTREMEgen can be misused to develop harmful software. We urge anyone who uses this research to act responsibly with /textsfEXTREMEgen.

References

[1] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, July 2019. Conference Name: IEEE Std 754-2019 (Revision of IEEE 754-2008).

[2] Mate Soos Anna L.D. Latour. SharpVelvet, 2024.

[3] Anicet Bart, Fr&#233 Koriche, d&#233, ric, Jean-Marie Lagniez, and Pierre Marquis. An Improved CNF Encoding Scheme for Probabilistic Inference. In *ECAI 2016*, pages 613–621. IOS Press, 2016.

[4] 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.

[5] Mark Chavira and Adnan Darwiche. Compiling Bayesian networks with local structure. In *Proceedings of the 19th international joint conference on Artificial intelligence*, IJCAI'05, pages 1306–1312, San Francisco, CA, USA, July 2005. Morgan Kaufmann Publishers Inc.

[6] Yipei Deng, Junping Zhou, Jiaxin Liang, and Le Xin. SharpSAT-TD-CH, 2024.

[7] Paulius Dilkas. Generating Random Instances of Weighted Model Counting. In Andre A. Cire, editor, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, pages 395–416, Cham, 2023. Springer Nature Switzerland.

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

[9] Johannes Fichte, Markus Hecher, and Arijit Shaw. Model Counting Competition 2024: Submitted Solvers, November 2024.

[10] Johannes K. Fichte, Markus Hecher, and Florim Hamiti. The Model Counting Competition 2020. *ACM J. Exp. Algorithmics*, 26, October 2021. Place: New York, NY, USA Publisher: Association for Computing Machinery.

[11] Johannes K Fichte, Markus Hecher, and Arijit Shaw. Model Counting Competition Data Format (version 1.1). June 2024.

[12] Daan Fierens, Guy Van Den Broeck, Joris Renkens, Dimitar Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. Inference and learning in probabilistic logic programs using weighted Boolean formulas. *Theory and Practice of Logic Programming*, 15(3):358–401, May 2015.

[13] Steven Holtzen, Guy Van den Broeck, and Todd Millstein. Scaling exact inference for discrete probabilistic programs. *Software Artifact for: Scaling Exact Inference for Discrete Probabilistic Programs*, 4(OOPSLA):140:1–140:31, November 2020.

[14] Laura Kaikkonen, Tuuli Parviainen, Mika Rahikainen, Laura Uusitalo, and Annukka Lehikoinen. Bayesian Networks in Environmental Risk Assessment: A Review. *Integrated Environmental Assessment and Management*, 17(1):62–78, 2021. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/ieam.4332.

[15] Tuukka Korhonen and Matti Järvisalo. SharpSAT-TD in Model Counting Competitions 2021-2023, 2023. _eprint: 2308.15819.

[16] Mostafa Langarizadeh and Fateme Moghbeli. Applying Naive Bayesian Networks to Disease Prediction: a Systematic Review. *Acta Informatica Medica*, 24(5):364–369, October 2016.

[17] Carlos Segundo Muñoz-Valencia, José Antonio Quesada, Domingo Orozco, and Xavier Barber. Employing Bayesian Networks for the Diagnosis and Prognosis of Diseases: A Comprehensive Review, October 2023. arXiv:2304.06400 [stat].

[18] T. Sang, P. Beame, and Henry A. Kautz. Performing Bayesian Inference by Weighted Model Counting. July 2005.

[19] Shubham Sharma, Subhajit Roy, Mate Soos, and Kuldeep S. Meel. GANAK: A Scalable Probabilistic Exact Model Counter. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence*, pages 1169–1176, Macao, China, August 2019. International Joint Conferences on Artificial Intelligence Organization.

[20] M. A. Shwe, B. Middleton, D. E. Heckerman, M. Henrion, E. J. Horvitz, H. P. Lehmann, and G. F. Cooper. Probabilistic diagnosis using a reformulation of the INTERNIST-1/QMR knowledge base. I. The probabilistic model and inference algorithms. *Methods of Information in Medicine*, 30(4):241–255, October 1991.

[21] Bram Snelten. RP-Breaking-Solver-Bram, 2025. Publication Title: GitHub repository, https://github.com/bramsnelten/RP-Breaking-Solver-Bram.

[22] Ryosuke Suzuki, Kenji Hashimoto, and Masahiko Sakai. Improvement of projected model-counting solver with component decomposition using SAT solving in components. Technical report, JSAI Technical Report, SIG-FPAI-506-07, 2017.

[23] Muhammad Usman, Wenxi Wang, and Sarfraz Khurshid. TestMC: Testing Model Counters using Differential and Metamorphic Testing. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 709–721, September 2020. ISSN: 2643-1572.

[24] Ke Xu and Wei Li. The SAT phase transition. *Science in China Series E: Technological Sciences*, 42:494–501, 1999.

[25] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2024.

[26] Hamza Zerrouki, Hector Estrada-Lugo, Hacene Smadi, and Edoardo Patelli. Applications of Bayesian Networks in Chemical and Process Industries: A Review. September 2019.