Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning
Applications and Systems Track

Gaussian Elimination Meets Maximum Satisfiability*

Mate Soos and **Kuldeep S. Meel**
National University of Singapore

Abstract

Given a set of constraints F and a weight function W over the assignments, the problem of MaxSAT is to compute a maximum weighted solution of F. MaxSAT is a fundamental problem with applications in numerous areas. The success of MaxSAT solvers has prompted researchers in AI and formal methods communities to develop algorithms that can use MaxSAT solver as oracle. One such problem that stands to benefit from advances in MaxSAT solving is discrete integration. Recently, Ermon et al. achieved a significant breakthrough by reducing the problem of integration to polynomially many queries to an optimization oracle where F is conjuncted with randomly chosen XOR constraints.

The primary contribution of this paper is a new MaxSAT solver, GaussMaxHS, with built-in XOR support. The architecture of GaussMaxHS is inspired by CryptoMiniSat, which has been the workhorse of hashing-based approximate model counting techniques. Our solver, GaussMaxHS, outperforms MaxHS over 9628 benchmarks arising from spin glass models and network reliability domains. In particular, with a timeout of 5000 seconds, MaxHS could solve only 5473 benchmarks while GaussMaxHS could solve 6120 benchmarks.

1 Introduction

Given a formula F and weight function W over assignments, the problem of MaxSAT is to find maximum weighted solution of F. MaxSAT is the optimization version of the canonical NP-complete decision problem of Boolean satisfiability (SAT) and has found applications in numerous areas such as probabilistic inference, cost-optimal planning, causal discovery, model-based diagnosis and the like (Park 2002; Chen et al. 2009; Hyttinen et al. 2013).

The success of MaxSAT solvers has led to the development of algorithms for several other fundamental problems that rely on usage of MaxSAT solvers as oracles. To motivate our work, we discuss one such problem, called discrete integration, in detail. We would like to emphasize that the focus of this work is development of general purpose MaxSAT solver and not just limited to improving discrete integration tools. Discrete Integration is a fundamental problem in artificial intelligence. Given a set of constraints F and a weight function W, the problem of discrete integration is to compute the total weight of the set of solutions of input constraints. This has applications in numerous areas, including probabilistic reasoning, machine learning, planning, statistical physics, inexact computing, and constrained-random verification (Bacchus, Dalmao, and Pitassi 2003; Domshlak and Hoffmann 2007; Ermon et al. 2014; Gomes, Sabharwal, and Selman 2009; Jerrum and Sinclair 1996; Madras and Piccioni 1999; Murphy 2012).

Given computational intractability of discrete integration, efforts have focused on the study of approximate variants of the problem (Gogate and Dechter 2007; Gomes et al. 2007). Over the past decade, there has been a surge of interest in the design of hashing-based approaches to approximate #SAT wherein the weight function assigns equal weight to all the assignments. The core idea of the hashing-based framework is to employ $2-$universal hash functions, expressed as XOR constraints, to partition the solution space into *roughly equal small* cells, wherein each cell can be explored with the usage of a SAT solver. Since the SAT solver is invoked with the query consisting of the original constraints conjuncted with a set of randomly chosen XOR constraints, prior work on hashing-based techniques have advocated usage of CryptoMiniSat, an efficient SAT solver designed to handle CNF+XOR formulas by interleaving of search and Gaussian elimination procedures. The availability of CryptoMiniSat has allowed hashing-based techniques such as ApproxMC4 to handle formulas involving hundreds of thousands of variables (Soos and Meel 2019; Soos, Gocht, and Meel 2020).

Recently, in a significant breakthrough, Ermon et al. showed that the problem of discrete integration could be reduced to polynomially many calls to optimization queries (Ermon et al. 2013a). Their proposed approach, called WISH, also employs 2-universal hash functions to compose n queries to MaxSAT oracle where each query is composed of the original set of constraints F augmented with randomly chosen XOR formulas. In contrast to ApproxMC4's success for #SAT, WISH has so far not been able to achieve similar scalability. In a series of follow-up papers, Ermon et al. identify the hardness of solving optimization problems conjuncted with random XOR constraints as one of the primary bottlenecks to the scalability of WISH. Unlike CryptoMiniSat, none of state of the art MaxSAT solvers are capable of performing Gaussian Elimination at levels other than top-level.

*The accompanied tool will be released open-source at https://github.com/meelgroup/gaussmaxhs

Motivated by the success of CryptoMiniSat in aiding the scalability of hashing-based approximate counters for #SAT, one may ask: *Is it possible to design a* MaxSAT *solver with native support for XORs and whether such a solver would be efficient in practice?*

The primary contribution of this paper is an affirmative answer to the above question. We augment the state of the art MaxSAT solver MaxHS (Davies and Bacchus 2013) with native XOR support such that Gaussian elimination can be performed at every level of the search process The resulting MaxSAT solver, GaussMaxHS outperforms MaxHS in runtime performance over 9628 benchmarks arising from two applications domains: partition function computation of spin glass models and network reliability for power grids of small to medium size cities in US. While the development of GaussMaxHS was motivated by its applications in discrete integration and network reliability, we hope that having a MaxSAT solver with native support for XORs would open new research directions and new applications in a way similar to new research direction owing to existence of CryptoMiniSat.

The rest of the paper is organized as follows: we first discuss notations and preliminaries in Section 2 followed by a survey of prior work in Section 3. We then present primary technical contribution of this paper, the architecture of our solver GaussMaxHS in Section 4. We then present experimental analysis in Section 5 and finally conclude in Section 6.

2 Notations and Preliminaries

A literal is a Boolean variable or its negation. Let $X = \{x_1, x_2, \dots x_n\}$ be the set of Boolean variables. Given a Boolean formula F, the set of variables appearing in F is called the *support* of F. A *satisfying assignment* or *witness*, denoted by σ, of F is an assignment of truth values to variables in its support such that F evaluates to true. We denote the set of all witnesses of F as $Sol(F)$.

We say F is in conjunctive normal form if F can be expressed as $C_1 \wedge C_2 \wedge \cdots C_m$, where every CNF clause C_i is disjunction of literals. An XOR clause is of the form $a_1x_1 \oplus a_2x_2 \oplus \dots a_nx_n = b$ wherein $a_i, b \in \{0, 1\}$. A formula that is conjunction of CNF and XOR clauses is called CNF-XOR formula. $GF(2)$ refers to the Galois field over two elements.

Given a weight function $W : \{0,1\}^n \mapsto [0,1]$, we use $W(\sigma)$ to denote the weight of an assignment σ. To avoid notational clutter, we overload $W(\cdot)$ to denote the weight of an assignment or a formula, depending on the context. Given a set Y of assignments, we use $W(Y)$ to denote $\sum_{\sigma \in Y} W(\sigma)$. Consequently, for a given formula F and weight function W, we have $W(F) = \sum_{\sigma \in Sol(F)} W(\sigma)$ Given F and $W : \{0,1\}^n \mapsto [0,1]$, the discrete integration problem is to compute $W(F)$.

A weighted MaxSAT instance consists of two kinds of CNF clauses: hard clauses, denoted by HC, and soft clauses, denoted by SC such that $F = \mathsf{HC} \wedge \mathsf{SC}$. The weight function in the context of a weighted MaxSAT problem is defined over soft clauses. For clarity of exposition, we denote weight function defined over soft clauses as ρ. For a given F and ρ, we define weight of an assignment

$$W(\sigma) = \begin{cases} 0, \sigma \not\models C_i, \text{ where } C_i \in \mathsf{HC} \\ \sum\limits_{C_i \in \mathsf{SC} | \sigma \models C_i} \rho(C_i) \end{cases}$$

For sake of brevity, we use MaxSAT to denote the weighted MaxSAT instance.

3 Background

Prior work (Heule and van Maaren 2004; Chen 2009; Soos, Nohl, and Castelluccia 2009) has focused on augmented the CDCL solvers with native support for XOR constraints. Tightly integrating Gaussian elimination into CDCL solvers was shown to be efficient in cryptographic scenarios by Soos et al. in (Soos, Nohl, and Castelluccia 2009). The close integration of these radically different solving mechanisms was achieved by invoking Gaussian elimination before every branching decision in the CDCL solver. If the Gaussian elimination finds any truths, be them propagation(s) or a conflict, the correct action is taken by CDCL: either further propagations are carried out, or the conflict analysis routine is used to analyze the conflict returned by the Gaussian elimination procedure. Recently, Soos et al. proposed a new architecture, called BIRD, to handle CNF+XOR formulas, which has since been further improved (Soos and Meel 2019; Soos, Gocht, and Meel 2020). Integration of BIRD for MaxSAT solving is beyond the scope of this work, and is therefore, left to future work.

4 Architecture

We now describe the primary technical contribution of this paper: GaussMaxHS, a new MaxSAT solver with native support of XORs. GaussMaxHS is built by augmenting the framework of MaxHS with Gaussian elimination. The core architecture to support Gaussian elimination builds on (Soos, Nohl, and Castelluccia 2009; Han and Jiang 2012; Laitinen, Junttila, and Niemelä 2012), which has been employed in the SAT solvers: significant differences have been discussed in detail below along with MaxSAT solver specific low level optimizations.

We first discuss the representation of the matrix in Section 4.2. We describe several key low-level optimizations and the integration of Gaussian elimination in MaxHS in Section 4.3.

4.1 From Soft XOR Clauses to Hard XOR Clauses

We first discuss how every instance with soft XOR clauses can be transformed into an equivalent instance consisting of XOR clasues as only hard clauses: Let C_i be a soft XOR clause with weight $\rho(C_i) = w$. Without loss of generality, assume RHS of C_i is set to 1, i.e., C_i is of the form $\Sigma a_i x_i \oplus b = 1$ for some choices of $\{a_i\}$ and b. Now we can introduce a new variable y and replace C_i with a hard XOR clause $y \oplus C_i = 0$ conjuncted with soft unit clause $C_j := (y)$ such that $\rho(C_j) = w$.

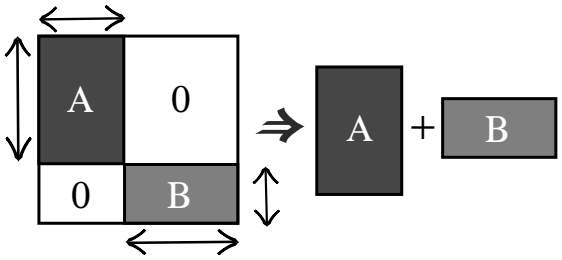

Figure 1: A matrix with two separate components.

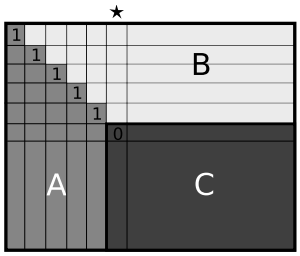

Figure 2: Progressively smaller matrices

4.2 R- and S-Matrix Sets

Since the XOR constraints are a linear set of equations over $GF(2)$, we use the matrix representation to perform Gaussian elimination. There are three key objectives that our framework needs to achieve: (i) O1: Keep track of assignments to variables, (ii) O2: For a given set of assignments, determine if the system of XOR constraints imply a *unit propagation*, (iii) O3: if a given set of assignments cause conflict in XOR constraints, determine the *conflict clause* for the same. The two objectives highlight the trade-off in the representation of the state of XOR constraints at a given level. One possible representation is to use a matrix to represent original XOR constraints (after Gaussian elimination), denoted by the matrix $\mathbf{R}$ and store the current assignment in a vector, denoted by $\mathbf{V}$. In this representation, objective O1 is trivially handled but to achieve objective O2 and objective O3; we need to substitute $\mathbf{V}$ in $\mathbf{R}$ to obtain the effective matrix at the current level, denoted by $\mathbf{S}$, which is very expensive since $\mathbf{V}$ is modified at every decision level.

To save computationally expensive recomputation, Soos et al. proposed to store both $\mathbf{R}$ and $\mathbf{S}$, where the matrix $\mathbf{S}$ is updated with the current assignment of variables and is kept upper-triangular. The matrix R is never updated with variable assignments, but every row transformations on $\mathbf{S}$ (such as those arising during Gaussian elimination on $\mathbf{S}$) is applied to $\mathbf{R}$ as well. We initialize $\mathbf{R}$ and $\mathbf{S}$ to be the same, which is determined by the XOR constraints arising from the 2-universal hash functions. An example setup is present in Fig. 3.

As the search progresses, we observe that XOR constraints can be partitioned into a disjoint set of clauses such that two sets of clauses are defined over a mutually exclusive set of variables. Therefore, in contrast to Soos et. al (Soos, Nohl, and Castelluccia 2009), we maintain multiple pairs of $\mathbf{S}$ and $\mathbf{R}$ matrices. To construct such pairs, we perform a component search over the current set of XORs by starting out with all the variables in individual groups and iteratively group variables such that variable x_1 and x_3 appear in the same group if there exists an XOR where x_1 and x_3 appear together. We then make a linear pass over all the XORs and separate XORs according to the groups' variables in a XOR belong to. Note that such a grouping of XORs does not come at the cost of the reduction in the algorithmic power of Gaussian elimination: if these matrices were in one larger matrix, and the columns and rows of the matrix were suitably ordered such as to obtain a matrix that resembles the left-hand side of Fig. 1, then running Gaussian elimination on such a matrix would lead to the same result as that on running Gaussian elimination on two separate matrices, as illustrated on the right-hand side of the same figure.

On the other hand, having separate matrices results in the reduction of complexity of Gaussian elimination. For example, the algorithmic complexity of performing Gaussian elimination on a $n \times m$ matrix is roughly $O(nm^2)$. Suppose the procedure described above can discover and separate two equally-sized separate matrices, the complexity is reduced from cnm^2 (where c is a suitable constant) to $2c(n/2)(m/2)^2 = cnm^2/4$, i.e., to the quarter of the original complexity.

4.3 Important Data Structures

Similar to SAT solving, data structures play a significant role in the efficiency of GaussMaxHS. We now discuss three key data structures that we maintain and their role in crucial subroutines of GaussMaxHS.

Assume that at some depth, let matrix $\mathbf{R}$ be as presented in Figure 2. Recall that we perform Gaussian elimination and ensure that the top left part of $\mathbf{R}$ consists of the identity matrix. Let column marked with $\star$ be the leftmost column that was updated by the CNF part of the underlying SAT solver. We use the diagonal to divide $\mathbf{R}$ into three parts as shown above: $\mathbf{A}$, $\mathbf{B}$, and $\mathbf{C}$. Now we make an important observations: The reduction to row-echelon form needs to perform only for the sub-matrix $\mathbf{C}$ while cells in $\mathbf{B}$ can only participate in propagation and conflict clause generation. Therefore, we first perform Gaussian elimination over $\mathbf{C}$ and then look for propagation and conflict clause generation over $\mathbf{B}$ and $\mathbf{C}$. To identify $\mathbf{C}$, we maintain two data structures: $left\_col\_mod$, which keeps track of the variable in the leftmost column that was assigned by the solver and $last\_1\_row$ for every column keeps the index of the row that has the last 1 in the column. Note that $left\_col\_mod$ and $last\_1\_col$ can identify $\mathbf{C}$ in constant time; thus saving potentially a search procedure that would take time linear in the size of the matrix. This leads to a progressively smaller matrix as the search gets deeper into the search tree as shown in Fig 2. This property is advantageous as CDCL spends the majority of the time deep in the search tree, where the active part of the matrix is expected to be the smallest and consequently the Gaussian elimination to be the fastest.

Note that a row in the matrix $\mathbf{R}$ can participate in propagation only if all except one element are 0. Furthermore, we need to check only the rows that were modified due to assignments by the solver. To compute whether a row $\mathbf{B}$ has exactly one element as 1, we need to scan the complete

$\mathbf{S} - matrix$ with x_3 assigned to `true`

x_1	x_2	x_3	x_4	x_5	aug
1	1	0	1	1	0
0	**1**	**0**	**0**	**0**	**0**
0	0	0	1	1	1
0	**0**	**0**	**0**	**1**	**0**
0	0	0	0	0	0

$\mathbf{R} - matrix$

x_1	x_2	x_3	x_4	x_5	aug
1	1	0	1	1	0
0	**1**	**1**	**0**	**0**	**1**
0	0	0	1	1	1
0	**0**	**1**	**0**	**1**	**1**
0	0	1	0	0	1

Figure 3: The $\mathbf{S}$-matrix indicates propagation of $x_2 =$ `false` and $x_5 =$ `false`. The XOR constraint causing these propagations are in the $\mathbf{R}$-matrix: $x_2 \oplus x_3 =$ `true` and $x_3 \oplus x_5 =$ `true`, respectively. The $\mathbf{S}$-matrix is kept upper triangular, while the $\mathbf{R}$-matrix is kept in a state that any XOR constraint in it is a combination of the original problem's XOR constraints.

row in the worst case. To this end, we maintain the variable $first\_1\_col$ for every row, which keeps track of the index of the first column with "1". As the search progresses, this allows us to reduce our search space from entire row to the length determined by $first\_1\_col$.

Low-level Optimizations We next discuss two low-level optimizations that are crucial to the performance of GaussMaxHS and are of potential interest to the satisfiability and constraints community.

The first optimization concerns the storage of our matrix data structure. We considered the sparse matrix representation (Gibbs, Poole, and Stockmeyer 1976); however, sparser representation is useful only with a low density. In contrast for XOR arising from $H_{xor}(n, m)$ the density is set to 0.5. Therefore, we store both $\mathbf{R}$ and $\mathbf{S}$ in a dense, bit-packed format while we store the augmented column in unpacked format. The dense, bit-packed format has three key advantages: (1) XOR-ing rows can be done 32, 64 or more bits at a time, depending on the available SIMD instruction set of the processor such as SSE, AVX; (2) it is easy to check the augmented column's value, which is checked often; and (3) the bit-packed format is memory efficient, which is important for speed of storage and retrieval of matrices.

The second optimization concerns the storage of matrices $\mathbf{R}$ and $\mathbf{S}$. We store $\mathbf{R}$ and $\mathbf{S}$ in an interlaced fashion in a memory array. The rows follow each other in the memory as: $\mathbf{S}[0], \mathbf{R}[0], \ldots S[r-1], \mathbf{R}[r-1] \ldots$, where $S[0]$ is the first row of the A-matrix, $\mathbf{R}[0]$ is the first row of the $\mathbf{R}$-matrix. This storage structure is advantageous, since when the two m-long rows x and y need to be swapped or XOR-ed, they need to be swapped/XOR-ed in both matrices, and the memory swap/XOR operation can work on two $2m$-long memory areas $S[x] \ldots S[x] + 2m$ and $S[y] \ldots S[y] + 2m$ instead of working on four m-long memory areas. Since row swapping and XOR-ing can account for up to 1/3rd of the total time spent in Gaussian elimination, this is an important optimization.

Implementation Details: We focused on modular integration of Gaussian elimination into MaxHS to allow further development.

We focused on storage of clauses inside MaxHS. Clauses are stored using the `Packed_vecs` class in `Wcnf`'s `hard_cls` and `soft_cls`. These store literals in contiguous memory, with a separate vector of `sizes` keeping track of when one clause starts and the other ends. We modified `sizes` to store a struct `sz_and_xor` instead of only the size, where we now also store whether the clause is an XOR or a regular clause. We then modified the `getVec` function of `Packed_vecs` and made sure to insert the clauses according to their type into the two MiniSat instances that GaussMaxHS keeps.

5 Evaluation

Our solver, called GaussMaxHS, is built by augmenting the code of state of the art MaxSAT solver, MaxHS[1], with Gaussian elimination. To determine runtime performance and test correctness of GaussMaxHS, we conduct experiments over instances constructed by queries of hashing-based techniques for benchmarks arising from two application domains: (i) computation of partition function for spin-glass models, and (ii) network reliability. It is worth pointing out that prior work on discrete integration present results corresponding to spin-glass models but we have also incorporated results from another domain of network reliability to showcases generalizability of the performance of GaussMaxHS. In total, our benchmark suite consisted of 9628 instances.

It is worth emphasizing that each of our benchmark is a MaxSAT instance generated by WISH and in particular, none of the instances are *synthetic benchmarks* constructed by us.

The primary objective of our experimental evaluation was to answer the following question: *How does the performance of* GaussMaxHS *compare to that of* MaxHS*?*

In summary, we observe that GaussMaxHS is able to achieve speedup of up to 2 orders of magnitude (i.e., 100× speedup) over MaxHS augmented with top-level Gaussian elimination. Furthermore, for all the cases where MaxHS terminated, the weight of the answers computed by GaussMaxHS and MaxHS match. Note that GaussMaxHS and MaxHS do not necessarily compute the same solution if there are more than one optimal solutions. The baseline version that we compared against performs the following steps: first perform Gaussian Elimination (GE) at the top level, then cuts XORs into short XORs and convert into CNF

[1] MaxHS won 2020 MaxSAT competition in complete track

Instance	vars	cls	xors	Max HS	Gauss MaxHS	Speed-up
sping2-x_11	49	193	11	246.70	15.67	15.74
sping1-x_6	49	193	9	22.43	1.77	12.64
sping2-x_5	49	193	50	0.02	0.02	1.00
sping3-x_11	49	193	22	1557.77	116.89	13.33
sping5-x_3	49	193	15	310.97	9.70	32.06
sping5-x_15	49	193	29	-	2930.45	-
sping5-x_11	49	193	45	-	0.01	-
sping6-x_7	49	193	46	-	0.02	-
Net12_24_13_cnt_106	191	320	14	591.33	10.84	54.57
Net12_24_13_cnt_106	191	320	12	596.24	21.08	28.29
Net22_60_8_cnt_116	219	350	22	-	353.91	-
Net22_60_8_cnt_116	219	350	9	40.54	8.99	4.51
Net3_51_19_cnt_65	117	197	21	2652.90	671.09	3.95
Net22_84_16_cnt_116	219	350	190	-	501.74	-
Net22_84_16_cnt_116	219	350	196	-	16.22	-
Net27_81_58_cnt_118	230	356	202	-	1133.91	-
Net27_81_58_cnt_118	230	356	199	-	4825.98	-
Net27_90_62_cnt_118	230	356	202	-	311.05	-

Table 1: Runtime performance comparison of GaussMaxHS vis-a-vis MaxHS for a subset of benchmarks

formulas and passes the resulting formula to MaxHS. In other words, we integrated top-level GE in MaxHS to demonstrate that performing GE at top-level is not sufficient, thereby emphasizing the need for the design of our solver. It is again worth emphasizing that our focus is on evaluating MaxSAT solvers and we defer a detailed analysis of impact of our solver on underlying algorithms for different applications to future work.

The experiments were conducted on a high performance computing cluster with nodes consisting of E5-2690v3 CPUs with 24 cores and 96GB of RAM each. We conducted experiments for over 9628 benchmarks. For lack of space, we present individual results for only a subset of representative benchmarks. Every experiment ran GaussMaxHS and MaxHS with a 5000s timeout. Our benchmarks were derived from the following two application domains:

Spin Glass Model : A spin glass model is defined with variables $x_i \in \{-1, +1\}$ for $i \in [n]$, where each variable represents a spin. We focused on grid models where each spin variable has 4 neighbors. The interactions between neighbors x_i and x_j are captured by $\theta_{i,j}$ such $\theta_{i,j}(x_i, x_j) = \beta_{i,j} x_i x_j$. Finally, the potential function of the spin glass model is defined as $\theta(x_1, x_2, \cdots x_n) = \sum_{i \in V} \alpha_i x_i + \sum_{(i,j) \in E} (\beta_{i,j} x_i x_j)$. Keeping in line with the previous work (Kuck, Sabharwal, and Ermon 2018), we focus on 7×7 spin glass model. The parameters α_i are chosen uniformly at random from $[-1, 1]$ while β are chosen uniformly at random from $[0, c)$ for different values of c.

Network Reliability A power grid can be modeled as a graph $G = (V, E)$ each of whose edges $e \in E$ fail independently with probability p_e. Given two particular nodes of interest, say s and t, the problem of network reliability is to compute the probability that s and t are connected. It was shown that the problem of network reliability can be reduced to discrete integration (Duenas-Osorio et al. 2017). In this work, similar to recent work (Duenas-Osorio et al. 2017), we consider benchmarks arising from transmission grids of medium sized cities in USA. We refer to the reader to (Duenas-Osorio et al. 2017) for more details on encoding.

5.1 Results

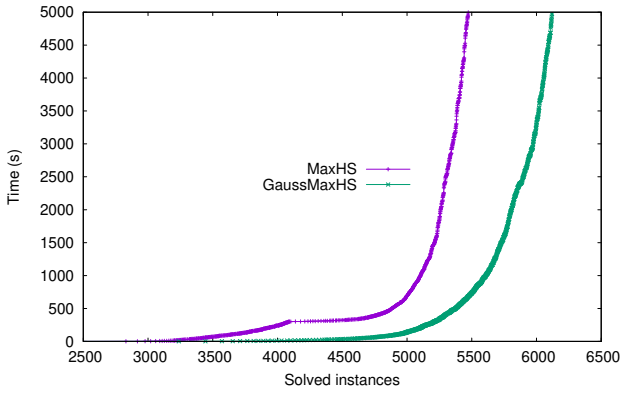

Figure 4: Cactus plot showing behavior of MaxHS and GaussMaxHS

Figure 4 shows the cactus plot for GaussMaxHS vis-a-vis MaxHS. We present the number of benchmarks on x-axis and the time taken on $y-$axis. A point (x, y) implies that x benchmarks took less than or equal to y seconds to solve. With a timeout of 5000 seconds, MaxHS could solve only 5473 benchmarks while GaussMaxHS could solve 6120

benchmarks.

Table 1 presents the performance of GaussMaxHS vis-a-vis MaxHS over a subset of our benchmarks. Column 1 of this table gives the benchmark name, while columns 2, 3, and 4 list the number of variables, non-XOR clauses, and XOR clauses respectively. Columns 5 and 6 list the runtime (in seconds) of GaussMaxHS and MaxHS respectively. We use "–" to denote timeout after 5000 seconds. It is natural to wonder how the problem on just 49 variables is challenging: in this context, it is worth emphasizing that the hardness comes from XORs as the problem without XORs is very simple for MaxSAT solvers, thereby highlighting the need for MaxSAT solvers with native support for XORs. Also, there is no noticeable overhead in the performance of GaussMaxHS. Table 1 clearly demonstrates GaussMaxHS significantly outperforms MaxHS. Furthermore, as the number of XOR constraints increase, GaussMaxHS performs better than MaxHS. Recall that for a given formula F over n variables, WISH constructs MaxSAT queries where queries are formed by conjuncting the original formula F with i XOR constraints where i ranges from 1 to n. Therefore, every invocation of WISH would produce MaxSAT queries with large number of XOR constraints. The superior performance of GaussMaxHS over MaxHS is encouraging as the previous implementations of hashing-based techniques had to settle for lower bound approximations of integrals due to inability of existing optimization tools to handle XOR constraints (Ermon et al. 2013b)

While we have presented empirical comparisons vis-a-vis MaxHS, the performance of other MaxSAT solvers on these instances is similar to that of MaxHS owing to their lack of Gaussian Elimination.

6 Conclusion

The success of MaxSAT solvers have led to the development of algorithms that invoke the state of the art MaxSAT solvers as oracles. Recent hashing-based approaches have highlighted the need for MaxSAT solvers that can support queries conjuncted with random XOR constraints. In this paper, we propose a new MaxSAT solver, GaussMaxHS, with native XOR support. We have shown GaussMaxHS outperforms the state of the art MaxSAT solver, MaxHS, by 1-2 orders of magnitude. We hope the success of GaussMaxHS will lead to new research directions and new applications of MaxSAT. From technical perspective, an interesting line of work would be to augment GaussMaxHS with the recently proposed BIRD architecture (Soos and Meel 2019; Soos, Gocht, and Meel 2020).

Acknowledgements

This work was supported in part by National Research Foundation Singapore under its NRF Fellowship Programme[NRF-NRFFAI1-2019-0004] and AI Singapore Programme [AISG-RP-2018-005], and NUS ODPRT Grant [R-252-000-685-13]. The computational work for this article was performed on resources of the National Supercomputing Centre, Singapore (https://www.nscc.sg).

References

Bacchus, F.; Dalmao, S.; and Pitassi, T. 2003. Algorithms and complexity results for #SAT and Bayesian inference. In *Proc. of FOCS*, 340–351.

Chen, Y.; Safarpour, S.; Veneris, A.; and Marques-Silva, J. 2009. Spatial and temporal design debug using partial maxsat. In *Proceedings of the 19th ACM Great Lakes symposium on VLSI*, 345–350. ACM.

Chen, J. 2009. Building a hybrid SAT solver via conflict-driven, look-ahead and XOR reasoning techniques. In *Proc. of SAT*, 298–311.

Davies, J., and Bacchus, J. 2013. Postponing optimization to speed up MAXSAT solving. In *Proc. of CP*, volume 8124 of *Lecture Notes in Computer Science*, 247–262. Springer.

Domshlak, C., and Hoffmann, J. 2007. Probabilistic planning via heuristic forward search and weighted model counting. *Journal of Artificial Intelligence Research* 30(1):565–620.

Duenas-Osorio, L.; Meel, K. S.; Paredes, R.; and Vardi, M. Y. 2017. Counting-based reliability estimation for power-transmission grids. In *Thirty-First AAAI Conference on Artificial Intelligence*.

Ermon, S.; Gomes, C. P.; Sabharwal, A.; and Selman, B. 2013a. Optimization with parity constraints: From binary codes to discrete integration. In *Proc. of UAI*.

Ermon, S.; Gomes, C. P.; Sabharwal, A.; and Selman, B. 2013b. Taming the curse of dimensionality: Discrete integration by hashing and optimization. In *Proc. of ICML*, 334–342.

Ermon, S.; Gomes, C. P.; Sabharwal, A.; and Selman, B. 2014. Low-density parity constraints for hashing-based discrete integration. In *Proc. of ICML*, 271–279.

Gibbs, N. E.; Poole, Jr., W. G.; and Stockmeyer, P. K. 1976. A comparison of several bandwidth and profile reduction algorithms. *ACM Trans. Math. Softw.* 2(4):322–330.

Gogate, V., and Dechter, R. 2007. Approximate counting by sampling the backtrack-free search space. In *Proc. of the AAAI*, volume 22, 198.

Gomes, C. P.; Hoffmann, J.; Sabharwal, A.; and Selman, B. 2007. From sampling to model counting. In *Proc. of IJCAI*, 2293–2299.

Gomes, C. P.; Sabharwal, A.; and Selman, B. 2009. Model counting. In Biere, A.; Heule, M.; Maaren, H. V.; and Walsh, T., eds., *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press. 633–654.

Han, C.-S., and Jiang, J.-H. R. 2012. When boolean satisfiability meets gaussian elimination in a simplex way. In *Proc. of CAV*, 410–426.

Heule, M., and van Maaren, H. 2004. Aligning CNF- and equivalence-reasoning. In Hoos, H. H., and Mitchell, D. G., eds., *SAT (Selected Papers*, volume 3542 of *LNCS*, 145–156. Springer.

Hyttinen, A.; Hoyer, P. O.; Eberhardt, F.; and Jarvisalo, M. 2013. Discovering cyclic causal models with latent

variables: A general sat-based procedure. *arXiv preprint arXiv:1309.6836*.

Jerrum, M. R., and Sinclair, A. 1996. The Markov Chain Monte Carlo method: an approach to approximate counting and integration. *Approximation algorithms for NP-hard problems* 482–520.

Kuck, J.; Sabharwal, A.; and Ermon, S. 2018. Approximate inference via weighted rademacher complexity. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

Laitinen, T.; Junttila, T.; and Niemelä, I. 2012. Extending clause learning sat solvers with complete parity reasoning. In *2012 IEEE 24th International Conference on Tools with Artificial Intelligence*, volume 1, 65–72. IEEE.

Madras, N., and Piccioni, M. 1999. Importance sampling for families of distributions. *Annals of applied probability* 1202–1225.

Murphy, K. 2012. *Machine Learning: A Probabilistic Perspective*. MIT Press.

Park, J. D. 2002. Map complexity results and approximation methods. In *Proceedings of UAI*, 388–396.

Soos, M., and Meel, K. S. 2019. Bird: Engineering an efficient cnf-xor sat solver and its applications to approximate model counting. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, 1592–1599.

Soos, M.; Gocht, S.; and Meel, K. S. 2020. Tinted, detached, and lazy CNF-XOR solving and its applications to counting and sampling. In *Proc. of CAV*.

Soos, M.; Nohl, K.; and Castelluccia, C. 2009. Extending SAT solvers to cryptographic problems. In *SAT*, 244–257.