On the Usefulness of Linear Modular Arithmetic in Constraint Programming

Gilles Pesant[1], Kuldeep S. Meel[2], Mahshid Mohammadalitajrishi[1]

[1] Polytechnique Montréal, Canada
gilles.pesant@polymtl.ca
[2] National University of Singapore
meel@comp.nus.edu.sg

Abstract. Linear modular constraints are a powerful class of constraints that arise naturally in cryptanalysis, checksums, hash functions, and the like. Given their importance, the past few years have witnessed the design of combinatorial solvers with native support for linear modular constraints, and the availability of such solvers has led to the emergence of new applications. While there exist global constraints in CP that consider congruence classes over domain values, linear modular arithmetic constraints have yet to appear in the global constraint catalogue despite their past investigation in the context of model counting for CSPs. In this work we seek to remedy the situation by advocating the integration of linear modular constraints in state-of-the-art CP solvers.
Contrary to previous belief, we conclude from an empirical investigation that Gauss-Jordan Elimination based techniques can provide an efficient and scalable way to handle linear modular constraints. On the theoretical side, we remark on the pairwise independence offered by hash functions based on linear modular constraints, and then discuss the design of hashing-based model counters for CP, supported by empirical results showing the accuracy and computational savings that can be achieved. We further demonstrate the usefulness of native support for linear modular constraints with applications to checksums and model counting.

1 Introduction

Given a set of variables $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ with their associated domains of values $\mathcal{D} = \{D_1, D_2, \ldots, D_n\}$ and set of constraints $\mathcal{C}$ over $\mathcal{X}$, the Constraint Satisfaction Problem (CSP), denoted $\varphi = (\mathcal{X}, \mathcal{D}, \mathcal{C})$, seeks to assign to each variable $x_i \in \mathcal{X}$ a value from D_i such that every constraint in $\mathcal{C}$ is satisfied. It is often convenient and effective to use constraints that can succinctly express recurring relations of arbitrary arity. The global constraints catalogue [2] has grown over the years to encompass a wide variety of such constraints, including the case of values considered modulo a given parameter (e.g. ALLDIFFERENT_MODULO, AMONG_MODULO, MAXIMUM_MODULO). But despite the investigation of linear modular arithmetic constraints by Gomes et al. [11] in the context of model counting for CSPs, the latter constraints seem to have gone largely unnoticed in the CP community and indeed do not appear in that catalogue.[3]

[3] One exception is the work on bit-vector domains, involving some modular arithmetic, with applications to software verification and cryptography [1, 13, 8].

The purpose of this paper is to advocate the inclusion of modular arithmetic constraints in CP solvers, motivated by important applications such as model counting, and to investigate the algorithmic opportunities currently available for efficient inference on such constraints as well as remaining challenges, through an empirical evaluation featuring linear modular arithmetic constraints.

In this paper we address the question of how to efficiently integrate linear modular constraints in a CP solver. As mentioned before, Gomes et al. [11] studied linear modular equalities in the context of model counting and remarked that a system based on Gauss-Jordan Elimination (GJE) would be inefficient. As a result, they proposed an adaptation of Trick's dynamic programming algorithm [18] to handle individual constraints. Their empirical evaluation was limited to short constraints (i.e. on about six variables). We take advantage of the compact table implementation for extensional constraints [9] to revisit GJE and thus reach the opposite conclusion: GJE applied to a system of linear modular constraints achieves significantly better performance than the alternative dynamic programming algorithm on individual constraints.

We demonstrate the scalability of our framework through an empirical evaluation on large linear modular constraints and show the opportunities offered by a solver with native support for linear modular constraints. Here we can draw a parallel with the availability of CryptoMiniSat, a SAT solver with native support for linear modular constraints in the Boolean domain (i.e., XOR constraints), which has opened up several applications. Linear modular arithmetic constraints naturally occur in several domains such as checksums, error correcting codes, cryptography, learning parity without noise, and model counting. In this paper we present applications to checksums and model counting.

The rest of the paper is organized as follows. We first present background and formal definition and representation of linear modular arithmetic constraints in Section 2. We present domain filtering algorithms for linear modular constraints in Section 3. We then present applications to checksums and model counting in Sections 4 and 5. Finally, we conclude in Section 6.

2 Background

An integer *modulus* $p > 1$ defines a congruence equivalence relation on the set of all integers $\mathbb{Z}$: integers i and j are said to be congruent if there exists an integer k such that $i - j = kp$. Thus it partitions $\mathbb{Z}$ into p congruence classes, the ring of integers modulo p, on which addition and multiplication are defined in the obvious way.

We are interested in linear modular arithmetic constraints of the general form

$$\ell \leq ax \leq u \pmod{p}$$

where x is a vector of n integer finite-domain variables, a a vector of integer coefficients, ℓ and u two integers, and p the modulus. We will also be interested in systems of m linear modular equalities in n integer finite-domain variables,

$$Ax = b \pmod{p}.$$

An integer i in such a ring has a multiplicative inverse if and only if i and p are coprime. When p is prime then clearly every $0 < i < p$ is coprime with p. In fact the ring of integers modulo p is a finite field $\mathbb{F}_p$ — every non-zero element having a multiplicative inverse — if and only if p is prime.

Gauss-Jordan Elimination can solve systems of linear equations not only over the real numbers but also over any field, such as $\mathbb{F}_p$. We take advantage of this in Section 3.

The linear modular equations are closely related to the universal hash functions. Given two finite sets N and M, let $\mathcal{H}(N, M) \triangleq \{h : N \to M\}$ be a family of hash functions mapping N to M. We use $h \xleftarrow{R} \mathcal{H}(N, M)$ to denote the probability space obtained by choosing a function h uniformly at random from $\mathcal{H}(N, M)$.

Definition 1. *A family of hash functions $\mathcal{H}(N, M)$ is k-wise independent if $\forall \alpha_1, \alpha_2, \ldots \alpha_k \in M$ and for distinct $y_1, y_2, \ldots y_k \in N$, $h \xleftarrow{R} \mathcal{H}(N, M)$,*

$$\Pr\left[(h(y_1) = \alpha_1) \wedge (h(y_2) = \alpha_2) \ldots \wedge (h(y_k) = \alpha_k)\right] = \left(\frac{1}{M}\right)^k \tag{1}$$

Note that every k-wise independent hash family is also $k - 1$ wise independent. The phrase *strongly 2-universal* is also used to refer to 2-wise independent as noted by Vadhan in [19], although the concept of 2-universal hashing proposed by Carter and Wegman [3] only required that $\Pr[h(x) = h(y)] \leq \frac{1}{2^m}$.

3 Domain Filtering for Linear Modular Constraints

Gomes et al. [11] proposed filtering algorithms for linear finite-domain constraints over $\mathbb{F}_p$ — in this section we describe our implementation,[4] including some important improvements. On equality constraints one can apply both GJE (provided p is prime) to simplify the system and optionally reach domain consistency, and also the dynamic programming representation for individual constraints to reach domain consistency. On inequality constraints only the latter applies. Note that domain values belonging to the same congruence class in $\mathbb{F}_p$ can be managed as a single one since their supports for these constraints will always be identical.

3.1 Gauss-Jordan Elimination for Systems of Linear Modular Equality Constraints with a Prime Modulus

When p is prime every element of the finite field has a multiplicative inverse, which is required to apply GJE in order to simplify and solve systems of linear equations over $\mathbb{F}_p$. We precompute multiplicative inverses using the Extended Euclidean algorithm, which also allows us to confirm that p is prime. We do not reproduce these two algorithms here as they are well known.

Because our variables are not free but each have a finite domain restricting their value, deciding satisfiability for the system is not immediate given the reduced row

[4] available at https://github.com/PesantGilles/MiniCPBP

Algorithm 1: Filtering algorithm for system $Ax = b \pmod p$

```
τ^ub ← 1
for i ← 1 to n_p do
    if x_{p[i]} is bound then
        transfer index p[i] into b
    else
        τ^ub ← τ^ub × |D(x_{p[i]})|
if τ^ub ≤ τ^max then
    T ← ∅
    enumParamVars(1)
    if T is empty then
        fail
    else
        post TABLE(⟨x_i⟩_{i∈p} ⟨x_i⟩_{i∈d}, T)
        set Algorithm 1 as inactive
```

echelon form. We may find that the system is inconsistent in which case we report it. Otherwise the resulting parametric form yields a more efficient domain consistency algorithm and smaller (i.e. with fewer variables) individual equality constraints to feed potentially to the dynamic programming filtering algorithm.

3.2 Domain Consistency for a System of Linear Modular Equality Constraints in Parametric Form

Recall that Gomes et al. [11] chose not to implement GJE. We present a straightforward algorithm to achieve domain consistency on such systems and which is tractable when the number of parametric variables is small enough. Basically we enumerate the combinations of values for the parametric variables and check that each equation in the parametric form is satisfiable, i.e. that the required value belongs to the domain of the corresponding nonparametric variable. Any unsupported value in the domain of a parametric variable should be removed — any never-required value in the domain of a nonparametric variable should also be removed. Actually there already exists a constraint that can enforce this for us and even provide an efficient incremental algorithm: a TABLE constraint on the enumerated tuples using the compact table implementation. However as the number of tuples grows exponentially with the number of parametric variables we only enforce domain consistency once the number of tuples falls below a given threshold $\tau^{\max}$ as variables become bound and domains are reduced.

Let $\mathbf{p}$, $\mathbf{b}$, and $\mathbf{d}$ denote the array of indices of unbound parametric, bound parametric, and non-parametric (dependent) variables respectively, and n_p, n_b, n_d their size. We call Algorithm 1 whenever a parametric variable becomes bound. It first transfers newly-bound variables from $\mathbf{p}$ to $\mathbf{b}$ while at the same time computing the size of the Cartesian product of the domains of the remaining parametric variables, which is an upper bound on the number of valid tuples we would enumerate. If that upper bound does not exceed our threshold $\tau^{\max}$ we proceed to enumerate valid tuples (see Algorithm 2)

Algorithm 2: enumParamVars(r)

```
if r ≤ n_p then
    foreach v ∈ D(x_{p[r]}) do
        τ[r] ← v
        enumParamVars(r + 1)
else
    for i ← 1 to n_d do
        s ← b[d[i]]
        for j ← 1 to n_b do
            s ← s − A[d[i]][b[j]] × x_{b[j]}
        for j ← 1 to n_p do
            s ← s − A[d[i]][p[j]] × τ[j]
        s ← s (mod p)
        if s ∉ D(x_{d[i]}) then
            return
        τ[n_p + i] ← s
    T ← T ∪ {τ}
```

and then post a TABLE constraint on the unbound variables. Once this happens, Algorithm 1 will no longer be called until we backtrack over that posted TABLE constraint.[5]

Theorem 1. *Algorithm 1 has a worst-case running time in* $\Theta(m(n-m)p^{n_p})$.

Proof. Its time complexity is dominated by that of Algorithm 2. We map domains to the set $\{0, 1, \ldots, p-1\}$ and there are n_p parametric variables, so we have at most p^{n_p} tuples to enumerate. For each tuple there are at most m equations (the rank of the row-reduced matrix) on $n - m + 1$ variables to evaluate. We can check whether a value belongs to a domain in constant time (sparse set representation). □

In practice our choice of threshold $\tau^{\max}$ keeps the exponential factor p^{n_p} in check.

3.3 Dynamic Programming for a Single Linear Modular Constraint

We next describe a simple adaptation of an existing filtering algorithm for individual linear constraints to be used when dealing with an inequality constraint or in conjunction with the previous algorithm for systems of linear equalities, as previously proposed [11].

First observe that the usual bounds consistency algorithm for linear constraints does not work correctly here. Consider for example

$$2x + y = 4 \qquad x, y \in \{1, 2, 3, 4\}.$$

[5] In practice we actually implement the compact table filtering algorithm and apply it directly instead of repeatedly posting and retracting TABLE constraints.

Reasoning from the smallest value in the domain of x allows us to determine that the largest feasible value for y is 2, thereby declaring values 3 and 4 unsupported and filtering them out of the domain of y. But if we have instead

$$2x + y = 4 \pmod 5 \qquad x, y \in \{1, 2, 3, 4\}$$

then that same reasoning is incorrect since, for example, value 3 for y is supported by value 3 for x since $2 \cdot 3 + 3 = 9 \equiv 4 \pmod 5$. Note that the domain value yielding the smallest contribution of a variable with positive coefficient to the equation is not necessarily the smallest one: here value 3 gives the smallest contribution, 1, for x.

Consider the general linear modular constraint $\ell \leq ax \leq u \pmod p$ with an equality constraint corresponding to the special case $\ell = u$. The pseudo-polytime domain consistency algorithm based on dynamic programming that was originally proposed for knapsack constraints [18] can be easily adapted for modular arithmetic, leading to a worst-case time complexity in $\Theta(np \min(d, p))$ where d stands for the domain size. It potentially becomes less time- and space-consuming than its original version if the modulus is not too large, which is typically the case in many applications, and even truly polynomial if p is polynomially-related to the domain size or to the number of variables. [6] If there are several equality constraints of same prime modulus, GJE will have reduced the number of variables in each constraint, making the algorithm even faster.

We use that algorithm once the number of unbound variables falls below some chosen threshold $v^{\max}$: modular arithmetic makes the state space very densely connected which makes it hard to filter anything in the presence of several variables providing many degrees of freedom. That same observation led Gomes et al. [11] to apply it with at most six variables.

4 Application to Checksums

Checksums are commonly used to ensure data integrity of various identifiers such as social security and medicare numbers. This section is meant as an illustration of the usefulness of CP equipped with linear modular constraints, here for checksums.

The International Standard Book Number (ISBN) is a unique identifier for books that uses a checksum in order to ensure its integrity. Originally ISBNs append a check digit (actually ranging from 0 to 10) to a nine-digit identifier. That check digit x_{10} is determined through a weighted sum with the other digits $x_1, \ldots, x_9$ in modular arithmetic:

$$\sum_{i=1}^{10} (11 - i)x_i \equiv 0 \pmod{11}$$

This added redundancy helps detect some common transcription errors: one can detect any single digit mistake as well as any pair of swapped digits. However double

[6] For example if p is chosen as the smallest prime number larger than the domain size, as one can always find a prime between d and $2d$ for any $d > 1$.

digit mistakes may go undetected. Arguably some digit mistakes are more likely than others, particularly from a handwritten version. For example digit "1" is easily confused with a "7" but not with an "8". So a natural question is: If we restrict double digit mistakes to such easily confused pairs, can they still go undetected?

We can write a CP model to help investigate this. Consider the very conservative set of confused ordered pairs $\mathcal{P} = \{(1,7),(7,1),(3,5),(5,3),(5,8),(8,5)\}$ and let $a_k = 11 - k \ (1 \leq k \leq 10)$, the coefficients of the ISBN checksum. Sequences of variables $\langle x_1, x_2, \ldots, x_{10}\rangle$ and $\langle y_1, y_2, \ldots, y_{10}\rangle$ each model an ISBN. For every two digit positions $\langle i, j\rangle_{\ 1\leq i<j\leq 10}$ we ask whether, given a valid ISBN, replacing each digit at these positions by another from a confused pair can yield another valid ISBN:

$$
\begin{array}{ll}
\textsc{Sum\_modulo}(\langle a_k\rangle_{1\leq k\leq 10}, \langle x_k\rangle_{1\leq k\leq 10}, 0, 11) & \\
\textsc{Sum\_modulo}(\langle a_k\rangle_{1\leq k\leq 10}, \langle y_k\rangle_{1\leq k\leq 10}, 0, 11) & \\
\textsc{Table}(\langle x_i, y_i\rangle, \mathcal{P}) & \\
\textsc{Table}(\langle x_j, y_j\rangle, \mathcal{P}) & \\
y_k = x_k & 1\leq k\leq 10,\ k\neq i, k\neq j \\
x_i < y_i & \\
x_k \in \{0,1,\ldots,9\} & 1\leq k\leq 9 \\
x_{10} \in \{0,1,\ldots,10\} &
\end{array}
$$

The validity of each ISBN is enforced by a linear modular constraint SUM_MODULO. The close relationship of these ISBNs is enforced by using TABLE constraints for positions i and j constrained to exchange digits from a confused pair and by setting the other digits to be equal. We also add an inequality between the digits at position i in order to avoid symmetric solutions. Because many of the digits in the two ISBNs are identical and since we only seek to know whether or not the model is satisfiable, prior to search we arbitrarily set most of them to zero while leaving enough degrees of freedom, which greatly accelerates search.

Solving this model we find many solutions, indicating a real risk that such mistakes go undetected even when we consider few pairs of confused digits. Inspecting these solutions we find for example that if the leading digit is a "1" or a "7" being exchanged, the second exchanged digit yielding an undetected mistake (i.e. a valid ISBN) must occur at a position among the set $\{2,3,8,9,10\}$. We also notice that any confused pair can be used twice at positions $\langle 1, 10\rangle$, $\langle 2, 9\rangle$, $\langle 3, 8\rangle$ and so forth. This is actually true for any arbitrary pair of digits (d_1, d_2) and can be derived analytically:

$$a_k d_\ell + a_{11-k} d_\ell = (11-k)d_\ell + k d_\ell = 11 d_\ell \equiv 0 \pmod{11} \quad 1\leq k\leq 10,\ 1\leq \ell\leq 2$$

So even a single allowed pair of exchangeable digits, occurring at the right combination of positions, can lead to an undetected mistake.

Now what if we added a second checksum? For example we rotate left by one position the vector of coefficients $\langle a_k\rangle_{1\leq k\leq 10}$ and add the corresponding SUM_MODULO constraint with a new check digit. Solving this augmented model reveals that all double digit mistakes are now detected. Can it even detect triple digit mistakes? No — even restricting to the set $\mathcal{P}$ of confused pairs, each triplet of digit positions admits exactly one

combination of three pairs hiding the mistake. If we restrict further the confused pairs solely to "1" and "7" then any such mistake will be detected (i.e. we find no solution for any triplet). However if we had chosen instead a left rotation by three positions for the second checksum, we discover by solving the corresponding CP model that there is a single (though unlikely) undetected mistake at positions $\langle 5, 8, 9\rangle$:

$$a_5 + a_8 + a_9 = \ 6 + 3 + 2 \ = 11 \equiv 0 \pmod{11}$$
$$a_8 + a_1 + a_2 = 3 + 10 + 9 = 22 \equiv 0 \pmod{11}$$

Again this serves only as an illustration of the kind of analysis made easier with CP.

5 Application to Model Counting

We now focus on the problem of model counting and demonstrate how the native support of linear modular constraints can lead to the development of scalable model counting techniques.

Given a CSP φ, let $\mathsf{sol}(\varphi)$ represent the set of solutions of φ. The problem of model counting is to estimate $|\mathsf{sol}(\varphi)|$. An approximate model counter takes in a CSP instance φ, tolerance parameter ε, and confidence parameter δ as input and returns an estimate c such that $\Pr[\frac{|\mathsf{sol}(\varphi)|}{1+\varepsilon} \leq c \leq (1+\varepsilon)|\mathsf{sol}(\varphi)|] \geq 1 - \delta$.

The seminal work of Valiant [20] showed that this problem is #P-complete and the hardness manifests itself in the practical implementations of exact counting. Consequently, there has been a surge of interest in the design of approximate techniques. Hashing-based techniques have emerged as a dominant approach over the past few years with its promise of scalability and rigorous (ε, δ)-guarantees. The core idea is to employ pairwise independent hash functions to partition the solution space of φ into *roughly equal small* cells of solutions. To this end, the standard family of pairwise independent hash functions in the context of Boolean variables consists of linear polynomials over $\mathbb{F}_2$. The past few years have witnessed the development of scalable approximate model counters such as ApproxMC [5, 6, 10]. The availability of CryptoMiniSat [15, 17], a solver with native support for XORs has been crucial for the scalability of these hashing-based techniques. The importance of CryptoMiniSat can be witnessed in Soos and Meel's recent work [16, 15] that shows runtime improvements of two to three orders of magnitude solely in the handling of CNF-XORs drastically improved the performance of the underlying model counter, ApproxMC.

Gomes et al. [11] generalize the XOR counting framework for CSPs by using linear modular constraints. Their approach, which repeatedly tests satisfiability in cells defined by randomly-generated linear modular constraints, provides lower bounds on the solution count of a given problem φ. Another approach to counting for variables over finite domains is due to Chakraborty et al. [4] in the context of SMT constraints. They proposed the idea of using a conjunction of hash functions defined over a set of distinct primes $\{p_1, p_2, \ldots p_k\}$ to ensure that one can partition the solution space into the desired number of cells M by considering the prime factorization of M.

The primary focus of our work is to showcase the potential of linear modular arithmetic constraints, not (yet) to design a scalable approximate model counter for CSPs.

Accordingly we focus on a simple procedure proposed by Chakraborty et al. [7]: Let d be the maximum size of the domain of a variable in φ and let n be the number of variables in φ. Then, let $N = d^n$. Chakraborty et al. [7] proposed the following simple algorithmic procedure that takes in a formula φ and c and returns $Y = 1$ if $|\mathsf{sol}(\varphi)| \geq c$ and returns $Y = 0$ otherwise. The procedure is guaranteed to be correct with confidence at least $1 - \delta$.

The procedure is as follows: Repeat the following $\mathcal{O}(\log 1/\delta)$ times: at iteration i, choose a hash function $h \in \mathcal{H}(N, 2^{\lceil c \rceil})$ and check if $\varphi \wedge h^{-1}(0)$ is satisfiable, then set $Z_i = 1$ else $Z_i = 0$. Now we return $Y = 1$ if the median of $\{Z_1, Z_2, \ldots Z_i \ldots\}$ is 1, else we return $Y = 0$. We refer the reader to [7] for the proof.

For the purpose of this paper, we make a simple observation that the analysis of Chakraborty et al. can be extended with respect to any bound on the number of solutions of $\varphi \wedge h^{-1}(0)$, i.e., the current analysis checks whether the number of solutions of $\varphi \wedge h^{-1}(0)$ is greater than 1 but one could substitute any fixed threshold, as is also done in the context of (ε, δ) approximate counting algorithms. We refer the reader to [12] for a longer discussion. The implementation of this scheme is simple enough to illustrate the power of our framework yet retains the core aspect of the (ε, δ)-counter, thereby allowing one to extrapolate the importance of results in the context of approximate model counting for CSPs.

In the rest of this section we present experiments using linear modular *equality* constraints in order to evaluate both GJE on a system of constraints and the dynamic programming algorithm on individual constraints, in the context of approximately counting the solutions of CSPs. All experiments were run on a cluster of dual core AMD Opteron 275 @ 2.2 GHz processors running Java SE 11 on Linux CentOS 7.6 using the MiniCP 1.0 solver. For search we branch on the parametric variables identified during GJE (since the rest are dependent on them) using variable ordering heuristic *min-domain*. Individual entries in the tables of results are the average of thirty runs.

5.1 Synthetic Problem

We conducted a controlled experiment using a CSP on n variables and ten domain values (with $p = 11$). For half of the variables, one third of them must take value 0 (modeled using an EXACTLY constraint, which is decomposed into a SUM constraint over indicator variables); for the other half, all values must be different (modeled using an ALLDIFFERENT constraint enforcing domain consistency). The clean combinatorial nature of such a CSP allows us to derive analytically the exact number of solutions without the need to enumerate them, thus making it possible to measure the accuracy of an approximate count even when the full exploration of the search space is computationally prohibitive.

Figure 1 first evaluates the impact on efficiency of some choices of threshold $\tau^{\max}$ about the number of tuples that can be included in a TABLE constraint and of using or not the filtering algorithm in Section 3.3 for individual constraints. Though instances are not identified on the plot, search tree size for a given instance does not tend to vary a lot across configurations — hence its data points appear at about the same height. We make two observations from the horizontal spread of the points for a given instance: the dynamic programming algorithm's occasional small reduction in search effort does

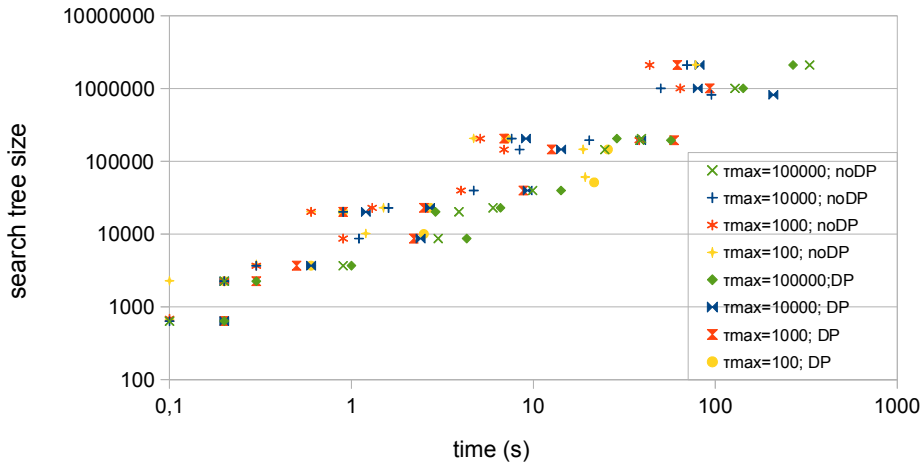

Fig. 1. Search tree size and computation time for different choices of $\tau^{\max}$ and using the dynamic programming algorithm for individual constraints ("DP" with $\upsilon^{\max} = 6$) or not. Each data point corresponds to an instance (n, m).

not make up for the frequent significant increase in computation time; a choice of $\tau^{\max} = 1000$ generally works best here. These pragmatic choices were also confirmed on several of the instances from Section 5.2. Accordingly all remaining experiments use these settings.

It is interesting to note that our more efficient implementation of GJE using a TABLE constraint leads to a conclusion that is the opposite of [11]: it is better to use GJE alone instead of the dynamic programming algorithm on each constraint and without GJE.

As mentioned before, the objective of this paper is not to build a full-blown approximate model counter but develop the underlying techniques to support such a model counter in the context of CP. Therefore, we demonstrate the effectiveness of our techniques via the approach due to Chakraborty et al. [7]. To simulate such an experiment, for a fixed number m of linear modular constraints, we seek to enumerate the solutions. It is perhaps worth recalling that the core idea of hashing-based counting is to enumerate solutions in a cell after adding a certain number of constraints, and then extrapolate the count of the original formula by scaling the count in a cell by the number of cells. In the context of hashing-based counters for SAT, one often needs to balance the tradeoff of handling cells with large number of solutions and the error in the approximation due to small cells. We seek to study whether such tradeoffs exist in CP as well, so as to allow the future developers of CP-based approximate model counters to make informed choices.

Table 1 reports the accuracy of our approximate count as we vary the number m of linear modular constraints added. As expected computation time decreases as m increases but so does the accuracy of our approximation. Nevertheless on instances with trillions of solutions we manage to produce approximate counts with a relative error under 1% in a matter of seconds. To achieve comparable accuracy we spend about

			linear modular constraints			naive
n	total #solns	m	time (s)	#solns in cell	error (%)	error (%)
10	9.92×10^8	0	17829.8	992023200	–	–
		3	52.9	745356	0.04	20.49
		4	5.3	67806	0.26	36.14
		5	0.6	6171	0.92	45.38
		6	0.1	564	3.76	52.91
		7	0.0	50	11.42	57.32
15	2.25×10^{12}	6	373.2	1269977	0.07	76.73
		7	41.1	115414	0.15	84.51
		8	6.8	10497	0.75	86.80
		9	1.2	950	2.45	99.01
		10	0.2	85	9.28	118.33
20	2.08×10^{15}	9	2379.2	883113	0.09	79.04
		10	232.5	80301	0.24	101.13
		11	54.2	7322	1.12	79.31
		12	13.9	665	3.64	94.41
		13	3.3	59	9.72	101.79

Table 1. Impact of the number of linear modular constraints m on our approximate count for synthetic instances. We also report the accuracy of a naive approach that simply fixes m variables at random.

one order of magnitude more time for an instance with three orders of magnitude more solutions in a search space that is five orders of magnitude larger. Consider as well that computing an exact count by exhaustive enumeration for as few as ten variables ($n = 10$; $m = 0$ line in the table) required almost five hours whereas an approximation with a relative error under 1% is obtained under a second.

We also report in Table 1 the accuracy of a much simpler yet naive approach to extrapolating the number of solutions enumerated in a subspace (cell): choose m variables uniformly at random and fix them to some value in their domain, also chosen uniformly at random (note that after each variable is fixed we perform constraint propagation to filter the remaining domains). We see clearly that the relative error is much larger and does not improve much as m decreases. It illustrates how the theoretical guarantees of linear modular constraints do make a difference.

These instances admit many solutions, with a ratio of the number of solutions to the size of the search space ranging from 1e-1 ($n = 10$) to 1e-5 ($n = 20$). We will see in the next section that the gain in performance may not always be as spectacular when that ratio is lower.

5.2 Benchmarks from [11]

In order to make some comparisons, we now consider the benchmark problems used in Gomes et al. [11]: the n-queens problem, DIMACS graph colouring instances, and the Spatially Balanced Latin Square problem. For each instance we set p to the smallest prime number greater or equal to the domain size.

			number of solutions		
instance	m	time (s)	in cell	extrapolated	error (%)
sbls12	0	1252	672	672	–
	1	994	53	685	1.95
	2	356	4	682	1.43
sbls14	0	170860	1968	1968	–
	1	145781	116	1968	0.00
	2	46170	7	2123	7.86
sbls15	0	2411411	13248	13248	–
	2	668312	45	13019	1.73
	3	140114	3	13101	1.11
queens13	0	98	73712	73712	–
	2	49	433	73115	0.81
	3	33	33	71767	2.64
queens15	0	3231	2279184	2279184	–
	3	918	464	2278649	0.02
	4	482	28	2360860	3.58
queens17	0	175140	95815104	95815104	–
	4	23415	1141	95319733	0.52
	5	11876	65	92432691	3.53
myciel4	0	224	142282920	1.423e+8	–
	3	79	1138181	1.423e+8	0.01
	5	36	45554	1.424e+8	0.05
	7	9	1821	1.422e+8	0.04
	9	3	73	1.417e+8	0.39
2_insertions_3	0	??	??	??	–
	11	29844	116705	5.698e+12	??
	13	8776	4662	5.691e+12	??
	15	1665	187	5.692e+12	??
	17	285	8	5.824e+12	??

Table 2. Impact of the number of linear modular constraints m on our approximate count for several benchmark problems.

Spatially Balanced Latin Squares. A Latin square of order n is an $n \times n$ matrix in which each cell is assigned one of n distinct symbols such that each row and column contains each symbol. A *spatially balanced Latin square* (SBLS) additionally requires that for each pair of symbols, the sum of their distance in each row be equal to a given constant. These find applications in experimental design. There are very few such combinatorial objects of any given order, i.e. their search space is very sparsely populated with solutions. As in [14] we consider particular *streamlined* SBLS, a subclass restricted to column order permutations of a cyclically-constructed Latin square. Our CP model uses n variables with identical domain $\{1, 2, \ldots, n\}$ to specify the order of the columns, an ALLDIFFERENT constraint over them to enforce a permutation, and combinations of SUM and TABLE contraints to enforce spatial balance. We further fix the first column in order to break some amount of symbol symmetry. Its search space is much smaller yet solutions are still very sparsely distributed: the solution-to-search-space ratio ranges

instance	vertices	edges	colours	search space	solutions	ratio
myciel4	23	71	5	5^{21}	142282920	3e-7
myciel5	47	236	6	6^{45}	??	??
2_insertions_3	37	72	4	4^{35}	??	??

Table 3. Searching the space of vertex colourings for graphs.

from 2.4e-9 to 1.2e-12 for the order-12, -14, and -15 instances we consider (there can be no solution whenever $n \equiv 1 \pmod 3$).

Table 2 reports our results on these instances. Because there are so few solutions, after adding one or two linear modular constraints the cell does not contain many solutions. As a result the computational savings are modest. Still, for the order-15 instance we obtain approximations to within 1% in under two days whereas enumerating the solutions took 28 days. Recall that the focus of [11] was to compute lower bounds with high confidence using short (i.e. on at most six variables) linear modular constraints. For instances sbls14 and sbls15 they report lower bounds of 591 and 1748 respectively, computed in a few minutes. Though correct, these significantly underestimate the true counts, 27552 and 198720, obtained by multiplying our $m = 0$ counts by n to account for fixing the first column.

n-Queens Problem. One must place n queens on an $n \times n$ chessboard so that no two queens can attack each other. As usual we model this problem using n variables and three ALLDIFFERENT constraints. The solution-to-search-space ratio ranges from 2.4e-10 to 1.2e-13, slightly lower than that of the previous problem. One can get an approximate count with relative error under 1% at a computational cost reduced by a growing factor ranging here from 2 to 7.5 (Table 2). For queens15, [11] report 3.9e+5 as lower bound whereas the true count is close to 2.3e+6.

Graph Colouring. Given an undirected graph, assign a colour from a given set to each vertex so that vertices linked by an edge bear distinct colours. Our CP model has one variable per vertex whose domain is the set of colours and a binary disequality for each edge. We considered the four instances used by [11]. Because in our case we ultimately explore an entire cell, most of these instances were out of reach: we report on instance 2_insertions_3 and on the next smaller instance from myciel5, whose characteristics are given in Table 3. Here the search space is much more densely populated with solutions but the number of variables in the model is also significantly higher. Despite breaking some colour symmetry by arbitrarily colouring both endpoints of some edge, we only managed to enumerate the solutions of myciel4. On this instance we obtain an approximation with relative error under 1% at a computational cost reduced by close to two orders of magnitude (Table 2). While we cannot measure the error on instance 2_insertions_3 our converging results suggest that the true count is near 6.83e+13 ($4 \times 3 \times$ 5.69e+12, factoring in the pre-coloured edge) and that a close approximation can be computed in under 30 minutes ($m = 15$) by enumerating solutions in one of $5^{15} \approx$ 3e+10 cells — exploring the whole search space would take much much longer. The lower bound computed in [11] is 2.3e+12.

5.3 Towards a Practical Scalable Model Counter

The encouraging empirical evaluation in the preceding section leads one to ask: *what would be needed to design a practical efficient model counter?* To this end, we believe a general recipe would be the one followed by Chakraborty et al. in their design of SMTApproxMC but a direct translation of their approach would induce linear modulo constraints over different primes. In this context, one wonders whether there is an alternate approach that can ensure all the constraints are over the same modulus. We sketch out a promising direction below by observing the construction of hash functions based on inequalities. Instead of the usage of hash functions with different primes in order to partition the solution space into the desired number of cells, we seek to use inequalities. In particular we propose hash functions such that all the items x that map to a cell α are represented using: $Ax + b \leq \alpha \pmod p$ wherein p is a prime, and A, x, b, and α are matrices of sizes $m \times n$, $n \times 1$, $m \times 1$, and $m \times 1$ respectively with entries in $[0, p-1]$. Let $\alpha[i]$ represent the value of the i-th coordinate of α. We now state the desired properties of pairwise independence:

Lemma 1. *For $x, y \in [p]^n$, we have*

$$\Pr[Ax + b \leq \alpha] = \frac{\prod_m(\alpha[i]+1)}{p^m} \tag{2}$$

$$\Pr[Ax + b \leq \alpha \mid Ay + b \leq \alpha] = \frac{\prod_m(\alpha[i]+1)}{p^m} \tag{3}$$

Proof. Chakraborty et al. [4] showed

$$\Pr[Ax + b = \alpha] = \Pr[Ax + b = \alpha \mid Ay + b = \alpha] = \frac{1}{p^m} \tag{4}$$

For $u, v \in [p]^n$, we define $u \prec v$ if for all i, $u[i] \leq v[i]$.

$$\begin{aligned}\Pr[Ax + b \leq \alpha] &= \Pr[\bigcup_{\beta \prec \alpha} Ax + b = \beta] \\ &= \Pr[Ax + b = 0] \prod_m (\alpha[i]+1) = \frac{\prod_m(\alpha[i]+1)}{p^m}\end{aligned}$$

Similarly, we have

$$\Pr[Ax + b \leq \alpha \mid Ay + b \leq \alpha] = \Pr[\bigcup_{\beta \prec \alpha} Ax + b = \beta \mid Ay + b \leq \alpha] \quad = \frac{\prod_m(\alpha[i]+1)}{p^m}$$

□

The expected number of solutions is $\frac{|sol(\varphi)| \times \prod_m(\alpha[i]+1)}{p^m}$. Since $\alpha[i] \in [0, p-1]$, similar to the case of random XORs, there exists an appropriate assignment to $\alpha[i]$ such that the expected number of solutions is in the desired range.

6 Conclusion and Future Outlook

Motivated by the recent surge of interest in applications based on model counting in the SAT domain and the concurrent development of efficient hashing-based model counting, we examined the key enabling factors for such a development. We observed that the availability of solvers with native support for hashing constraints was a crucial contributing factor to the aforementioned development. In the context of CSPs, the hashing constraints with pairwise independence can be represented by linear modular arithmetic constraints. We provided an efficient implementation of such constraints in a CP solver, reversing previous choices of approach, and demonstrated their usefulness for model counting but also for other applications such as checksums. Our empirical evaluation highlighted the potential computational savings it can bring as well as the tradeoffs that should be taken into account when developing hashing-based techniques for approximate model counting on CSPs.

From our experiments in Section 5, despite our success in being able to reach close approximate counts at a fraction of the computational cost, we currently see two obstacles to the widespread use of hashing-based techniques for model counting in CP. The first is when the total number of solutions s is relatively small with respect to p (e.g. for SBLS): m must be smaller than $\log_p s$ to expect the resulting cell to contain solutions so if that quantity does not exceed two or three we cannot gain much speedup. The approach outlined in Section 5.3 may remove that first obstacle. The second is when the number of variables n is large (e.g. for graph coloring): $\log_p s$ may be large enough for us to add many linear modular constraints but having $n - m$ parametric variables may still be too many to fix before any GJE filtering can occur (recall threshold $\tau^{\max}$) and so the process remains time consuming even though in principle we are limiting our search to a single cell. This relates more generally to the lack of filtering opportunities for linear modular constraints on a large number of variables, as mentioned in Section 3: propagation will only appear late in the search tree, once enough variables have been instantiated.

Follow-up work in the short term includes building on this work to implement approximate model counting schemes, and improving the filtering capability of our GJE algorithm by replacing our simple $\tau^{\max}$ threshold by a more sophisticated mechanism and possibly by introducing smart tables [21] in order to attempt earlier propagation in the search tree.

The broader objective of this paper is to initiate discussion among the CP community on the development of solvers with native support for linear modular arithmetic constraints. Akin to the SAT community where the initial framework proposed by Soos et al. [17] in CryptoMiniSat received widespread attention and subsequent studies improved the framework considerably, we hope the same would hold true with respect to our work.

Acknowledgements

We thank the anonymous reviewers for their constructive criticism which helped us improve the original version of the paper. Financial support for this research was provided

in part by NSERC Discovery Grant 218028/2017 and by National Research Foundation Singapore under its NRF Fellowship Programme [NRF-NRFFAI1-2019-0004].

References

1. Sébastien Bardin, Philippe Herrmann, and Florian Perroud. An Alternative to SAT-Based Approaches for Bit-Vectors. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 16th International Conference, TACAS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6015 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 2010.
2. Nicolas Beldiceanu, Mats Carlsson, Sophie Demassey, and Thierry Petit. Global Constraint Catalogue: Past, Present and Future. *Constraints An Int. J.*, 12(1):21–62, 2007.
3. J Lawrence Carter and Mark N Wegman. Universal classes of hash functions. In *ACM Symposium on Theory of Computing*, pages 106–112. ACM, 1977.
4. Supratik Chakraborty, Kuldeep S. Meel, Rakesh Mistry, and Moshe Y. Vardi. Approximate Probabilistic Inference via Word-Level Counting. In *Proc. of AAAI*, 2016.
5. Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. A Scalable Approximate Model Counter. In *Proc. of CP*, pages 200–216, 2013.
6. Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. Algorithmic Improvements in Approximate Counting for Probabilistic Inference: From Linear to Logarithmic SAT Calls. In *Proc. of IJCAI*, 2016.
7. Supratik Chakraborty, Kuldeep S. Meel, and Moshe Y. Vardi. On the Hardness of Probabilistic Inference Relaxations. In *Proc. of AAAI*, 2019.
8. Zakaria Chihani, Bruno Marre, François Bobot, and Sébastien Bardin. Sharpening Constraint Programming Approaches for Bit-Vector Theory. In Domenico Salvagnin and Michele Lombardi, editors, *Integration of AI and OR Techniques in Constraint Programming - 14th International Conference, CPAIOR 2017, Padua, Italy, June 5-8, 2017, Proceedings*, volume 10335 of *Lecture Notes in Computer Science*, pages 3–20. Springer, 2017.
9. Jordan Demeulenaere, Renaud Hartert, Christophe Lecoutre, Guillaume Perez, Laurent Perron, Jean-Charles Régin, and Pierre Schaus. Compact-Table: Efficiently Filtering Table Constraints with Reversible Sparse Bit-Sets. In Michel Rueher, editor, *Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings*, volume 9892 of *Lecture Notes in Computer Science*, pages 207–223. Springer, 2016.
10. Carla P. Gomes, Ashish Sabharwal, and Bart Selman. Model counting: A new strategy for obtaining good bounds. In *Proc. of AAAI*, volume 21, pages 54–61, 2006.
11. Carla P. Gomes, Willem Jan van Hoeve, Ashish Sabharwal, and Bart Selman. Counting CSP Solutions Using Generalized XOR Constraints. In *AAAI*, pages 204–209. AAAI Press, 2007.
12. Kuldeep S. Meel and S. Akshay. Sparse Hashing for Scalable Approximate Model Counting: Theory and Practice. In *Proceedings of Logic in Computer science (LICS)*, 7 2020.
13. Laurent D. Michel and Pascal Van Hentenryck. Constraint Satisfaction over Bit-Vectors. In Michela Milano, editor, *Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*, volume 7514 of *Lecture Notes in Computer Science*, pages 527–543. Springer, 2012.
14. Casey Smith, Carla P. Gomes, and Cèsar Fernández. Streamlining Local Search for Spatially Balanced Latin Squares. In Leslie Pack Kaelbling and Alessandro Saffiotti, editors, *IJCAI-05, Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence, Edinburgh, Scotland, UK, July 30 - August 5, 2005*, pages 1539–1540. Professional Book Center, 2005.

15. Mate Soos, Stephan Gocht, and Kuldeep S. Meel. Tinted, Detached, and Lazy CNF-XOR solving and its Applications to Counting and Sampling. In *Proceedings of International Conference on Computer-Aided Verification (CAV)*, 7 2020.
16. Mate Soos and Kuldeep S. Meel. BIRD: Engineering an Efficient CNF-XOR SAT Solver and its Applications to Approximate Model Counting. In *Proceedings of AAAI Conference on Artificial Intelligence (AAAI)*, 1 2019.
17. Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT Solvers to Cryptographic Problems. In *SAT*, volume 5584 of *Lecture Notes in Computer Science*, pages 244–257. Springer, 2009.
18. Michael A. Trick. A Dynamic Programming Approach for Consistency and Propagation for Knapsack Constraints. *Annals OR*, 118(1-4):73–84, 2003.
19. Salil P Vadhan et al. Pseudorandomness. *Foundations and Trends® in Theoretical Computer Science*, 7(1–3):1–336, 2012.
20. Leslie G. Valiant. The complexity of enumeration and reliability problems. *SIAM Journal on Computing*, 8(3):410–421, 1979.
21. Hélène Verhaeghe, Christophe Lecoutre, Yves Deville, and Pierre Schaus. Extending Compact-Table to Basic Smart Tables. In J. Christopher Beck, editor, *Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, volume 10416 of *Lecture Notes in Computer Science*, pages 297–307. Springer, 2017.