Abacus: A New Hybrid Encoding for SAT Problems

Claudia Vasconcellos-Gaete, Vincent Barichard, Frédéric Lardeux

Abacus: A New Hybrid Encoding for SAT Problems

Claudia Vasconcellos-Gaete, Vincent Barichard and Frédéric Lardeux
Laboratoire d'Etude et de Recherche en Informatique d'Angers, LERIA,
EA 2645, SFR MathSTIC, UNIV Angers,
Faculté des Sciences, 2 Bd Lavoisier, 49045 Angers, FRANCE
Email: {claudia.vasconcellos, vincent.barichard, frederic.lardeux}@univ-angers.fr

***Abstract*—Encoding an instance of a Constraint Satisfaction Problem (CSP) into a Propositional Satisfiability Problem (SAT) instance is usually a good way to benefit from the highly efficient SAT solvers. However, an encoding may not be suitable for all the constraints in the model, because it produces a large instance, the modeling of a constraint is complicated or, it does not propagate adequately.**

In general, a good encoding should aim to improve resolution effectiveness (independently of the solver chosen) and produce an instance of a reasonable size. Standard CSP-to-SAT encodings usually focus on certain aspects like straightforwardness, compactness, or good performance for specific constraints. Hybrid encodings combine encodings to obtain the best from each one of them.

This article presents the Abacus Encoding, a new hybrid encoding that combines Log and Order encodings and provides a good trade-off between the instance size and the resolution effectiveness. Like the Chinese abacus, this encoding represents integer values as the addition of units and tens. Units are set with Log encoding and tens with Order encoding. This approach allows a compact representation of values, and it is easily adaptable to improve solving efficiency.

I. Introduction

Modeling a problem as a Constraint Satisfaction Problem (CSP) is usually quite intuitive but, despite the significant progress in recent years [**?**], [**?**], the resolution of some constraints is still problematic. Other paradigms, such as the Propositional Satisfiability (SAT) [**?**], uses highly efficient solvers [**?**], [**?**] but at expenses of the complexity, which is often transferred to the modeling. Indeed, writing the SAT model of a given problem is unnatural and typically a source of errors.

In order to exploit the simplicity of modeling in CSP and the solving power of SAT solvers, many authors have proposed model transformations between CSP and SAT [**?**], [**?**], [**?**]. In SAT, the efficiency of solvers is well known, but the quality of models is nevertheless essential. There are several possible encodings. Some of them, as the Log encoding [**?**], [**?**], focus on compactness, but they tend to inhibit solvers capabilities as their propagation capacities. Others, like the Order [**?**], [**?**] or the Direct [**?**] encodings, promote the propagation during the search but to the detriment of the instance size.

During the last years, the necessity of a hybrid encoding providing instances of reasonable size and fairly efficient resolution has emerged. Several hybrid encodings have been proposed mixing log and support encoding [**?**], [**?**], [**?**], [**?**] or Log and Order encoding [**?**]. Nowadays, these hybrid encodings are either very close to Log encoding or, they propose a duplication of some parts of the model using two encodings (one favoring size and the other an efficient resolution).

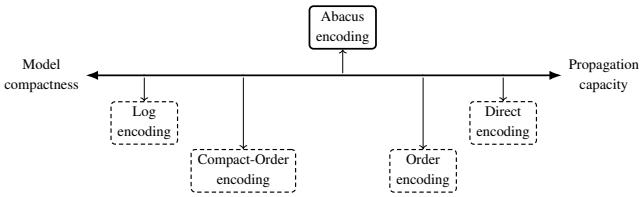

Fig. 1. Some encodings perform well on compactness and others on propagation capacity. The Abacus encoding is a good trade-off between both.

We propose a new hybrid encoding providing a trade-off between the size of the instance and the solvers efficiency. It is a *log-order* encoding named Abacus Encoding. The name comes from the Chinese abacus, the ancient calculation tool in which vertical rods represent digits (units, tens, or hundreds) and carries and shifting are similar to the decimal number system. At first glance, the Abacus encoding might look like the Compact Order encoding [**?**], which combines Order encoding and Log encoding, but the Abacus encoding uses Order encoding in a different way. The Compact Order encoding produces smaller instances, but it cannot propagate so effectively.

The article is organized as follows: Section II summarizes some of the current SAT encodings, including some Hybrid encodings as well. Section III presents the definition of the Abacus Encoding, its parameters, and the relation between Log, Abacus, and Order encodings. Section IV presents the rules to encode the addition using the Abacus Encoding along with a preliminary experiment over equation systems. Section V reports and compares the results of our encoding for the Golomb Rulers and Magic Square problems. Finally, in Section VI, we present the main conclusions and propose some future work.

II. SAT Encodings

Encoding a CSP model into SAT involves modeling each integer variable and constraints using the created Boolean variables created without losing the meaning of the CSP constraint itself. In this section, we summarize some of the well known SAT encodings, and we take a glance at hybrid encodings, describing their main characteristics.

1) Direct encoding: Proposed by De Kleer [**?**], this is a straightforward SAT encoding. For each integer variable X with $Dom(X) = \{d_0 \dots d_n\}$ it creates n propositional variables x_i representing an assignment $X = d_i$. To ensure that exactly one value is chosen for each integer variable represented, the encoding adds *at-least-one* clauses ($\bigvee_0^n x_i$) and *at-most-one* clauses ($\bigwedge_{i<j}(\neg x_i \vee \neg x_j)$). In case of prohibited values ($X \neq d_i$), unary clauses of type $\neg x_i$ are posted. For two conflicting variables ($X \neq Y$), clauses of type ($\neg x_i \vee \neg y_i$) are introduced.

2) Support Encoding: The Support Encoding encodes variables just like in the Direct encoding but for constraints, except for *at-least-one* and *at-most-one* constraints, it encodes the support of the values (instead of the conflicts). Assigning a value to a variable in a constraint does not prohibit certain values for the other variables in the constraint (like in Direct encoding), but, on the contrary, it only allows certain values. For two conflicting variables ($X \neq Y$), clauses of type ($\neg x_i \vee (\bigvee_{j \neq i} y_j)$) are introduced.

3) Log encoding: Based on the machine representation, this is a compact encoding that uses $\log_2(n)$ propositional variables to represent an integer variable as a combination of power of 2 values. In this encoding, each Boolean variable x^k is true if the k-th bit of X is true. For negative values, the complement-2 representation is used. For prohibited values, it adds clauses of type $\neg(x^0 \wedge x^1 \dots \wedge x^b)$.

4) Order encoding: Proposed to model scheduling problems [**?**], this is an efficient encoding for handling linear constraints [**?**], [**?**], [**?**]. Unlike the Direct and Log encodings (where variables represent an assignment), variables in Order encoding represent a relation $X \leq d_i$. For each integer variable, the encoding uses $n-1$ propositional variables x_i where the order relation between variables is given by a set of clauses $x_{i-1} \rightarrow x_i$. For the treatment of prohibited values $X \neq d_i$, the encoding uses the same type of clauses as in the Log encoding.

5) Hybrid Encodings: The basic principle of hybrid encodings is to combine two (or more) encodings in such a way that single features can be increased; there is not a single way to combine these encodings because it depends on each particular case. Some of these encodings are:

a) Direct-Support: As the Direct and Support Encodings represent integer variables in the same way, this hybrid encoding focuses on constraints, applying the encoding that provides the smaller possible output [**?**]. In general terms, it uses Direct encoding to encode conflicts (like in $X \neq Y$) and Support Encoding to encode supports (such as in $X = Y$). For inequalities like $X < Y$, the encoding considers the number of variables affected to determine the best encoding to use, always aiming to minimize the size of the output produced.

b) Direct-Order: This encoding [**?**] applies the Direct and Order encoding over the alldifferent() global constraint, considering its proven efficiency for certain problems. When an alldifferent() constraint is found, all the related variables are encoded in the Direct and Order encodings and then linked with translation clauses $x_i^o \leftrightarrow (\neg x_{i-1}^d \wedge x_i^d)$. For the other constraints, specialized encodings are used.

c) Log-Support encoding: Proposed by Gavanelli in [**?**], it uses a logarithmic number of variables, like the Log encoding. Propagation is improved by replacing conflict clauses of size $2\lceil \log_2 n \rceil$ by support clauses of size $\lceil \log_2 n \rceil + 1$.

d) Compact Order encoding: It belongs to the *log-order* encoding family [**?**]. Using a base $B \geq 2$, an integer variable X is represented by $\sum_{i=0}^{\lceil \log_B d \rceil} B^i x^{(i)}$ where d is the maximum value in the *Dom*(X) and $0 \leq x^{(i)} \leq B$. Each $x^{(i)}$ is then encoded in the Order encoding. This encoding reduces the size of the generated instances and propagates on the most significant digits.

III. Abacus Encoding

We propose a new encoding based on the abacus, the famous eponymous Chinese computation tool whose principle is to decompose a value into units and tens.

A. Definition

For an integer variable, the Abacus encoding decomposes its value in two parts that we call *units* and *tens*; the choice of names permits to be consistent with decimal system vocabulary. For the units, we apply the Log encoding, and for the tens, we use the Order encoding.

The choice of the combination of encodings (units Log-encoded and tens Order-encoded) intends to favor the trade-off between propagation and encoding size. The cases with units Order-encoded ("Order/Order" or "Order/Log"), produce less compact representations due to the number of variables required for units. The case "Log/Log", is compact but propagation would be affected similarly to the standard Log encoding.

To determine which part corresponds to units or tens, we add a parameter B called *encoding base*. Considering the use of the Log encoding, this base should be a value in the power of two $B \in \{1, 2, 4, 8, 16 \dots\}$.

$$\underbrace{x^{b^+} \dots x^{b^-}}_{\text{order enc.}} \quad \underbrace{x_{\log_2(B)-1} \dots x_1 x_0}_{\text{log enc.}} \qquad \begin{matrix}\text{Tens} & \text{Units}\end{matrix}$$

1) Encoding: For an integer variable X with a domain $Dom(X) = [lb..ub]$, the Abacus encoding (in base B) is:

$$x^{b^+} \; .. \; x^{b^-} \; x_{u-1} \; .. \; x_0 \tag{1}$$

$$b^+ = \left\lfloor \frac{ub}{B} \right\rfloor, \quad b^- = \left\lceil \frac{lb}{B} \right\rceil, \quad u = \log_2 B$$

In this form, the encoding over-represents the domain of a variable as it considers all values between $B \times (b^- - 1)$ and $B \times (b^+ + 1) - 1$. To represent the domain exactly as it is, we introduce an *offset* parameter (Section III-A2) along with additional clauses to avoid forbidden values (Section III-A4). Note that for negative values, negative tens are generated.

2) Offset: The use of offsets limits the number of Boolean variables required to encode a variable, avoiding the over-representation of domains. Each variable can have its own offset according to its domain. The value of the offset must be a multiple of the encoding base B to ensure no gaps in the domain represented.

For a variable X with $Dom(X) = [lb..ub]$ and encoded in base B, we define the offset as:

$$o^X = lb - lb \bmod B \tag{2}$$

We can notice that only the offset has to be taken into account when there are no tens. The use of offsets thus allows one to save one variable (tens of index 0) and imposes that the tens start at index 1 whatever the domain. There will be $b^{\#} = \lfloor \frac{ub - o^X}{B} \rfloor$ Boolean variables for the tens.

Example 1. Let $B = 8$ be the base, and variable X be a variable in the domain $[-20 \dots 33]$. The encoding with offset is as follows:

	tens							*units*			
	x^7	x^6	x^5	x^4	x^3	x^2	x^1	x_2	x_1	x_0	
value	56	48	40	32	24	16	8	4	2	1	offset
-20	0	0	0	0	0	0	0	1	0	0	-24
33	1	1	1	1	1	1	1	0	0	1	-24

3) Identifying the Value of the Tens: A consequence of using offsets is that variables do not always start at the same tens. To overcome this, we defined the functions $val(X, i)$ to get the tens value associated to an index and, $ind(X, v)$ to return the index where a certain ten is located.

$$val(X, i) = i \times B + o^X \tag{3}$$

$$ind(X, v) = \left\lceil \frac{(v - o^X)}{B} \right\rceil \tag{4}$$

4) Forbidden Values: To avoid forbidden values because of over-representation of domains, we introduce clauses of type $\neg(x^{\#} \wedge \dots \wedge x^1 \wedge x_{\log_2(B)-1} \wedge \dots \wedge x_0)$ based on the maximum domain that can be represented. For instance, in Example 1 the domain is $[-20..33]$ but it is possible to encode a domain up to $[-24..39]$ as well.

For a variable X with domain $[lb..ub]$, encoded in base B, we denote the maximum domain that can be represented as $MaxDom(X) = [L..U]$, where:

$$L = o^X \tag{5}$$

$$U = (b^{\#} + 1) \times B + o^X - 1 \tag{6}$$

Therefore, it is necessary to forbid all values in $[L..lb - 1]$ and $[ub + 1..U]$. Using an appropriate offset, the number of forbidden values should be less than $2B - 1$.

5) Ensuring consistency: We recall that Order encoding requires only one pair of consecutive bits with two different values. This means that for all Boolean variables encoding the same integer variable, we impose clauses to ensure that higher tens are always bigger or equal than the preceding tens.

$$\bigwedge_{i=2}^{b_x^{\#}} (x^i \rightarrow x^{i-1}) \tag{7}$$

$$\forall_{i > b_x^{\#}}\ x^i = \bot \qquad \forall_{i \leq 0}\ x^i = \top \tag{8}$$

where $b_x^{\#}$ is the maximum tens for the variable X, $\bot$ is the logical *false* value and $\top$ to the logical *true* value.

6) Encode and decode an Abacus representation: To encode an integer value $X \in [lb..ub]$ with Abacus encoding in base B, we decompose the value into offset (o^X), units (x_i) and tens (x^i).

$$\begin{aligned} &\text{offset:} && o^X = lb - lb \bmod B \\ &\text{units:} && x_i = \frac{X}{2^i} \% B \quad \text{with } i \in [0.. \log_2 B - 1] \\ &\text{tens:} && x^i = i \times B + o^X \quad \text{with } i \in [1..b^{\#}] \end{aligned}$$

To decode an integer value represented by the Abacus encoding, the computation is as follows[1]:

$$\begin{aligned} X = & \, max_{i \in [1..b^{\#}]}(x^i \times i \times B) \\ & + o^x + \sum_{i \in [0.. \log_2 B - 1]} (x_i \times 2^i) \end{aligned} \tag{9}$$

7) Link between Log, Order and Abacus encodings: Note that with a base $B = 1$, the Abacus encoding is similar to the Order encoding with offset. Furthermore, if the base is greater than the maximum value to represent, then the Abacus encoding corresponds to the Log encoding with offset.

IV. Encoding Addition

Using classical encodings for the addition requires choosing between to generate a small instance where very few propagations are possible, or to generate an instance for which propagation is efficient at the expense of a size that is often prohibitive for solvers. As the Abacus encoding provides a good trade-off between these two options it is therefore perfectly appropriate to this constraint.

A. Addition of Units

As unit variables are Log-encoded, this addition is handled by a series of full adders (Figure 2). Each full-adder takes a pair of bits (x_i,y_i) and the incoming carry c_i (the first carry is always false $c_0 = \bot$). The CNF clauses were obtained applying Boolean algebra and Tseitin transformations to the logical gates *and*, *or* and *xor* present in the full-adder. For the $\log_2 B$ variables corresponding to the part of units, the addition constraint is encoded with the following clauses:

The Abacus encoding does not use a sign bit (like in the classical full-adder), because the unit part is always positive. In case of overflow (when the sum of the units is greater than or equal to B), we store it in a variable $c = c_{\log_2(B)}$ that will be used in the addition of tens.

[1] $max_{i \in I}(f(i))$ returns $f(i)$ such that $\forall (i, i') \in I^2,\ f(i) > f(i')$

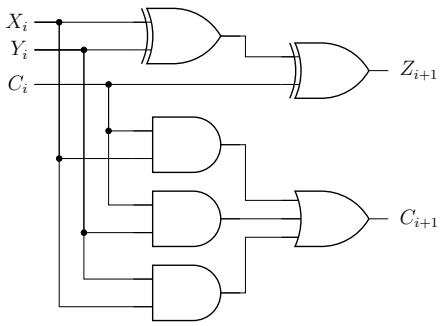

Fig. 2. Two-bit full adder schema

$$\bigwedge_{i\in[0..(\log_2 B)-1]} \begin{array}{l}\big((c_i \vee \neg z_i \vee \neg c_{i+1}) \wedge \\ (\neg c_i \vee \neg x_i \vee \neg y_i \vee z_i) \wedge \\ (\neg x_i \vee \neg y_i \vee c_{i+1}) \wedge \\ (x_i \vee y_i \vee \neg c_{i+1}) \wedge \\ (x_i \vee \neg z_i \vee \neg c_{i+1}) \wedge \\ (\neg c_i \vee z_i \vee c_{i+1}) \wedge \\ (\neg c_i \vee \neg x_i \vee c_{i+1}) \wedge \\ (c_i \vee x_i \vee \neg c_{i+1}) \wedge \\ (y_i \vee \neg z_i \vee \neg c_{i+1}) \wedge \\ (\neg x_i \vee z_i \vee c_{i+1}) \wedge \\ (\neg c_i \vee \neg y_i \vee c_{i+1}) \wedge \\ (c_i \vee y_i \vee \neg c_{i+1}) \wedge \\ (c_i \vee x_i \vee y_i \vee \neg z_i) \wedge \\ (\neg y_i \vee z_i \vee c_{i+1})\big)\end{array}$$

Fig. 3. CNF clauses for the 2-bit full adder

B. Addition of Tens

To define the encoding rules for the addition of the tens, we take as example the addition $X + Y = Z$ where X, Y, and Z are three integer variables encoded with the Abacus encoding in base B and offsets o^X, o^Y and o^Z. For the sake of simplicity, we define k as the index in Z of the partial addition computed by the indices i and j (related to X and Y, respectively).

$$\begin{aligned} k &= ind(Z, val(X,i) + val(Y,j)) \\ i_{min} &= min(0, (ind(Z, o^Z) - ind(Y, o^Y))) \\ i_{max} &= max(b_X^{\#} + 1, ind(val(Z, b_Z^{\#}) - val(Y, b_Y^{\#}))) \\ j_{min} &= min(0, (ind(Z, o^Z) - ind(X, o^X))) \\ j_{max} &= max(b_Y^{\#} + 1, ind(val(Z, b_Z^{\#}) - val(X, b_X^{\#}))) \end{aligned}$$

We recall that some variables corresponding to tens out of the domain are replaced by $\top$ or $\bot$ as indicated in the *encoding consistency* part of Section III-A5.

The encoding of the tens is entirely defined by the following four rules [2]:

$$\forall\ i \in [i_{min}..i_{max}],\ j \in [j_{min}..j_{max}]$$

$$(x^i \wedge y^j) \rightarrow z^k \tag{10}$$

$$(x^i \wedge y^j \wedge c) \rightarrow z^{k+1} \tag{11}$$

$$(\neg x^i \wedge \neg y^j) \rightarrow \neg z^k \tag{12}$$

$$(\neg x^i \wedge \neg y^j \wedge \neg c) \rightarrow \neg z^{k-1} \tag{13}$$

- Rule 10: If a ten of X and a ten of Y are true, then the ten of Z corresponding to the sum of the two true tens must be true.
- Rule 11: Same as Rule 10, but considering the overflow coming from the addition of units.
- Rule 12: If a ten of X and a ten of Y are false, then the ten of Z corresponding to the sum of the two false tens must be false.
- Rule 13: The same as Rule 12, but considering the overflow which is coming from the addition of units.

Note that these rules may generate unit clauses in specific situations. The generated instance is, therefore, easily simplified using the unit propagation mechanism (UP). The unit propagation mechanism assigns the value of a literal l appearing in a clause having only one literal to make this clause true. All clauses where l appears are deleted, and those where $\neg l$ appears are simplified by removing $\neg l$ from its literals. In the following, we will present the raw instances but also their simplification by unit propagation in order to indicate the number of variables and clauses that require some work from the solvers.

C. Structural Analysis

To analyze the impact of the Abacus encoding on adder constraints, we performed a statistical study on 1000 equation systems of the form:

$$\begin{cases} X + Y = Z \\ X + Y = W \end{cases}$$

where each domain is randomly chosen in $[0..100]$. Of course, it may happen that some systems thus generated do not have any solution. Each equation system is generated with four different bases. Base 1 to simulate the Order encoding (all values are represented using the tens). Bases 4 and 8 to exploit different units and tens balance of the Abacus encoding. Finally, a base that we note max, which is the base simulating the Log encoding (all values are represented using units). This max bound can be different from one system to another, depending on the domains.

Table I summarizes the results we obtained. These instances were directly generated and then solved by a SAT solver. Results are presented in terms of their base B, their CNF structure and the propagation power. The CNF structure is expressed in number of variables (var) and clauses (*clauses*), before and after Unit Propagation (noted as *Raw* and *UP*

[2]Here, min and max correspond to the classical functions returning respectively the minimum and the maximum of the parameters.

respectively). For the Propagation Power, we use the *prop/dec* ratio which corresponds to the number of propagations over the number of decisions made by the solver. The columns *SAT* and *UNSAT* are the percentage of instances trivially (i.e., without decision) proved.

Results in Table I shows that the higher the base B is, the more the number of variables and clauses decreases. Contrarily, when the base simulates a Log encoding ($B = max$), many variables are not necessary (units of high degrees and carries for additions), and their number increases. This is also the case for clauses when more values have to be forbidden. Concerning the propagation power ($prop/dec$), we observe that the lower the base is, the higher the number of propagations per decision is. This propagation power allows the encoding in base $B = 1$ to trivially solve 57.9% of the equation systems. On the contrary, when base $B = max$, no equation system is trivially solved. This result is consistent with the fact that the Log encoding can only propagate the upper bounds of the domains, whereas the Direct encoding can propagate both the lower and upper bounds of the domains.

We rediscover the well-known properties of the Order and Log encodings: propagation power versus instance size. We see here that Abacus encoding using bases between 1 and max provides a trade-off between these two properties.

Variables and clauses removed by unit propagation increase as the base decreases. For example, for base B=1, the number of variables is divided by 2 and the number of clauses by 4, whereas for high bases, these numbers only slightly decrease. Nevertheless, the ranking observed on the raw instances remains the same on the instances reduced by unit propagation. The simplified version by unit propagation corresponds to the real encoded instance where unit clauses have been removed. Moreover, this simplification phase is automatically done by all SAT solvers.

D. Complete Propagation

One of the criteria to evaluate encodings is the power of unit propagation. This metric corresponds to the quantity of information propagated, i.e., the number of decision variables affected as a result of the application of unit propagation.

Based on the algorithm of Brain et al. [**?**] to generate automatically propagation complete encodings (PCE), we developed a simplified version of it (see Algorithm 1). We show that our Abacus encoding is PCE in the sense that it infers the maximum possible amount of information by using unit propagation.

Our algorithm browses through all partial assignments, applies unit propagation, and checks that no new information can be inferred, stopping as soon as a counter-example is found. As inputs, it takes an encoding E (the CNF formula with the Abacus encoding of a given problem) and a set of Boolean variables Σ (the set of Boolean variables from E). It returns true if the encoding is PCE and false otherwise.

For each partial assignment, we examine the variables v that cannot be inferred by unit propagation. We test if v or $\neg v$ can be added to the partial assignment using an oracle (the SAT solver) that tells us whether or not the problem is still satisfiable.

Algorithm 1 Testing a Propagation Complete Encoding

Input: Σ, E
Output: *true/false*

```
{pa: partial assignment}
for all pa ∈ Σ do
  {pa' is empty iff UP(E)(pa) = UNSAT}
  pa' ← UP(E)(pa)
  if pa' ≠ ∅ then
    for all v ∈ {x | x ∈ Σ and x ∉ pa'} do
      for all l ∈ {v, ¬v} do
        pa" ← pa' ⊓ assign(l)
        if SATSolver(E,pa') = UNSAT then
          return FALSE
        end if
      end for
    end for
  end if
end for
return TRUE
```

We tested $x + y = z$, where x, y and z are integer variables encoded with the Abacus encoding. Algorithm 1 only proves the propagation completeness of encoding for a given number of bits. To cover the majority of the basic cases, we vary on the number of bits, offsets and bases. To build a test case, we fix the number of bits, a base (the same for all variables), and an offset for each variable. Then, we post the Abacus addition constraint $x + y = z$. Table II summarizes the basic cases tested here.

Algorithm 1 proves the completeness of each tested case. It is possible to do the calculation for a different number of bits and different base and offset values, but the runtime quickly becomes prohibitive.

V. Experimental Results

To verify our assumptions, we tested the Abacus encoding on two problems, the Magic Square and the Golomb Rulers; for each of them, we provide a results table and a discussion on the experiments.

For solving the SAT instances, we use `bc_minisat_all 1.1.2` [**?**], a blocking AllSAT solver based on Minisat [**?**]. The solver was compiled with the `continuous` macro option in order to force it to search from the point where the last solution was found. This makes the study of the proposed encoding less dependent on the heuristics of the solver.

All the experiments were carried out on a computing cluster with Intel-E5-2695 CPUs and 128 GB of memory. Each run had a dedicated processor, and the execution time was limited to 2 hours.

A. Magic Square problem

The *Magic Square* (MS) is a mathematical puzzle that aims to find an assignment of different natural values $x \in [1 \dots N^2]$

TABLE I
ADDITION RESULTS

B	CNF Structure				Propagation Power		
	Raw		UP				
	var	clauses	var	clauses	prop/dec	SAT(%)	UNSAT(%)
1	102.5	6400.5	57.5	1457.8	28.9	1.7	56.2
4	39.1	538.7	27.5	236.0	13.6	0.0	51.0
8	32.5	275.9	26.4	177.2	9.8	0.0	44.2
max	42.4	614.5	38.8	589.6	7.4	0.0	0.0

TABLE II
PROPAGATION COMPLETE ENCODING TEST CASES

Bits	Bases	Offsets
4	$b = \{1, 2, 4, 8, 16\}$	$o^x = \{-b, 0, b\}$
		$o^y = \{-b, 0, b\}$
		$o^z = \{-b, 0, b\}$
5	$b = \{1, 2, 4, 8, 16, 32\}$	$o^x = \{-b, 0, b\}$
		$o^y = \{-b, 0, b\}$
		$o^z = \{-b, 0, b\}$

disposed in a $N \times N$ matrix so that the sum across the rows, columns and diagonals always results in the constant value M called the *magic number* ($M = N(N^2 + 1)/2$) [**?**].

A basic CSP model for the Magic Square is detailed in Equations 14, 15, 16, and 17. Each x_{ij} is an integer CSP variable (with a domain=[0..N^2]) corresponding to the intersection of row i and column j.

$$\forall\, i \in [1..N] \quad \sum_{j=1}^{N} x_{ij} = M \tag{14}$$

$$\forall\, j \in [1..N] \quad \sum_{i=1}^{N} x_{ij} = M \tag{15}$$

$$\sum_{i=1}^{N} x_{ii} = \sum_{i=1}^{N} x_{i(N-i+1)} = M \tag{16}$$

$$\forall\, i \in [1..N^2] \quad \texttt{AllDifferent}(x_i) \tag{17}$$

The rotations of the values in the grid produce symmetries. Our model considers four symmetry breaking constraints to establish an order relationship between the values at the corners of the square: $x_{11} < x_{N1}$, $x_{11} < x_{1N}$, $x_{11} < x_{NN}$ and $x_{1N} < x_{N1}$.

B. Golomb Rulers

The *Golomb Rulers problem* of order N and length L consists of finding a sequence of N integer values such that no two pairs of integers are the same distance apart and the largest distance between two of its is L.

A basic CSP model for the Golomb Rulers problem is detailed in Equations 18 and 19. Each x_i is an integer CSP variable with a domain [0..L].

$$x_i \in [0..L] \quad i \in [1..N]$$

$$\forall\, i, j \in [1..N], i < j, \quad x_i < x_j \tag{18}$$

$$\forall\, i \neq j \quad \texttt{AllDifferent}(x_j - x_i) \tag{19}$$

Some couples (N,L) cannot produce a correct sequence. On the contrary, when there is a sequence solution, there are always symmetrical solutions. To avoid these symmetries, the first value of the sequence is fixed at 0 ($x_1 = 0$), and the distance between the first two integers of the sequence must be less than the distance between the last two ones ($x_1 - x_0 < x_L - x_{L-1}$).

C. Results

Tables III and IV summarize the results obtained after applying the Abacus encoding to several instances of the Magic Square and Golomb Rulers. Instances are named ms$X$ and gX respectively; in both cases, the "X" denotes the order of the instance.

For each instance, we test the Log, Order and Abacus encodings. For the Abacus, we use different encoding bases (noted as *Abacus*X, with "X" indicating the base). From a structural point of view, the number of variables (var) and the number of clauses (*clauses*) is provided. From a resolution point of view (running `bc_minisat_all`), the number of solutions found during the run (sol), the propagation power (*prop/dec*) as well as the execution time in seconds (*time*) are provided; a "-" means that resolution was not completed after the 2 hours authorized time. These results are proposed for raw instances (*Raw*) but also for simplified instances (application of unit propagation until a fixed point is reached) noted UP. Best results in terms of the number of solutions (*sol*) and then execution times ($time$) are highlighted in bold for Raw and UP instances. Note that for the Golomb Rulers results (Table IV), bracketed bases indicate bases larger than the one corresponding to the Order encoding.

We observe in Tables III and IV that the instance sizes, as well as the propagation power, differ a lot from one base to another. These results are in line with those obtained in Section IV-C, despite the presence of other constraints different from the adder constraint.

The Log encoding always provides the smallest instances, but it obtains poor results in terms of propagation power. On the contrary, the Order encoding provides large instances, but the propagation power is hight.

Best results correspond to encodings providing the highest number of solutions within the given time limit and then, when all solutions are found, to those with the shortest running time. They are mainly obtained for bases that do not correspond to the Log encoding or the Order encoding. Note that on

TABLE III
MAGIC SQUARE RESULTS

Inst.	Encoding	CNF Structure				Propagation Power		
		Raw		UP				
		var	clauses	var	clauses	sol	prop/dec	time
ms4	Order	8 322	135 725	6 680	73 550	880	606	**12,14**
	Abacus4	2 670	16 315	2 300	13 984	880	246	19,23
	Abacus8	1 986	9 681	1 670	8 138	880	167	17,25
	Log	1 838	8 688	1 470	6 966	880	142	13,47
ms5	Order	31 599	730 328	26 936	425 620	18 704	1 139	-
	Abacus4	9 182	72 762	8 203	65 291	29 164	429	-
	Abacus8	5 877	32 786	5 137	28 949	19 780	298	-
	Log	4 153	19 008	3 465	16 079	**29 872**	212	-
ms6	Order	98 984	2972 594	87 730	1806 558	4 972	2 196	-
	Abacus4	26 636	257 130	24 402	231 472	**20 811**	725	-
	Abacus8	15 214	95 642	13 640	85 580	11 220	513	-
	Log	7 538	33 258	6 322	28 618	20 249	339	-
ms7	Order	267 309	9866 264	243 308	6152 704	2 630	3 012	-
	Abacus4	70 031	806 688	65 444	741 341	3 851	1 148	-
	Abacus8	38 228	277 388	35 186	254 501	**11 187**	761	-
	Log	14 404	62 148	12 401	55 379	7 645	452	-
ms8	Order	642 534	28078 259	596 000	17806 396	1 830	4 562	-
	Abacus4	167 620	2255 276	158 808	2141 176	**8 959**	1 495	-
	Abacus8	88 962	736 901	83 350	697 130	7 476	796	-
	Log	25 706	108 990	22 508	99 772	2 570	604	-

small instances (ms4 and g7), the propagation power widely counterbalances the number of variables and clauses for small bases. When the instances become larger (ms7, ms8, and g10), the high bases limit the explosion of the number of variables and clauses.

We can observe that an encoding base bigger than this corresponding to the Log encoding (instance g7 in Table IV with Abacus128) increases the number of forbidden values and then the number of variables and clauses.

Overall, the results show a direct relationship between the instance size and the encoding base. We notice that an intermediate base between 1 (Order encoding) and this corresponding to the Log encoding is a trade-off between instance size and propagation power and provides the best results.

VI. CONCLUSION AND FUTURE WORK

In this article, we presented the Abacus encoding, a new hybrid encoding to model a CSP into a SAT problem. It combines the Log and Order encodings and decomposes CSP variables into units and tens; units are Log-encoded and tens are Order-encoded. An in-depth study has been carried out on the adder constraint for this new encoding. Extensive experiments shows that this encoding is a good trade-off between the propagation power and the instance size.

The critical point in the Abacus encoding is the choice of the base, as it allows to favor either the instance size or the propagation power. Our experiments show that using an intermediate base (between the one simulating the Log encoding and the one simulating the Order encoding) allows obtaining good results. Currently, we propose to use a common base for all the variables. This is relevant in our experiments because all treated problems have CSP variables with quite similar domains. In our future work, we intend to study the possibility of using and mixing different bases depending on the CSP variables and constraints. This can be suitable for problems where variable domains are quite different. In this article, we focused on the adder constraint, but we need more constraints in order to model more varied CSP problems. Our ongoing work is about the proposal of new Abacus encoded CSP constraints. We aim to gather a set of core constraints that will be sufficient to model more CSP problems.

TABLE IV
GOLOMB RULERS RESULTS

Inst.	Encoding	CNF Structure				Propagation Power		
		Raw		UP				
		var	clauses	var	clauses	sol	prop/dec	time
g7	Order	15 343	215 715	14 476	142 605	5	439,42	1,31
	Abacus2	8 140	70 247	7 555	60 288	5	463,14	0,74
	Abacus4	4 549	26 437	4 108	23 097	5	428,53	0,58
	Abacus8	3 065	14 197	2 678	12 450	5	256,27	0,39
	Abacus16	2 484	10 519	2 115	9 165	5	159,74	**0,22**
	Abacus32	2 204	9 191	1 841	7 926	5	244,76	0,30
	Log	2 225	9 464	1 821	7 975	5	221,00	0,33
	Abacus128	2 547	10 946	2 119	9 280	5	192,54	0,20
g8	Order	33 525	517 594	32 194	346 618	1	1075,46	22,50
	Abacus2	17 515	164 484	16 606	144 602	1	851,41	7,97
	Abacus4	9 767	61 920	9 059	55 439	1	748,26	6,28
	Abacus8	5 907	28 372	5 303	25 645	1	542,14	4,06
	Abacus16	4 477	19 122	3 905	17 225	1	379,57	2,82
	Abacus32	4 019	16 668	3 453	14 927	1	389,49	**2,28**
	Abacus64	4 047	16 856	3 467	15 039	1	397,91	3,23
	Log	4 075	17 220	3 444	15 118	1	370,83	3,22
g9	Order	67 274	1098 209	65 330	743 497	1	1827,05	262,03
	Abacus2	34 802	344 275	33 452	308 014	1	1396,04	202,00
	Abacus4	18 962	125 887	17 898	114 721	1	991,63	59,95
	Abacus8	11 438	57 660	10 506	53 131	1	788,06	82,73
	Abacus16	7 694	33 064	6 832	30 454	1	572,00	**28,66**
	Abacus32	6 218	25 445	5 384	23 272	1	570,78	36,96
	Abacus64	6 254	25 688	5 404	23 428	1	565,54	37,47
	Log	6 290	26 156	5 378	23 543	1	571,65	41,83
g10	Order	125 974	2127 198	123 238	1453 386	1	3421,94	1820,52
	Abacus2	64 729	661 511	62 794	599 790	1	2331,40	2245,02
	Abacus4	34 129	231 625	32 599	213 915	1	1516,38	1314,80
	Abacus8	19 419	98 651	18 084	92 262	1	1176,07	785,63
	Abacus16	12 654	54 598	11 409	51 123	1	874,27	559,01
	Abacus32	10 429	42 727	9 205	39 819	1	780,44	**481,65**
	Abacus64	9 339	37 889	8 121	35 103	1	754,94	601,44
	Log	9 384	38 474	8 092	35 260	1	771,08	596,04
g11	Order	239 238	4363 707	235 338	2985 094	0	5402,87	7150,44
	Abacus2	122 143	1335 505	119 369	1221 555	0	3410,03	7138,35
	Abacus4	64 448	466 619	62 224	434 210	1	2488,76	7162,35
	Abacus8	36 453	196 251	34 491	185 047	**2**	1701,71	7156,77
	Abacus16	23 308	104 857	21 464	99 270	1	1273,60	7162,90
	Abacus32	16 763	68 917	14 983	64 969	0	1067,04	7153,32
	Abacus64	15 168	61 285	13 394	57 550	**2**	1056,70	7156,99
	Log	15 223	61 659	13 429	57 815	1	994,12	7160,94