Compiling Finite Domain Constraints to SAT with BEE: the Director's Cut

Michael Codish Yoav Fekete Amit Metodi

Department of Computer Science, Ben-Gurion University, Israel

Abstract

BEE is a compiler which facilitates solving finite domain constraints by encoding them to CNF and applying an underlying SAT solver. In BEE constraints are modeled as Boolean functions which propagate information about equalities between Boolean literals. This information is then applied to simplify the CNF encoding of the constraints. We term this process *equi-propagation*. A key factor is that considering only a small fragment of a constraint model at one time enables to apply stronger, and even complete reasoning to detect equivalent literals in that fragment. Once detected, equivalences propagate to simplify the entire constraint model and facilitate further reasoning on other fragments. BEE is described in several recent papers: [20], [19] and [21]. In this paper, after a quick review of BEE, we elaborate on two undocumented details of the implementation: the hybrid encoding of cardinality constraints and complete equi-propagation. We then describe on-going work aimed to extend BEE to consider binary representation of numbers.

1 Introduction

BEE (Ben-Gurion Equi-propagation Encoder) is a tool which applies to encode finite domain constraint models to CNF. BEE was first introduced in [19] and is further described in [21]. During the encoding process, BEE performs optimizations based on equi-propagation [20] and partial evaluation to improve the quality of the target CNF. BEE is implemented in (SWI) Prolog and can be applied in conjunction with any SAT solver. It can be downloaded from [18] where one can also find examples of its use. This version of BEE is configured to apply the CryptoMiniSAT solver [24] through a Prolog interface [6]. CryptoMiniSAT offers direct support for `xor` clauses, and BEE can be configured to take advantage of this feature.

A main design choice of BEE is that integer variables are represented in the unary order-encoding (see, e.g. [9, 3]) which has many nice properties when applied to small finite domains. In the *order-encoding*, an integer variable X in the domain $[0, \dots, n]$ is represented by a bit vector $X = [x_1, \dots, x_n]$. Each bit x_i is interpreted as $X \geq i$ implying that X is a monotonic non-increasing Boolean sequence. For example, the value 3 in the interval $[0, 5]$ is represented in 5 bits as $[1, 1, 1, 0, 0]$.

It is well-known that the order-encoding facilitates the propagation of bounds. Consider an integer variable $X = [x_1, \dots, x_n]$ with values in the interval $[0, n]$. To restrict X to take values in the range $[a, b]$ (for $1 \leq a \leq b \leq n$), it is sufficient to assign $x_a = 1$ and $x_{b+1} = 0$ (if $b < n$). The variables $x_{a'}$ and $x_{b'}$ for $1 \leq a' < a$ and $b < b' \leq n$ are then determined *true* and *false*, respectively, by *unit propagation*. For example, given $X = [x_1, \dots, x_9]$, assigning $x_3 = 1$ and $x_6 = 0$ propagates to give $X = [1, 1, 1, x_4, x_5, 0, 0, 0, 0]$, signifying that $dom(X) = \{3, 4, 5\}$. We observe an additional property of the order-encoding for $X = [x_1, \dots, x_n]$: its ability to specify that a variable cannot take a specific value $0 \leq v \leq n$ in its domain by equating two variables: $x_v = x_{v+1}$. This indicates that the order-encoding is well-suited not only to propagate lower and upper bounds, but also to represent integer variables with an arbitrary, finite set, domain. For example, given $X = [x_1, \ldots, x_9]$, equating $x_2 = x_3$ imposes that $X \neq 2$. Likewise $x_5 = x_6$ and $x_7 = x_8$ impose that $X \neq 5$ and $X \neq 7$. Applying these equalities to X gives, $X = [x_1, \underline{x_2, x_2}, x_4, \underline{x_5, x_5}, \underline{x_7, x_7}, x_9]$ (note the repeated literals), signifying that $dom(X) = \{0, 1, 3, 4, 6, 8, 9\}$.

The order-encoding has many additional nice features that can be exploited to simplify constraints and their encodings to CNF. To illustrate one, consider a constraint of the form $\mathtt{A} + \mathtt{B} = \mathtt{5}$ where A and B are integer values in the range between 0 and 5 represented in the order-encoding. At the bit level (in the order encoding) we have: $\mathtt{A} = [\mathtt{a}_1, \ldots, \mathtt{a}_5]$ and $\mathtt{B} = [\mathtt{b}_1, \ldots, \mathtt{b}_5]$. The constraint is satisfied precisely when $\mathtt{B} = [\neg \mathtt{a}_5, \ldots, \neg \mathtt{a}_1]$. Equi-propagation derives the equations $E = \{b_1 = \neg a_5, \ldots, b_5 = \neg a_1\}$ and instead of encoding the constraint to CNF, we apply the substitution indicated by E, and remove the constraint which is a tautology given E.

2 Compiling Constraints with BEE

BEE is a constraint modeling language similar to the subset of FlatZinc [22] relevant for finite domain constraint problems. The full language is presented in Table 1. Boolean constants "*true*" and "*false*" are viewed as (integer) values "1" and "0". Constraints are represented as (a list of) Prolog terms. Boolean and integer variables are represented as Prolog variables and may be instantiated when simplifying constraints. In Table 1, `X` and `Xs` (possibly with subscripts) denote a Boolean literal and a vector of literals, `I` (possibly with subscript) denotes an integer variable, and `c` (possibly with subscript) denotes an integer constant. On the right column of the table are brief explanations regarding the constraints. The table introduces 26 constraint templates.

Constraints (1-2) are about variable declarations: Booleans and integers. Constraint (3) expresses a Boolean as an integer value. Constraints (4-8) are about Boolean (and reified Boolean) statements. The special cases of Constraint (5) for $\mathtt{bool\_array\_or}([\mathtt{X}_1, \ldots, \mathtt{X}_n])$ and $\mathtt{bool\_array\_xor}([\mathtt{X}_1, \ldots, \mathtt{X}_n])$ facilitate the specification of clauses and of `xor` clauses (supported directly in the CryptoMiniSAT solver [24]). Constraint (8) specifies that sorting a bit pair $[\mathtt{X}_1, \mathtt{X}_2]$ (decreasing order) results in the pair $[\mathtt{X}_3, \mathtt{X}_4]$. This is a basic building block for the construction of sorting networks [4] used to encode cardinality (linear Boolean) constraints during compilation as described in [2] and [8]. Constraints (9-14) are about integer relations and operations. Constraints (15-20) are about linear (Boolean, Pseudo Boolean, and integer) operations. Constraints (21-26) are about lexical orderings of Boolean and integer arrays.

The compilation of a constraint model to a CNF using BEE goes through three phases: **(1)** Unary bit-blasting: integer variables (and constants) are represented as bit vectors in the order-encoding. **(2)** Constraint simplification: three types of actions are applied: equi-propagation, partial evaluation, and decomposition of constraints. Simplification is applied repeatedly until no rule is applicable. **(3)** CNF encoding: the best suited encoding technique is applied to the simplified constraints.

Bit-blasting is implemented through Prolog unification. Each declaration of the form $\mathtt{new\_int}(\mathtt{I}, \mathtt{c}_1, \mathtt{c}_2)$ triggers a unification $\mathtt{I} = [\mathtt{1}, \ldots, \mathtt{1}, \mathtt{X}_{\mathtt{c}_1+1}, \ldots, \mathtt{X}_{\mathtt{c}_2}]$. To ease presentation we assume that integer variables are represented in a positive interval starting from 0 but there is no such limitation in practice (BEE supports also negative integers). BEE applies ad-hoc equi-propagators. Each constraint is associated with a set of ad-hoc rules. The novelty is that the approach is not based on CNF, as in previous works (for example [15], [10], [13], and [16]), but rather driven by the bit blasted constraints that are to be encoded to CNF. For example, Figure 1 illustrates the rules for the `int_plus` constraint. For an integer $X = \langle x_1, \ldots, x_n \rangle$, we write: $X \geq i$ to denote the equation $x_i = 1$, $X < i$ to de-

Declaring Variables		
(1)	$\texttt{new\_bool(X)}$	declare Boolean $\texttt{X}$
(2)	$\texttt{new\_int}(\texttt{I}, \texttt{c}_1, \texttt{c}_2)$	declare integer $\texttt{I}$, $\texttt{c}_1 \leq \texttt{I} \leq \texttt{c}_2$
(3)	$\texttt{bool2int}(\texttt{X}, \texttt{I})$	$(\texttt{X} \Leftrightarrow \texttt{I} = 1) \wedge (\neg\texttt{X} \Leftrightarrow \texttt{I} = 0)$
Boolean (reified) Statements		$\texttt{op} \in \{\texttt{or}, \texttt{and}, \texttt{xor}, \texttt{iff}\}$
(4)	$\texttt{bool\_eq}(\texttt{X}_1, \texttt{X}_2)$ or $\texttt{bool\_eq}(\texttt{X}_1, -\texttt{X}_2)$	$\texttt{X}_1 = \texttt{X}_2$ or $\texttt{X}_1 = -\texttt{X}_2$
(5)	$\texttt{bool\_array\_op}([\texttt{X}_1, \ldots, \texttt{X}_n])$	$\texttt{X}_1\ \texttt{op}\ \texttt{X}_2 \cdots \texttt{op}\ \texttt{X}_n$
(6)	$\texttt{bool\_array\_op\_reif}([\texttt{X}_1, \ldots, \texttt{X}_n],\ \texttt{X})$	$\texttt{X}_1\ \texttt{op}\ \texttt{X}_2 \cdots \texttt{op}\ \texttt{X}_n \Leftrightarrow \texttt{X}$
(7)	$\texttt{bool\_op\_reif}(\texttt{X}_1, \texttt{X}_2,\ \texttt{X})$	$\texttt{X}_1\ \texttt{op}\ \texttt{X}_2 \Leftrightarrow \texttt{X}$
(8)	$\texttt{comparator}(\texttt{X}_1, \texttt{X}_2, \texttt{X}_3, \texttt{X}_4)$	$\texttt{sort}([\texttt{X}_1, \texttt{X}_2]) = [\texttt{X}_3, \texttt{X}_4]$
Integer relations (reified) and arithmetic	$\texttt{op} \in \{\texttt{plus}, \texttt{times}, \texttt{div}, \texttt{mod}, \texttt{max}, \texttt{min}\}$, $\texttt{op}' \in \{\texttt{plus}, \texttt{times}, \texttt{max}, \texttt{min}\}$	$\texttt{rel} \in \{\texttt{leq}, \texttt{geq}, \texttt{eq}, \texttt{lt}, \texttt{gt}, \texttt{neq}\}$
(9)	$\texttt{int\_rel}(\texttt{I}_1, \texttt{I}_2)$	$\texttt{I}_1\ \texttt{rel}\ \texttt{I}_2$
(10)	$\texttt{int\_rel\_reif}(\texttt{I}_1, \texttt{I}_2,\ \texttt{X})$	$\texttt{I}_1\ \texttt{rel}\ \texttt{I}_2 \Leftrightarrow \texttt{X}$
(11)	$\texttt{int\_array\_allDiff}([\texttt{I}_1, \ldots, \texttt{I}_n])$	$\bigwedge_{\texttt{i}<\texttt{j}} \texttt{I}_\texttt{i} \neq \texttt{I}_\texttt{j}$
(12)	$\texttt{int\_abs}(\texttt{I}_1,\ \texttt{I})$	$\lvert\texttt{I}_1\rvert = \texttt{I}$
(13)	$\texttt{int\_op}(\texttt{I}_1, \texttt{I}_2,\ \texttt{I})$	$\texttt{I}_1\ \texttt{op}\ \texttt{I}_2 = \texttt{I}$
(14)	$\texttt{int\_array\_op}'([\texttt{I}_1, \ldots, \texttt{I}_n],\ \texttt{I})$	$\texttt{I}_1\ \texttt{op}' \cdots \texttt{op}'\ \texttt{I}_n = \texttt{I}$
Linear Constraints		$\texttt{rel} \in \{\texttt{leq}, \texttt{geq}, \texttt{eq}, \texttt{lt}, \texttt{gt}\}$
(15)	$\texttt{bool\_array\_sum\_rel}([\texttt{X}_1, \ldots, \texttt{X}_n],\ \texttt{I})$	$(\Sigma\ \texttt{X}_\texttt{i})\ \texttt{rel}\ \texttt{I}$
(16)	$\texttt{bool\_array\_pb\_rel}([\texttt{c}_1, \ldots, \texttt{c}_n], [\texttt{X}_1, \ldots, \texttt{X}_n],\ \texttt{I})$	$(\Sigma\ \texttt{c}_\texttt{i} * \texttt{X}_\texttt{i})\ \texttt{rel}\ \texttt{I}$
(17)	$\texttt{bool\_array\_sum\_modK}([\texttt{X}_1, \ldots, \texttt{X}_n], \texttt{c},\ \texttt{I})$	$((\Sigma\ \texttt{X}_\texttt{i})\ \texttt{mod}\ \texttt{c}) = \texttt{I}$
(18)	$\texttt{int\_array\_sum\_rel}([\texttt{I}_1, \ldots, \texttt{I}_n],\ \texttt{I})$	$(\Sigma\ \texttt{I}_\texttt{i})\ \texttt{rel}\ \texttt{I}$
(19)	$\texttt{int\_array\_lin\_rel}([\texttt{c}_1, \ldots, \texttt{c}_n], [\texttt{I}_1, \ldots, \texttt{I}_n],\ \texttt{I})$	$(\Sigma\ \texttt{c}_\texttt{i} * \texttt{I}_\texttt{i})\ \texttt{rel}\ \texttt{I}$
(20)	$\texttt{int\_array\_sum\_modK}([\texttt{I}_1, \ldots, \texttt{I}_n], \texttt{c},\ \texttt{I})$	$((\Sigma\ \texttt{I}_\texttt{i})\ \texttt{mod}\ \texttt{c}) = \texttt{I}$
Lexical Order		
(21)	$\texttt{bool\_arrays\_lex}(\texttt{Xs}_1, \texttt{Xs}_2)$	$\texttt{Xs}_1$ precedes (leq) $\texttt{Xs}_2$ in the lex order
(22)	$\texttt{bool\_arrays\_lexLt}(\texttt{Xs}_1, \texttt{Xs}_2)$	$\texttt{Xs}_1$ precedes (lt) $\texttt{Xs}_2$ in the lex order
(23)	$\texttt{bool\_arrays\_lex\_reif}(\texttt{Xs}_1, \texttt{Xs}_2, \texttt{X})$	$\texttt{X} \Leftrightarrow \texttt{Xs}_1$ precedes (leq) $\texttt{Xs}_2$ in the lex order
(24)	$\texttt{bool\_arrays\_lexLt\_reif}(\texttt{Xs}_1, \texttt{Xs}_2, \texttt{X})$	$\texttt{X} \Leftrightarrow \texttt{Xs}_1$ precedes (lt) $\texttt{Xs}_2$ in the lex order
(25)	$\texttt{int\_arrays\_lex}(\texttt{Is}_1, \texttt{Is}_2)$	$\texttt{Is}_1$ precedes (leq) $\texttt{Is}_2$ in the lex order
(26)	$\texttt{int\_arrays\_lexLt}(\texttt{Is}_1, \texttt{Is}_2)$	$\texttt{Is}_1$ precedes (lt) $\texttt{Is}_2$ in the lex order

Table 1: Syntax of BEE Constraints.

note the equation $x_i = 0$, $X \neq i$ to denote the equation $x_i = x_{i+1}$, and $X = i$ to denote the pair of equations $x_i = 1, x_{i+1} = 0$. We view $X = \langle x_1, \ldots, x_n \rangle$ as if padded with sentinel cells such that all cells "to the left of" x_1 take value 1 and all cells "to the right of" x_n take the value 0. This facilitates the specification of the "end cases" in the formalism. The first four rules of Figure 1 capture the standard propagation behavior for interval arithmetic. The last two rules apply when one of the integers in the relation is a constant. There are symmetric cases when replacing the role of X and Y.

$c = \texttt{int\_plus}(X, Y, Z)$ where $X = \langle x_1, \ldots, x_n \rangle$, $Y = \langle y_1, \ldots, y_m \rangle$, and $Z = \langle z_1, \ldots, z_{n+m} \rangle$	
if	then propagate
$X \geq i,\ Y \geq j$	$Z \geq i + j$
$X < i,\ Y < j$	$Z < i + j - 1$
$Z \geq k,\ X < i$	$Y \geq k - i$
$Z < k,\ X \geq i$	$Y < k - i$
$X = i$	$z_{i+1} = y_1, \ldots, z_{i+m} = y_m$
$Z = k$	$x_1 = \neg y_k, \ldots, x_k = \neg y_1$

Figure 1: Ad-hoc rules for int_plus

When an equality of the form $X = L$ (between a variable and a literal or a constant) is detected, then equi-propagation is implemented by unifying X and L and applies to all occurrences of X thus propagating to other constraints involving X.

Decomposition is about replacing complex constraints (for example about arrays) with simpler constraints (for example about array elements). Consider, for instance, the constraint int_array_plus(As, Sum). It is decomposed to a list of int_plus constraints applying a straightforward divide and conquer recursive definition. At the base case, if `As=[A]` then the constraint is replaced by a constraint of the form `int_eq(A,Sum)` which equates the bits of `A` and `Sum`, or if $\texttt{As} = [\texttt{A}_1, \texttt{A}_2]$ then it is replaced by $\texttt{int\_plus}(\texttt{A}_1, \texttt{A}_2, \texttt{Sum})$. In the general case `As` is split into two halves, then constraints are generated to sum these halves, and then an additional `int_plus` constraint is introduced to sum the two sums.

CNF encoding is the last phase in the compilation of a constraint model. Each of the remaining simplified (bit-blasted) constraints is encoded directly to a CNF. These encodings are standard and similar to those applied in various tools such as Sugar [25].

3 Cardinality Constraints in BEE

Cardinality Constraints take the form $\sum\{x_1, \ldots, x_n\} \leq k$ where the x_i are Boolean literals, k is a constant, and the relation $\leq$ might be any of $\{=, <, >, \leq, \geq\}$. There is a wide body of research on the encoding of cardinality to CNF. We focus on those using sorting networks. For example, the presentations in [11], [5], and [1, 2] describe the use of odd-even sorting networks to encode pseudo Boolean and cardinality constraints to Boolean formula. We observe that for applications of this type, it suffices to apply "selection networks" [14] rather than sorting networks. Selection networks apply to select the k largest elements from n inputs. In [14], Knuth shows a simple construction of a selection network with $O(n \log^2 k)$ size whereas, the corresponding sorting network is of size $O(n \log^2 n)$. Totalizers [3] are similar to sorting networks except that the merger for two sorted sequences involves a direct encoding with $O(n^2)$ clauses instead of $O(n \log n)$ clauses. Totalizers have been shown to give better encodings when cardinality constraints are not excessively large. BEE enables the user to select encodings based on sorting networks, totalizers or a hybrid approach which is further detailed below.

Consider the constraint $\texttt{bool\_array\_sum\_eq}(\texttt{As}, \texttt{Y})$ in a context where `As` is a list of n Boolean literals and integer variable `Y` defined as $\texttt{new\_int}(\texttt{Y}, 0, \texttt{n})$. BEE applies a divide and conquer strategy. If $n = 1$, the constraint is trivial and satisfied by unifying $\texttt{Y} = \texttt{As}$. If $n = 2$ and $\texttt{As} = [\texttt{A}_1, \texttt{A}_2]$ then $\texttt{Ys} = [\texttt{Y}_1, \texttt{Y}_2]$ and the constraint is decomposed to $\texttt{comparator}(\texttt{A}_1, \texttt{A}_2, \texttt{Y}_1, \texttt{Y}_2)$. In the general case, where $n > 2$, the constraint is decomposed as follows where $\texttt{As}_1$ and $\texttt{As}_2$ are a partitioning of `As` such that $|\texttt{As}_1| = \texttt{n}_1$, $|\texttt{As}_2| = \texttt{n}_2$, and $|\texttt{n}_1 - \texttt{n}_2| \leq 1$:

$$\boxed{\texttt{bool\_array\_sum\_eq}(\texttt{As}, \texttt{Y})} \xrightarrow{\text{decompose}} \boxed{\begin{array}{ll} \texttt{new\_int}(\texttt{T}_1, 0, \texttt{n}_1), & \texttt{bool\_array\_sum\_eq}(\texttt{As}_1, \texttt{T}_1), \\ \texttt{new\_int}(\texttt{T}_2, 0, \texttt{n}_2), & \texttt{bool\_array\_sum\_eq}(\texttt{As}_2, \texttt{T}_2), \\ \texttt{int\_plus}(\texttt{T}_1, \texttt{T}_2, \texttt{Y}) & \end{array}}$$

This decomposition process continues as long as there remain `bool_array_sum_eq` and when it terminates the model contains only `comparator` and `int_plus` constraints. The interesting discussion is with regards to the `int_plus` constraints where BEE offers two options and depending on this choice the original `bool_array_sum_eq` constraint then takes the form either of a sorting network [4] or of a totalizer [3]. So, consider a constraint $\texttt{int\_plus}(\texttt{A}, \texttt{B}, \texttt{C})$ where $\texttt{A} = [\texttt{A}_1, \ldots, \texttt{A}_\texttt{m}]$, $\texttt{B} = [\texttt{B}_1, \ldots, \texttt{B}_\texttt{p}]$ and $\texttt{C} = [\texttt{C}_1, \ldots, \texttt{C}_{\texttt{m+p}}]$ represent integer variables in the order encoding. A unary adder leads to a direct encoding of the sum of two unary numbers. It involves $O(n^2)$ clauses where n is the size of the inputs and as a circuit it has "depth" 1. The encoding introduces the following clauses where $(1 \leq i \leq m)$ and $(1 \leq j \leq p)$:

- $\bigwedge_i (\texttt{A}_\texttt{i} \rightarrow \texttt{C}_\texttt{i})$
- $\bigwedge_j (\texttt{B}_\texttt{j} \rightarrow \texttt{C}_\texttt{j})$
- $\bigwedge_{i,j} (\texttt{A}_\texttt{i} \wedge \texttt{B}_\texttt{j} \rightarrow \texttt{C}_{\texttt{i+j}})$
- $\bigwedge_i (\neg\texttt{A}_\texttt{i} \rightarrow \neg\texttt{C}_{\texttt{p+i}})$
- $\bigwedge_j (\neg\texttt{B}_\texttt{j} \rightarrow \neg\texttt{C}_{\texttt{m+j}})$
- $\bigwedge_{i,j} (\neg\texttt{A}_\texttt{i} \wedge \neg\texttt{B}_\texttt{j} \rightarrow \neg\texttt{C}_{\texttt{i+j-1}})$

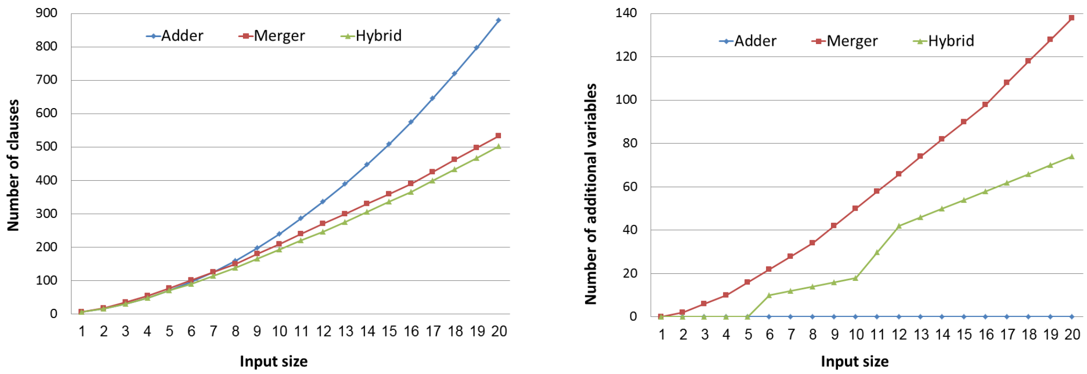


Figure 2: Relative size of CNF encodings for cardinality: adders, hybrid & mergers. On the left number of clauses, and on the right number of added variables.

An alternative encoding for $\texttt{int\_plus}(\texttt{A}, \texttt{B}, \texttt{C})$ is obtained by means of a recursive decomposition based on the so called odd-even merger from Batcher's construction [4]. It leads to an encoding with $O(n \log n)$ clauses where n is the size of the inputs and as a circuit it has "depth" $\log n$. The decomposition is as follows (ignoring the base cases) where $\texttt{A}_\texttt{o}$, $\texttt{A}_\texttt{e}$, $\texttt{B}_\texttt{o}$ and $\texttt{B}_\texttt{e}$ are partitions of $\texttt{A}$ and $\texttt{B}$ to their odd and even positioned elements, $\texttt{C}_\texttt{o}$, $\texttt{C}_\texttt{e}$ are new unary variables defined with the appropriate domains, and where $\texttt{combine}(\texttt{C}_\texttt{o}, \texttt{C}_\texttt{e}, \texttt{C})$ signifies a set of comparator constraints and is defined as $\bigwedge_i \texttt{comparator}(\texttt{C}_{\texttt{o}_{\texttt{i}+1}}, \texttt{C}_{\texttt{e}_\texttt{i}}, \texttt{C}_{\texttt{2i}}, \texttt{C}_{\texttt{2i}+1})$:

$$\boxed{\texttt{int\_plus}(\texttt{A}, \texttt{B}, \texttt{C})} \overset{\texttt{decompose}}{\Longrightarrow} \boxed{\begin{array}{l} \texttt{int\_plus}(\texttt{A}_\texttt{o}, \texttt{B}_\texttt{o}, \texttt{C}_\texttt{o}), \\ \texttt{int\_plus}(\texttt{A}_\texttt{e}, \texttt{B}_\texttt{e}, \texttt{C}_\texttt{e}), \\ \texttt{combine}(\texttt{C}_\texttt{o}, \texttt{C}_\texttt{e}, \texttt{C}) \end{array}}$$

In addition to the encodings based on unary adders (direct) and mergers (recursive decomposition), BEE offers a combination of the two which we call "hybrid". The intuition is simple: in the hybrid approach we perform recursive decomposition as for odd-even mergers, but only so long as the resulting CNF is predetermined to be smaller than the corresponding unary adder. So, it is just like a merger except that the base case is a unary adder. Before each decomposition of int_plus, BEE evaluates the benefit (in terms of CNF size) of decomposing the constraint as a merger and takes the smaller of the two.

Figure 2 depicts the size of CNF encodings for the constraint $\texttt{int\_plus}(\texttt{A}, \texttt{B}, \texttt{C})$ where $|\texttt{A}| = |\texttt{B}| = n$. The left graph illustrates the number of clauses in the three encodings. The unary adder has fewest number of clauses for inputs of size 7 or less. The hybrid encoding is always just slightly smaller than the merger. Each time a merger is decomposed to an adder it is just about of the same number of clauses. In contrast, in the right graph we see that the encoding never introduces fresh variables, and as the size of the input increases so does the benefit of the hybrid approach in number of added variables.

Now let us consider the constraint $\texttt{bool\_array\_sum\_leq}(\texttt{As}, \texttt{k})$ where As is a list of n Boolean literals and k is a constant. Assume as before that $\texttt{As}_1$ and $\texttt{As}_2$ are a partitioning of As such that $|\texttt{As}_1| = \texttt{n}_1$, $|\texttt{As}_2| = \texttt{n}_2$, and $|\texttt{n}_1 - \texttt{n}_2| \leq 1$. A naive decomposition might proceed as follows:

```
bool_array_sum_leq(As, k)  ==decompose==>  new_int(Y, 0, n),
                                           bool_array_sum_eq(As, Y),
                                           int_leq(Y, k)
```

```
==decompose==>  new_int(Y, 0, n),
                new_int(T1, 0, n1),      bool_array_sum_eq(As1, T1),
                new_int(T2, 0, n2),      bool_array_sum_eq(As2, T2),
                int_plus(T1, T2, Y),     int_leq(Y, k)
```

But we can do better. In BEE we decompose bool_array_sum_leq(As, k) as follows:

```
b._a._sum_leq(As, k)  ==decompose==>  new_int(T3, 0, k),
                                      new_int(T1, 0, min(n1, k)),   bool_array_sum_eq(As1, T1),
                                      new_int(T2, 0, min(n2, k)),   bool_array_sum_eq(As2, T2),
                                      int_plus(T3, T2, k),          int_leq(T1, T3)
```

This is correct because the constraint int_plus(T3, T2, k) defines $T_3 = k - T_2$ and so we have

$$(T_1 + T_2 \leq k) \leftrightarrow (T_1 \leq T_3) \wedge (T_2 + T_3 = k)$$

This encoding is preferable because the int_plus(T3, T2, k) constraint is encoded with 0 clauses (due to equi-propagation) and the int_leq(T1, T3) constraint in $O(k)$ clauses. Whereas in the naive version the int_plus(T1, T2, Y) is encoded in $O(n \log(n))$ or $O(n^2)$ (sorting network or direct) and the int_leq(Y, k) is encoded with 0 clauses.

4 Complete Equi-Propagation in BEE

Equi-propagation is about inferring Boolean Equalities, $x = \ell$, implied from a given CNF formula φ where x is a Boolean variable and ℓ a Boolean constant or literal. Complete equi-propagation (CEP) is about inferring all such equalities. Equi-propagation in BEE is based on ad-hoc rules and thus incomplete. However, BEE allows the user to specify, for given sets of constraints in a model, that CEP is to be applied (instead of ad-hoc equi-propagation). CEP generalizes the notion of a backbone [23]. The backbone of a CNF, φ, is the set of literals that are true in all models of φ, thus corresponding to the subset of equations, $x = \ell'$ obtained from CEP where ℓ' is a Boolean constant. Backbones prove useful in applications of SAT such as model enumeration, minimal model computation, prime implicant computation, and also in applications which involve optimization (see for example, [17]). Assigning values to backbone variables reduces the size of the search space while maintaining the meaning of the original formula. In exactly the same way, CEP identifies additional variables that can be removed from a formula, to reduce the search space, by equating pairs of literals, as in $x = y$ or $x = -y$.

Backbones are often computed by iterating with a SAT solver. In [17], the authors describe and evaluate several such algorithms and present an improved algorithm. This algorithm involves[1] exactly one unsatisfiable call to the sat solver and at most $n - b$ satisfiable calls, where n is the number of variables in φ and b the size of its backbone.

It is straightforward to apply an algorithm that computes the backbone of a CNF, φ, to perform CEP (to detect also equations between literals). Enumerating the variables of φ as $\{x_1, \ldots, x_n\}$. One simply defines

$$\varphi' = \varphi \wedge \left\{ \; e_{ij} \leftrightarrow (x_i \leftrightarrow x_j) \;\middle|\; 0 \leq i < j \leq n \; \right\} \tag{1}$$

[1] See Proposition 6 in http://sat.inesc-id.pt/~mikolas/bb-aicom-preprint.pdf.

	x_1	x_2	x_3	x_4	x_5	
θ_1	1	1	0	0	1	$\varphi_0 = \varphi$
θ_2	1	0	0	1	0	$\varphi_1 = \varphi_0 \land \neg\theta_1$
θ_3	*unsat*					$\varphi_2 = \varphi_1 \land (\neg x_1 \lor x_3)$

(a) Demo of backbone algorithm (Example 1)

	x_1	x_2	x_3	x_4	x_5	
θ_1	1	1	0	0	1	$\{x_1, x_2, x_3, x_4, x_5, 1\}$
θ_2	1	0	0	1	0	$\{x_1, x_3, 1\}, \{x_2, x_4, x_5\}$
θ_3	1	0	0	0	1	$\{x_1, x_3, 1\}, \{x_2\}, \{x_4, x_5\}$
θ_4	*unsat*					

(b) Demo of proof that CEP is linear (Example 3)

	x_1	x_2	x_3	x_4	x_5	e_{12}	e_{13}	e_{14}	e_{15}	e_{23}	e_{24}	e_{25}	e_{34}	e_{35}	e_{45}	
θ_1	1	1	0	0	1	1	0	0	1	0	0	1	1	0	0	$\varphi_0 = \varphi$
θ_2	1	0	0	1	0	0	0	1	0	1	0	1	0	1	0	$\varphi_1 = \varphi_0 \land \neg\theta_1$
θ_3	1	0	0	0	1	0	0	0	1	1	1	0	1	0	0	$\varphi_2 = \varphi_1 \land \left(\begin{array}{c}\neg x_1 \lor x_3 \lor e_{13} \lor \\ \neg e_{24} \lor \neg e_{25} \lor e_{45}\end{array}\right)$
θ_4	*unsat*															$\varphi_3 = \varphi_2 \land (\neg x_1 \lor x_3 \lor e_{13} \lor e_{45})$

(c) Demo of the CEP algorithm (Example 2)

Figure 3: Demonstrating Examples 1—3

introducing $\theta(n^2)$ fresh variables e_{ij}. If e_{ij} is in the backbone of φ' then $x_i = x_j$ is implied by φ, and if $\neg e_{ij}$ is in the backbone then $x_i = \neg x_j$ is implied. A major obstacle is that computing the backbone of φ is at least as hard as testing for the satisfiability of φ itself. Hence, for BEE, the importance of the assumption that φ is only a small fragment of the CNF of interest. Another obstacle is that the application of CEP for φ with n variables involves computing the backbone for φ' which has $\theta(n^2)$ variables.

The CEP algorithm applied in BEE is basically the same as that proposed for computing backbones in [17] extending φ to φ' as prescribed by Equation (1). We prove that iterated SAT solving for CEP using φ' involves at most $n+1$ satisfiable SAT tests, and exactly one unsatisfiable test, in spite of the fact that φ' involves a quadratic number of fresh variables.

We first describe the algorithm applied to compute the backbone of a given formula φ, which we assume is satisfiable. The algorithm maintains a table indicating for each variable x in φ for which values of x, φ can be satisfied: *true*, *false*, or both. The algorithm is initialized by calling the SAT solver with $\varphi_0 = \varphi$ and initializing the table with the information relevant to each variable: if the solution for φ_0 assigns a value to x then that value is tabled for x. If it assigns no value to x then both values are tabled for x.

The algorithm iterates incrementally. For each step $i > 0$ we add a single clause C_i (detailed below) and reinvoke the same solver instance, maintaining the learned data of the solver. This process terminates with a single unsatisfiable invocation. In words: the clause C_i can be seen as asking the solver if it is possible to flip the value for any of the variables for which we have so far seen only a single value. More formally, at each step of the algorithm, C_i is defined as follows: for each variable x, if the table indicates a single value v for x then C_i includes $\neg v$. Otherwise, if the table indicates two values for x then there is no corresponding literal in C_i. The SAT solver is then called with $\varphi_i = \varphi_{i-1} \land C_i$. If this call is satisfiable then the table is updated to record new values for variables (there must be at least one new value in the table) and we iterate. Otherwise, the algorithm terminates and the variables remaining with single entries in the table are the backbone of φ.

Example 1. *Figure 3 (a) where we assume given a formula, φ, which has models as indicated below illustrates the backbone algorithm. The first two iterations of the algorithm provide the models, θ_1 and θ_2. The next iteration illustrates that φ has no model which satisfies φ and flips* *the values of x_1 (to false) or of x_3 (to true). We conclude that x_1 and x_3 are the backbone variables of φ.*

Now consider the case where in addition to the backbone we wish to derive also equations between literals which hold in all models of φ. The CEP algorithm for φ is as follows: (1) enhance φ to φ' as specified in Equation 1, and (2) apply backbone computation to φ'. If $\varphi' \models e_{xy}$ then $\varphi \models x = y$ and if $\varphi' \models \neg e_{xy}$ then $\varphi \models x = \neg y$. As an optimization, it is possible to focus in the first two iterations only on the variables of φ.

Example 2. *Consider the same formula φ as in Example 1. This time, in the third iteration we ask to either flip the value for one of $\{x_1, x_3\}$ or for one of $\{e_{13}, e_{24}, e_{25}, e_{45}\}$ and there is such a model, θ_3. This is illustrated as Figure 3 (c)*

Theorem 1. *Let φ be a CNF, X a set of n variables, and $\Theta = \{\theta_1, \dots, \theta_m\}$ the sequence of assignments encountered by the CEP algorithm for φ and X. Then, $m \leq n + 1$.*

Before presenting a proof of Theorem 1 we introduce some terminology. Assume a set of Boolean variables X and a sequence $\Theta = \{\theta_1, \dots, \theta_m\}$ of models. Denote $\hat{X} = X \cup \{1\}$ and let $x, y \in \hat{X}$. If $\theta(x) = \theta(y)$ for all $\theta \in \Theta$ or if $\theta(x) \neq \theta(y)$ for all $\theta \in \Theta$, then we say that Θ *determines* the equation $x = y$. Otherwise, we say that Θ *disqualifies* $x = y$, intuitively meaning that Θ disqualifies $x = y$ from being determined. More formally, Θ *determines* $x = y$ if and only if $\Theta \models (x = y)$ or $\Theta \models (x = \neg y)$, and otherwise Θ *disqualifies* $x = y$.

The CEP algorithm for a formula φ and set of n variables X applies so that each iteration results in a satisfying assignment for φ which disqualifies at least one additional equation between elements of $\hat{X}$. Although there are a quadratic number of equations to be considered, we prove that the CEP algorithm terminates after at most $n + 1$ iterations.

Proof. (of Theorem 1) For each value $i \leq m$, $\Theta_i = \{\theta_1, \dots, \theta_i\}$ induces a partitioning, Π_i of $\hat{X}$ to disjoint and non-empty sets, defined such that for each $x, y \in \hat{X}$, x and y are in the same partition $P \in \Pi_i$ if and only if Θ_i determines the equation $x = y$. So, if $x, y \in P \in \Pi_i$ then the equation $x = y$ takes the same value in all assignments of Θ_i. The partitioning is well defined because if in all assignments of Θ_i both $x = y$ takes the same value and $y = z$ takes the same value, then clearly also $x = z$ takes the same value, implying that x, y, z are in the same partition of Π_i. Finally, note that each iteration $1 < i \leq m$ of the CEP algorithm disqualifies at least one equation $x = y$ that was determined by Θ_{i-1}. This implies that at least one partition of Π_{i-1} is split into two smaller (non-empty) partitions of Π_i. As there are a total of $n + 1$ elements in $\hat{X}$, there can be at most $n + 1$ iterations to the algorithm. □

Example 3. *Consider the same formula φ as in Examples 1 and 2. Figure 3 (b) illustrates the run of the algorithm in terms of the partitioning Π from the proof of Theorem 1.*

We illustrate the impact of CEP with an application where the goal is to find the largest number of edges in a simple graph with n nodes such that any cycle (length) is larger than 4. The graph is represented as a Boolean adjacency matrix A and there are two types of constraints: (1) constraints about cycles in the graph: $\forall_{i,j,k}.\ A[i,j] + A[j,k] + A[k,i] < 3$, and $\forall_{i,j,k,l}.\ A[i,j] + A[j,k] + A[k,l] + A[l,i] < 4$; and (2) constraints about symmetries: in addition to the obvious $\forall_{1 \leq i < j \leq n}.\ (A[i,j] \equiv A[j,i] \quad \text{and} \quad A[i,i] \equiv \mathit{false})$, we constrain the rows of the adjacency matrix to be sorted lexicographically (justified in [7]), and we impose lower and upper bounds on the degrees of the graph nodes as described in [12].

Table 2 illustrates results, running BEE with and without CEP. Here, we focus on finding a graph with the prescribed number of graph nodes with the known maximal number of edges (all

		with CEP				without CEP			
nodes	edges	comp.	clauses	vars	solve	comp.	clauses	vars	solve
15	26	0.24	13421	2154	0.07	0.10	23424	3321	0.08
16	28	0.26	18339	2851	0.19	0.12	30136	4328	0.34
17	31	0.39	21495	3233	0.07	0.16	37074	5125	0.12
18	34	0.49	26765	3928	0.12	0.21	45498	6070	0.13
19	38	0.46	30626	4380	0.11	0.22	54918	7024	0.15
20	41	0.55	43336	6005	5.93	0.25	68225	8507	12.70
21	44	0.77	52187	7039	1.46	0.31	81388	9835	69.46
22	47	0.88	61611	8118	71.73	0.35	96214	11276	45.43
23	50	1.10	73147	9352	35.35	0.38	113180	13101	27.54
24	54	2.02	81634	10169	96.11	0.50	130954	14712	282.99
25	57	1.40	99027	12109	438.91	0.53	152805	16706	79.11
26	61	4.58	110240	13143	217.72	0.73	175359	18615	815.55
27	65	2.16	127230	14856	35.36	0.75	201228	20791	114.55

Table 2: Search for graphs with no cycles of size 4 or less (comp. & solve times in sec.)

instances are satisfiable), and CEP is applied to the set of clauses derived from the symmetry constraints (2) detailed above. The table indicates the number of nodes, and for each CEP choice: the BEE compilation time, the number of clauses and variables, and the subsequent sat solving time. The table indicates that CEP increases the compilation time (within reason), reduces the CNF size (considerably), and (for the most) improves SAT solving time.[2]

5 Enhancing BEE for Binary Number Representation

This section describes an extension of BEE to support binary numbers. A naive extension is straightforward. There is a wide body of research specifying the bit-blasting of finite domain constraints for binary arithmetic. So, that is not the topic of this section. The interesting aspect of this exercise is how to obtain the constraint encodings together with support for equi-propagation on their bit representations. In the presentation we refer to the current version of BEE as the *unary core*, and to the extension for binary numbers as the *binary extension.* There are several possible approaches to define the binary extension:

1. CEP: A straightforward approach is to specify standard encodings for each of the new constraints in the binary extension and then to flag each of them (individually) as candidates for complete equi-propagation. In this way, as described in the previous section, BEE will infer at compile time all equi-propagations and perform the corresponding simplifications. However, the implementation of CEP involves calling a SAT solver and its application should be limited.

2. Ad-hoc rules: Another option is to introduce ad-hoc equi-propagation rules for each binary constraint similar to those already in BEE for the unary constraints (recall the example of Figure 1). However, besides being tedious, for the constraints of the binary extension there are very few relevant ad-hoc rules.

3. Decomposition to the unary kernel: In this approach we design encodings for binary constraints in terms of decompositions to unary constraints for which equi-propagation rules

[2]Experiments are performed on a single core of an Intel(R) Core(TM) i5-2400 3.10GHz CPU with 4GB memory under Linux (Ubuntu lucid, kernel 2.6.32-24-generic).

are already defined. For example, encoding the multiplication of two n-bit binary numbers decomposes to involve unary sums of at most $2n$ bits each. The unary core then performs equi-propagation on the decomposed constraints.

We describe encodings using the third approach for two constraints on binary representations: `binary_array_sum_eq` and `binary_times`. We also consider the special case where multiplication is applied to specify that $\mathtt{Z} = \mathtt{X}^2$ and demonstrate ad-hoc rules for that case.

Summing: Consider a constraint $\mathtt{binary\_array\_sum\_eq}(\mathtt{As}, \mathtt{Sum})$ where `As` is an array of binary numbers and `Sum` is the binary number representing their sum. In this context, we view `As` as a binary matrix. The rows correspond to binary numbers, and the columns to so-called buckets which are sets of bits with the same "weight" or position. The number of rows is typically not large so that it is reasonable to sum the columns using unary arithmetic. In this way the decomposition of the constraint on binary numbers relies on the underlying unary core of BEE. Assume that `As` consists of more than a single number, otherwise the decomposition is trivial. The decomposition proceeds as follows: **(1)** apply $\mathtt{transpose}(\mathtt{As}, \mathtt{Bs})$ which transposes the binary numbers in `As` to a bucket representation `Bs` (assume least significant bucket first). **(2)** introduce unary-core constraints $\mathtt{bool\_array\_sum\_eq}(\mathtt{B_i}, \mathtt{U_i})$ which sum the buckets to an array `Us` of unary numbers. **(3)** the recursively defined $\mathtt{buckets2binary}([\mathtt{U}|\mathtt{Us}], \mathtt{C}, [\mathtt{S}|\mathtt{Sum}])$ finishes the task and is defined as follows.

$$\boxed{\mathtt{buckets2binary}([\mathtt{U}|\mathtt{Us}], \mathtt{C}, [\mathtt{B}|\mathtt{Sum}])} \xrightarrow{\text{decompose}} \boxed{\begin{array}{l} \mathtt{int\_plus}(\mathtt{U}, \mathtt{C}, \mathtt{U}'), \\ \mathtt{int\_div}(\mathtt{U}', 2, \mathtt{C}'), \\ \mathtt{int\_mod}(\mathtt{U}', 2, \mathtt{B}), \\ \mathtt{buckets2binary}(\mathtt{Us}, \mathtt{C}', \mathtt{Sum}) \end{array}}$$

Here: `U` is the least significant (unary) bucket, `C` is a carry variable (unary integer, initially 0), and `B` is the least significant bit of the (binary) sum. When, eventually, the buckets are exhausted, decomposition proceeds as follows.

$$\boxed{\mathtt{buckets2binary}([\,], \mathtt{C}, [\mathtt{B}|\mathtt{Sum}])} \xrightarrow[\mathtt{C>0}]{\text{decompose}} \boxed{\begin{array}{l} \mathtt{int\_div}(\mathtt{C}, 2, \mathtt{C}'), \\ \mathtt{int\_mod}(\mathtt{C}, 2, \mathtt{B}), \\ \mathtt{buckets2binary}([\,], \mathtt{C}', \mathtt{Sum}) \end{array}}$$

Observe that, if applied without any buckets, $\mathtt{buckets2binary}([\,], \mathtt{Unary}, \mathtt{Binary})$ defines the channeling between unary and binary representations. We also note that for unary numbers, the encoding of division and modulo by 2 are efficient. Division (by 2) simply collects the even positioned bits, and modulo (2) takes advantage of the fact that the representation is "sorted".

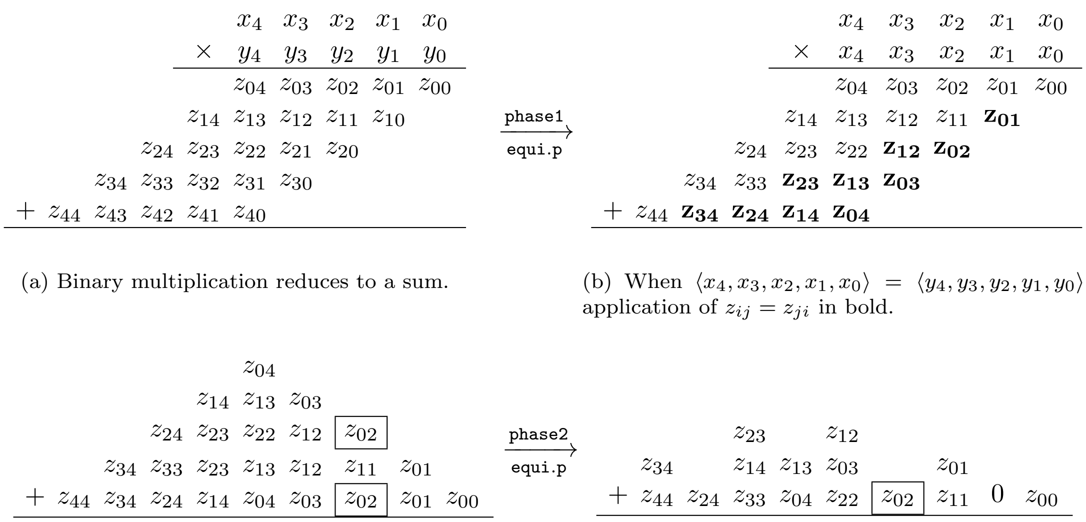
Below we evaluate our proposed encoding in BEE, but first let us introduce the encoding of binary multiplication.

Multiplying: Consider a constraint $\mathtt{binary\_times}(\mathtt{A}, \mathtt{B}, \mathtt{C})$ specifying that $\mathtt{C} = \mathtt{A} \times \mathtt{B}$. It is implemented in BEE as follows. Assume that $\mathtt{A} = [\mathtt{A_n} \dots \mathtt{A_1}]$ and $\mathtt{B} = [\mathtt{B_m} \dots \mathtt{B_1}]$ are the binary representations of A and B. Decomposition for this constraint introduces clauses defining

$$\bigwedge_{\substack{1 \leq i \leq n \\ 1 \leq j \leq m}} Z_{ij} \leftrightarrow A_i \wedge B_j \tag{2}$$

and an additional constraint $\mathtt{binary\_array\_sum\_eq}([\mathtt{Z_1}, \dots \mathtt{Z_m}], \mathtt{C})$ where for $1 \leq j \leq m$, Z_j is the binary number with bits

$$Z_{nj} \dots Z_{1j} \underbrace{0 \dots 0}_{j-1}$$



(a) Binary multiplication reduces to a sum.

(b) When $\langle x_4, x_3, x_2, x_1, x_0\rangle = \langle y_4, y_3, y_2, y_1, y_0\rangle$, application of $z_{ij} = z_{ji}$ in bold.

(c) Let the bits in each column float down.

(d) Double bits turn single and move left.

Figure 4: Decomposing the multiplication for the case of a square

The decomposition is illustrated in Figure 4(a) where rows 3–7 (with the z_{ij} variables) are binary numbers to be summed. The encoding focuses on the corresponding columns which are then encoded to sums (and carries) using the unary core of `BEE` as described above.

To evaluate the encodings of `binary_array_sum_eq(As, Sum)` and `binary_times(A, B, C)`, we consider the application of `BEE` to model and solve the n-fractions problem, also known as `CSPLIB 041`.[3] Here, one should find digit values $(1 - 9)$ for the variables in

$$\sum_{i=1}^{n} \frac{x_i}{10 * y_i + z_i} = 1$$

such that each digit value is used between 1 and $\lceil n/3 \rceil$ times. Table 3 depicts experimental results comparing two encodings of `binary_array_sum_eq(As, Sum)`: Both techniques sum the columns in the matrix `As` (where the rows are binary numbers). The *binary approach* repeatedly reduces triplets of bits in a column to a pair of bits: one in the same column (the sum bit), and one in the next (the carry bit). This is a standard "3×2" reduction. The alternative, *unary approach* is defined in terms of the unary core of `BEE`. One may note that the unary approach typically: gives slightly slower compilation times (there is more to optimize), smaller encoding sizes (equi-propagation kicks in), and significantly faster SAT solving times (it pays off) (see footnote [2] for details on machine).

Squaring: Consider the special case of multiplication `binary_times(A, A, C)` specifying that $A^2 = C$ where we introduce two additional optimizations. First, consider the variables z_{ij} introduced in Equation 2, we have $Z_{ij} = Z_{ji}$ and hence equi-propagation applies to remove the redundant variables. The result of this is illustrated in Figure 4(b). In Figure 4(c) we reorder

[3] See `http://www.cs.st-andrews.ac.uk/~ianm/CSPLib/prob/prob041/index.html`

	summing with full adders				summing in the unary core			
n	comp.	clauses	vars	sat	comp.	clauses	vars	sat
3	0.05	25492	4354	2.72	0.26	23793	4556	1.39
4	0.13	56125	9556	11.19	0.50	47743	9078	0.56
5	0.23	98712	16551	59.4	0.77	78607	14703	55.65
6	0.38	164908	27283	844.91	1.01	118850	21977	5.13
7	0.76	247082	40572	∞	1.87	164451	30125	36.83
8	1.29	363323	59183	∞	2.14	221262	40196	2653.68

Table 3: Comparison of encodings for the n-fractions problem (comp. and sat times in sec. with 4 hour timeout marked as ∞)

the bits in the columns, as if, letting the bits drop down to the baseline. Second, consider the "columns" in the `binary_array_sum_eq([Z1,... Zm], C)` constraint. Each variable of the form Z_{ij} with $i \neq j$ in a column occurs twice. So, both can be removed and one inserted back in the column to the left. This is illustrated in Figure 4(d) where we highlight the move of the two z_{02} instances. These optimizations reduce the size of the CNF and are applied both in the binary and in the unary encodings.

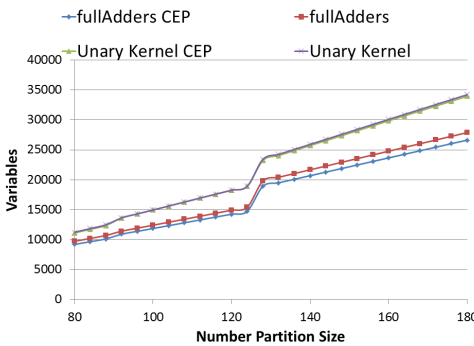
To evaluate the encoding of `binary_times(A,B,C)` for the special case when `A = B`, we consider the application of `BEE` to model and solve the number partitioning problem, also known as `CSPLIB 049`.[4] Here, one should finding a partition of numbers $\{1, \ldots, n\}$ into two sets A and B such that: A and B have the same cardinality, the sum of numbers in A equals the sum of numbers in B, and the sum of the squares of the numbers in A equals the sum of the squares of the numbers in B.

Figure 5 depicts our results. We consider four settings. The first two are the binary and unary approaches described above where buckets of bits of the same binary weight are summed using full adders or sorting networks respectively. In the other two settings, we apply complete equi-propagation per individual constraint (on binary numbers), on top of the ad-hoc rules implemented in `BEE`. Figure 5(a) illustrates the size of the encodings (number of CNF variables) for each of the four settings in terms of the instance size. The two top curves coincide and correspond to the unary encodings which create slightly larger CNFs. However note that the unary core of `BEE` with its ad-hoc (and more efficient) implementation of equi-propagation, detects all of the available equi-propagation. There is no need to apply CEP. The bottom two curves correspond to the binary encodings and illustrate that CEP detects further optimizations beyond what is detected using `BEE`.

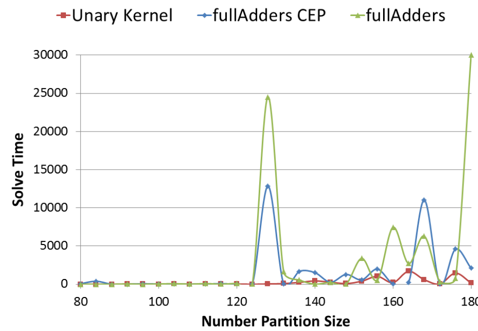
Figure 5(b) details the SAT solving times. Here we ignore the compilation times (which are high when using CEP) and focus on the quality of the obtained CNF. The graph indicates a clear advantage to the unary approach (where CEP is not even required). The average solving time using the unary approach approach (without CEP) is 270 (sec) vs 1503 (sec) using the binary approach (with CEP). This is in spite of the fact that unary approach involves larger CNF sizes.

Figures 5(c) and (d) further detail the effect of CEP in the binary and unary encodings depicting the numbers of clauses and of variables reduced by CEP in both techniques. The smaller this number, the more equi-propagation performed ad-hoc by `BEE`. In both graphs the lower curve corresponds to the encodings based on the unary core indicating that this is the one of better quality. See footnote [2] for details on machine.

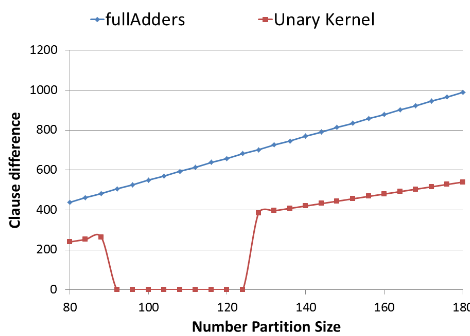
[4]See `http://www.cs.st-andrews.ac.uk/~ianm/CSPLib/prob/prob049/index.html`



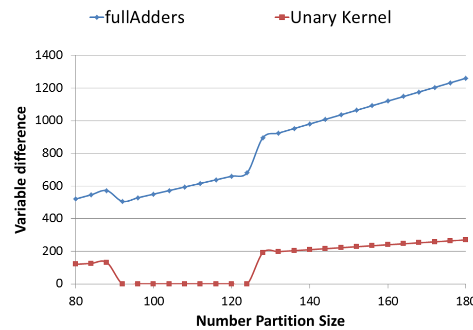
(a) Encoding sizes (number of variables): unary and binary approach, with and without CEP



(b) SAT solving time (sec.)



(c) Encoding size, CEP minus without (# clauses)



(d) Encoding size, CEP minus without (# vars)

Figure 5: Number Partitioning in BEE, encoding binary arithmetic.

6 Conclusion

We have detailed two features of BEE not described in previous publications. These concern the hybrid approach to encode cardinality constraints and the procedure for applying complete equi-propagation. We have also described our approach to enhance the unary kernel of BEE for binary numbers. Our approach is to rely as much as possible on the implementation of equi-propagation on unary numbers to build the task of equi-propagation for binary numbers. We have illustrated the power of this approach when encoding binary number multiplication. The extension of BEE for binary numbers is ongoing and still requires a thorough experimentation to evaluate its design.

References

[1] R. Asín, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell. Cardinality networks and their applications. In O. Kullmann, editor, *SAT*, volume 5584 of *Lecture Notes in Computer Science*, pages 167–180. Springer, 2009.

[2] R. Asín, R. Nieuwenhuis, A. Oliveras, and E. Rodríguez-Carbonell. Cardinality networks: a theoretical and empirical study. *Constraints*, 16(2):195–221, 2011.

[3] O. Bailleux and Y. Boufkhad. Efficient CNF encoding of Boolean cardinality constraints. In F. Rossi, editor, *CP*, volume 2833 of *LNCS*, pages 108–122, Kinsale, Ireland, 2003. Springer.

[4] K. E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, volume 32 of *AFIPS Conference Proceedings*, pages 307–314, Atlantic City, NJ, USA, 1968. Thomson Book Company, Washington D.C.

[5] M. Codish, Y. Fekete, C. Fuhs, and P. Schneider-Kamp. Optimal base encodings for pseudo-Boolean constraints. In P. A. Abdulla and K. R. M. Leino, editors, *TACAS*, volume 6605 of *Lecture Notes in Computer Science*, pages 189–204. Springer, 2011.

[6] M. Codish, V. Lagoon, and P. J. Stuckey. Logic programming with satisfiability. *TPLP*, 8(1):121–128, 2008.

[7] M. Codish, A. Miller, P. Prosser, and P. J. Stuckey. Breaking symmetries in graph representation. In *Proceedings of IJCAI*, 2013. (to appear).

[8] M. Codish and M. Zazon-Ivry. Pairwise cardinality networks. In E. M. Clarke and A. Voronkov, editors, *LPAR (Dakar)*, volume 6355 of *LNCS*, pages 154–172. Springer, 2010.

[9] J. M. Crawford and A. B. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In B. Hayes-Roth and R. E. Korf, editors, *AAAI*, volume 2, pages 1092–1097, Seattle, WA, USA, 1994. AAAI Press / The MIT Press.

[10] N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In F. Bacchus and T. Walsh, editors, *SAT*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005.

[11] N. Eén and N. Sörensson. Translating pseudo-Boolean constraints into SAT. *JSAT*, 2(1-4):1–26, 2006.

[12] D. K. Garnick, Y. H. H. Kwong, and F. Lazebnik. Extremal graphs without three-cycles or four-cycles. *Journal of Graph Theory*, 17(5):633–645, 1993.

[13] M. Heule, M. Järvisalo, and A. Biere. Efficient CNF simplification based on binary implication graphs. In K. A. Sakallah and L. Simon, editors, *SAT*, volume 6695 of *Lecture Notes in Computer Science*, pages 201–215. Springer, 2011.

[14] D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[15] C. Li. Equivalent literal propagation in the DLL procedure. *Discrete Applied Mathematics*, 130(2):251–276, 2003.

[16] N. Manthey. Coprocessor 2.0 - a flexible CNF simplifier - (tool presentation). In A. Cimatti and R. Sebastiani, editors, *SAT*, volume 7317 of *Lecture Notes in Computer Science*, pages 436–441. Springer, 2012.

[17] J. Marques-Silva, M. Janota, and I. Lynce. On computing backbones of propositional theories. In H. Coelho, R. Studer, and M. Wooldridge, editors, *ECAI*, volume 215 of *Frontiers in Artificial Intelligence and Applications*, pages 15–20. IOS Press, 2010. Extended version:`http://sat.inesc-id.pt/~mikolas/bb-aicom-preprint.pdf`.

[18] A. Metodi. BEE. `http://amit.metodi.me/research/bee/`, 2012.

[19] A. Metodi and M. Codish. Compiling finite domain constraints to SAT with BEE. *TPLP*, 12(4-5):465–483, 2012.

[20] A. Metodi, M. Codish, V. Lagoon, and P. J. Stuckey. Boolean equi-propagation for optimized SAT

encoding. In J. H.-M. Lee, editor, *CP*, volume 6876 of *LNCS*, pages 621–636, , 2011. Springer.

[21] A. Metodi, M. Codish, and P. J. Stuckey. Boolean equi-propagation for concise and efficient SAT encodings of combinatorial problems. *J. Artif. Intell. Res. (JAIR)*, 46:303–341, 2013.

[22] N. Nethercote, P. J. Stuckey, R. Becket, S. Brand, G. J. Duck, and G. Tack. Minizinc: Towards a standard CP modeling language. In C. Bessiere, editor, *CP2007*, volume 4741 of *Lecture Notes in Computer Science*, pages 529–543, Providence, RI, USA, 2007. Springer-Verlag.

[23] J. J. Schneider. Searching for backbones – an efficient parallel algorithm for the traveling salesman problem. *Comput. Phys. Commun*, 1996.

[24] M. Soos. CryptoMiniSAT, v2.5.1. `http://www.msoos.org/cryptominisat2`, 2010.

[25] N. Tamura, A. Taga, S. Kitagawa, and M. Banbara. Compiling finite linear CSP into SAT. *Constraints*, 14(2):254–272, 2009.