Optimal Base Encodings for Pseudo-Boolean Constraints*

Michael Codish[1], Yoav Fekete[1], Carsten Fuhs[2], and Peter Schneider-Kamp[3]

[1] Department of Computer Science, Ben Gurion University of the Negev, Israel
[2] LuFG Informatik 2, RWTH Aachen University, Germany
[3] IMADA, University of Southern Denmark, Denmark

Abstract. This paper formalizes the *optimal base problem*, presents an algorithm to solve it, and describes its application to the encoding of Pseudo-Boolean constraints to SAT. We demonstrate the impact of integrating our algorithm within the Pseudo-Boolean constraint solver MiniSat$^+$. Experimentation indicates that our algorithm scales to bases involving numbers up to 1,000,000, improving on the restriction in MiniSat$^+$ to prime numbers up to 17. We show that, while for many examples primes up to 17 do suffice, encoding with respect to optimal bases reduces the CNF sizes and improves the subsequent SAT solving time for many examples.

1 Introduction

The optimal base problem is all about finding an efficient representation for a given collection of positive integers. One measure for the efficiency of such a representation is the sum of the digits of the numbers. Consider for example the decimal numbers $S = \{16, 30, 54, 60\}$. The sum of their digits is 25. Taking binary representation we have $S_{(2)} = \{10000, 11110, 110110, 111100\}$ and the sum of digits is 13, which is smaller. Taking ternary representation gives $S_{(3)} = \{121, 1010, 2000, 2020\}$ with an even smaller sum of digits, 12. Considering the *mixed radix* base $B = \langle 3, 5, 2, 2\rangle$, the numbers are represented as $S_{(B)} = \{101, 1000, 1130, 10000\}$ and the sum of the digits is 9. The optimal base problem is to find a (possibly mixed radix) base for a given sequence of numbers to minimize the size of the representation of the numbers. When measuring size as "sum of digits", the base B is indeed optimal for the numbers of S. In this paper we present the optimal base problem and illustrate why it is relevant to the encoding of Pseudo-Boolean constraints to SAT. We also present an algorithm and show that our implementation is superior to current implementations.

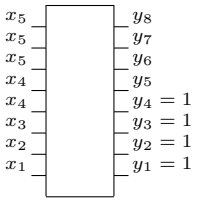
Pseudo-Boolean constraints take the form $a_1x_1+a_2x_2+\cdots+a_nx_n \geq k$, where $a_1, \ldots, a_n$ are integer coefficients, $x_1, \ldots, x_n$ are Boolean literals (i.e., Boolean variables or their negation), and k is an integer. We assume that constraints are in Pseudo-Boolean normal form [3], that is, the coefficients a_i and k are always positive and Boolean variables occur at most once in $a_1x_1 + a_2x_2 + \cdots + a_nx_n$.

* Supported by GIF grant 966-116.6 and the Danish Natural Science Research Council.

Pseudo-Boolean constraints are well studied and arise in many different contexts, for example in verification [6] and in operations research [5]. Typically we are interested in the satisfiability of a conjunction of Pseudo-Boolean constraints. Since 2005 there is a series of Pseudo-Boolean Evaluations [11] which aim to assess the state of the art in the field of Pseudo-Boolean solvers. We adopt these competition problems as a benchmark for the techniques proposed in this paper.

Pseudo-Boolean constraint satisfaction problems are often reduced to SAT. Many works describe techniques to encode these constraints to propositional formulas [1, 2, 9]. The Pseudo-Boolean solver MiniSat$^+$ ([9], cf. `http://minisat.se`) chooses between three techniques to generate SAT encodings for Pseudo-Boolean constraints. These convert the constraint to: (a) a BDD structure, (b) a network of sorters, and (c) a network of (binary) adders. The network of adders is the most concise encoding, but it has the weakest propagation properties and often leads to higher SAT solving times than the BDD based encoding, which, on the other hand, generates the largest encoding. The encoding based on sorting networks is often the one applied and is the one we consider in this paper.

To demonstrate how sorters can be used to translate Pseudo-Boolean constraints, consider the constraint $\psi = x_1 + x_2 + x_3 + 2x_4 + 3x_5 \geq 4$ where the sum of the coefficients is 8. On the right, we illustrate an 8×8 sorter where x_1, x_2, x_3 are each fed into a single input, x_4 into two of the inputs, and x_5 into three of the inputs. The outputs are $y_1, \ldots, y_8$. First, we represent the sorting network as a Boolean formula, φ, which in general, for n inputs, will be of size $O(n \log^2 n)$ [4]. Then, to assert ψ we take the conjunction of φ with the formula $y_1 \wedge y_2 \wedge y_3 \wedge y_4$.



But what happens if the coefficients in a constraint are larger than in this example? How should we encode $16x_1 + 30x_2 + 54x_3 + 30x_4 + 60x_5 \geq 87$? How should we handle very large coefficients (larger than 1,000,000)? To this end, the authors in [9] generalize the above idea and propose to decompose the constraint into a number of interconnected sorting networks. Each sorter represents a digit in a mixed radix base. This construction is governed by the choice of a suitable mixed radix base and the objective is to find a base which minimizes the size of the sorting networks. Here the optimal base problem comes in, as the size of the networks is directly related to the size of the representation of the coefficients. We consider the sum of the digits (of coefficients) and other measures for the size of the representations and study their influence on the quality of the encoding.

In MiniSat$^+$ the search for an optimal base is performed using a brute force algorithm and the resulting base is constructed from prime numbers up to 17. The starting point for this paper is the following remark from [9] (Footnote 8):

> *This is an ad-hoc solution that should be improved in the future. Finding the optimal base is a challenging optimization problem in its own right.*

In this paper we take the challenge and present an algorithm which scales to find an optimal base consisting of elements with values up to 1,000,000. We illustrate that in many cases finding a better base leads also to better SAT solving time.

Section 2 provides preliminary definitions and formalizes the optimal base problem. Section 3 describes how MiniSat$^+$decomposes a Pseudo-Boolean constraint with respect to a given mixed radix base to generate a corresponding propositional encoding, so that the constraint has a solution precisely when the encoding has a model. Section 4 is about (three) alternative measures with respect to which an optimal base can be found. Sections 5–7 introduce our algorithm based on classic AI search methods (such as cost underapproximation) in three steps: Heuristic pruning, best-first branch and bound, and base abstraction. Sections 8 and 9 present an experimental evaluation and some related work. Section 10 concludes. Proofs are given in [8].

2 Optimal Base Problems

In the classic base r radix system, positive integers are represented as finite sequences of digits $\mathbf{d} = \langle d_0, \ldots, d_k \rangle$ where for each digit $0 \leq d_i < r$, and for the most significant digit, $d_k > 0$. The integer value associated with $\mathbf{d}$ is $v = d_0 + d_1 r + d_2 r^2 + \cdots + d_k r^k$. A mixed radix system is a generalization where a base is an infinite radix sequence $B = \langle r_0, r_1, r_2, \ldots \rangle$ of integers where for each radix, $r_i > 1$ and for each digit, $0 \leq d_i < r_i$. The integer value associated with $\mathbf{d}$ is $v = d_0 w_0 + d_1 w_1 + d_2 w_2 + \cdots + d_k w_k$ where $w_0 = 1$ and for $i \geq 0$, $w_{i+1} = w_i r_i$. The sequence $weights(B) = \langle w_0, w_1, w_2, \ldots \rangle$ specifies the weighted contribution of each digit position and is called the *weight sequence of* B. A finite mixed radix base is a finite sequence $B = \langle r_0, r_1, \ldots, r_{k-1} \rangle$ with the same restrictions as for the infinite case except that numbers always have $k+1$ digits (possibly padded with zeroes) and there is no bound on the value of the most significant digit, d_k.

In this paper we focus on the representation of finite multisets of natural numbers in finite mixed radix bases. Let *Base* denote the set of finite mixed radix bases and $ms(\mathbb{N})$ the set of finite non-empty multisets of natural numbers. We often view multisets as ordered (and hence refer to their first element, second element, etc.). For a finite sequence or multiset S of natural numbers, we denote its length by $|S|$, its maximal element by $max(S)$, its i^{th} element by $S(i)$, and the multiplication of its elements by $\prod S$ (if S is the empty sequence then $\prod S = 1$). If a base consists of prime numbers only, then we say that it is a prime base. The set of prime bases is denoted $Base_p$.

Let $B \in Base$ with $|B| = k$. We denote by $v_{(B)} = \langle d_0, d_1, \ldots, d_k \rangle$ the representation of a natural number v in base B. The most significant digit of $v_{(B)}$, denoted $msd(v_{(B)})$, is d_k. If $msd(v_{(B)}) = 0$ then we say that B is redundant for v. Let $S \in ms(\mathbb{N})$ with $|S| = n$. We denote the $n \times (k+1)$ matrix of digits of elements from S in base B as $S_{(B)}$. Namely, the i^{th} row in $S_{(B)}$ is the vector $S(i)_{(B)}$. The most significant digit column of $S_{(B)}$ is the $k+1$ column of the matrix and denoted $msd(S_{(B)})$. If $msd(S_{(B)}) = \langle 0, \ldots, 0 \rangle^T$, then we say that B is redundant for S. This is equivalently characterized by $\prod B > max(S)$.

Definition 1 (non-redundant bases). *Let $S \in ms(\mathbb{N})$. We denote the set of non-redundant bases for S, $Base(S) = \left\{ B \in Base \,\middle|\, \prod B \leq max(S) \right\}$. The set of non-redundant prime bases for S is denoted $Base_p(S)$. The set of non-*

redundant (prime) bases for S, containing elements no larger than ℓ, is denoted $Base^{\ell}(S)$ ($Base^{\ell}_{p}(S)$). The set of bases in $Base(S)/Base^{\ell}(S)/Base^{\ell}_{p}(S)$, is often viewed as a tree with root $\langle\,\rangle$ (the empty base) and an edge from B to B' if and only if B' is obtained from B by extending it with a single integer value.

Definition 2 (*sum_digits*). *Let $S \in ms(\mathbb{N})$ and $B \in Base$. The sum of the digits of the numbers from S in base B is denoted $sum\_digits(S_{(B)})$.*

Example 3. The usual binary "base 2" and ternary "base 3" are represented as the infinite sequences $B_1 = \langle 2, 2, 2, \ldots\rangle$ and $B_2 = \langle 3, 3, 3, \ldots\rangle$. The finite sequence $B_3 = \langle 3, 5, 2, 2\rangle$ and the empty sequence $B_4 = \langle\,\rangle$ are also bases. The empty base is often called the "unary base" (every number in this base has a single digit). Let $S = \{16, 30, 54, 60\}$. Then, $sum\_digits(S_{(B_1)}) = 13$, $sum\_digits(S_{(B_2)}) = 12$, $sum\_digits(S_{(B_3)}) = 9$, and $sum\_digits(S_{(B_4)}) = 160$.

Let $S \in ms(\mathbb{N})$. A cost function for S is a function $cost_S : Base \to \mathbb{R}$ which associates bases with real numbers. An example is $cost_S(B) = sum\_digits(S_{(B)})$. In this paper we are concerned with the following *optimal base problem*.

Definition 4 (optimal base problem). *Let $S \in ms(\mathbb{N})$ and $cost_S$ a cost function. We say that a base B is an* optimal base for S *with respect to $cost_S$, if for all bases B', $cost_S(B) \leq cost_S(B')$. The corresponding* optimal base problem *is to find an optimal base B for S.*

The following two lemmata confirm that for the *sum_digits* cost function, we may restrict attention to non-redundant bases involving prime numbers only.

Lemma 5. *Let $S \in ms(\mathbb{N})$ and consider the sum_digits cost function. Then, S has an optimal base in $Base(S)$.*

Lemma 6. *Let $S \in ms(\mathbb{N})$ and consider the sum_digits cost function. Then, S has an optimal base in $Base_p(S)$.*

How hard is it to solve an instance of the optimal base problem (namely, for $S \in ms(\mathbb{N})$)? The following lemma provides a polynomial (in $max(S)$) upper bound on the size of the search space. This in turn suggests a pseudo-polynomial time brute force algorithm (to traverse the search space).

Lemma 7. *Let $S \in ms(\mathbb{N})$ with $m = max(S)$. Then, $\big|\, Base(S) \,\big| \leq m^{1+\rho}$ where $\rho = \zeta^{-1}(2) \approx 1.73$ and where ζ is the Riemann zeta function.*

Proof. Chor *et al.* prove in [7] that the number of ordered factorizations of a natural number n is less than n^{ρ}. The number of bases for all of the numbers in S is hence bounded by $\sum_{n \leq m} n^{\rho}$, which is bounded by $m^{1+\rho}$.

3 Encoding Pseudo-Boolean Constraints

This section presents the construction underlying the sorter based encoding of Pseudo-Boolean constraints applied in MiniSat^{+}[9]. It is governed by the choice of a mixed radix base B, the optimal selection of which is the topic of this paper. The construction sets up a series of sorting networks to encode the digits, in base

B, of the sum of the terms on the left side of a constraint $\psi = a_1x_1 + a_2x_2 + \cdots + a_nx_n \geq k$. The encoding then compares these digits with those from $k_{(B)}$ from the right side. We present the construction, step by step, through an example where $B = \langle 2, 3, 3\rangle$ and $\psi = 2x_1 + 2x_2 + 2x_3 + 2x_4 + 5x_5 + 18x_6 \geq 23$.

Step one - representation in base:

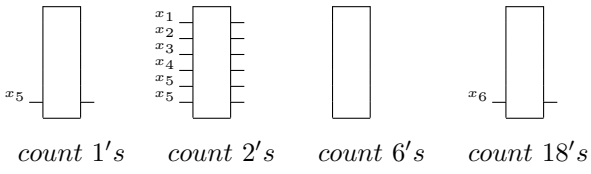
The coefficients of ψ form a multiset $S = \{2, 2, 2, 2, 5, 18\}$ and their representation in base B, a 6×4 matrix, $S_{(B)}$, depicted on the right. The rows of the matrix correspond to the representation of the coefficients in base B.

$$S_{(B)} = \begin{pmatrix} 0&1&0&0 \\ 0&1&0&0 \\ 0&1&0&0 \\ 0&1&0&0 \\ 1&2&0&0 \\ 0&0&0&1 \end{pmatrix}$$

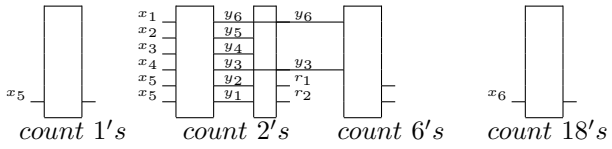
Step two - counting: Representing the coefficients as four digit numbers in base $B = \langle 2, 3, 3\rangle$ and considering the values $weights(B) = \langle 1, 2, 6, 18\rangle$ of the digit positions, we obtain a decomposition for the left side of ψ:

$$\begin{aligned} &2x_1 + 2x_2 + 2x_3 + 2x_4 + 5x_5 + 18x_6 = \\ &\quad \mathbf{1} \cdot (x_5) + \mathbf{2} \cdot (x_1 + x_2 + x_3 + x_4 + 2x_5) + \mathbf{6} \cdot (0) + \mathbf{18} \cdot (x_6) \end{aligned}$$

To encode the sums at each digit position $(1, 2, 6, 18)$, we set up a series of four sorting networks as depicted below. Given values for the variables, the sorted outputs from these networks represented unary numbers d_1, d_2, d_3, d_4 such that the left side of ψ takes the value $1 \cdot d_1 + 2 \cdot d_2 + 6 \cdot d_3 + 18 \cdot d_4$.



Step three - converting to base: For the outputs d_1, d_2, d_3, d_4 to represent the digits of a number in base $B = \langle 2, 3, 3\rangle$, we need to encode also the "carry" operation from each digit position to the next. The first 3 outputs must represent valid digits for B, i.e., unary numbers less than $\langle 2, 3, 3\rangle$ respectively. In our example the single potential violation to this restriction is d_2, which is represented in 6 bits. To this end we add two components to the encoding: (1) each third output of the second network (y_3 and y_6 in the diagram) is fed into the third network as an additional (carry) input; and (2) clauses are added to encode that the output of the second network is to be considered modulo 3. We call these additional clauses a *normalizer*. The normalizer defines two outputs $R = \langle r_1, r_2\rangle$ and introduces clauses specifying that the (unary) value of R equals the (unary) value of $d_2 \bmod 3$.



Step four - comparison: The outputs from these four units now specify a number in base B, each digit represented in unary notation. This number is now compared (via an encoding of the lexicographic order) to $23_{(B)}$ (the value from the right-hand side of ψ).

4 Measures of Optimality

We now return to the objective of this paper: For a given Pseudo-Boolean constraint, how can we choose a mixed radix base with respect to which the encoding of the constraint via sorting networks will be optimal? We consider here three alternative cost functions with respect to which an optimal base can be found. These cost functions capture with increasing degree of precision the actual size of the encodings.

The first cost function, *sum_digits* as introduced in Definition 2, provides a coarse measure on the size of the encoding. It approximates (from below) the total number of input bits in the network of sorting networks underlying the encoding. An advantage in using this cost function is that there always exists an optimal base which is prime. The disadvantage is that it ignores the carry bits in the construction, and as such is not always a precise measure for optimality. In [9], the authors propose to apply a cost function which considers also the carry bits. This is the second cost function we consider and we call it *sum_carry*.

Definition 8 (cost function: *sum_carry*). *Let $S \in ms(\mathbb{N})$, $B \in Base$ with $|B| = k$ and $S_{(B)} = (a_{ij})$ the corresponding $n \times (k+1)$ matrix of digits. Denote the sequences $\bar{s} = \langle s_0, s_1, \ldots, s_k \rangle$ (sums) and $\bar{c} = \langle c_0, c_1, \ldots, c_k \rangle$ (carries) defined by: $s_j = \sum_{i=1}^{n} a_{ij}$ for $0 \leq j \leq k$, $c_0 = 0$, and $c_{j+1} = (s_j + c_j) \text{ div } B(j)$ for $0 \leq j \leq k$. The "sum of digits with carry" cost function is defined by the equation on the right.*

$$sum\_carry(S_{(B)}) = \sum_{j=0}^{k} (s_j + c_j)$$

The following example illustrates the *sum_carry* cost function and that it provides a better measure of base optimality for the (size of the) encoding of Pseudo-Boolean constraints.

Example 9. Consider the encoding of a Pseudo-Boolean constraint with coefficients $S = \{1, 3, 4, 8, 18, 18\}$ with respect to bases: $B_1 = \langle 2, 3, 3 \rangle$, $B_2 = \langle 3, 2, 3 \rangle$, and $B_3 = \langle 2, 2, 2, 2 \rangle$. Figure 1 depicts the sizes of the sorting networks for each of these bases. The upper tables illustrate the representation of the coefficients in the corresponding bases. In the lower tables, the rows labeled "sum" indicate the number of bits per network and (to their right) their total sum which is the *sum_digits* cost. With respect to the *sum_digits* cost function, all three bases are optimal for S, with a total of 9 inputs. The algorithm might as well return B_3.

The rows labeled "carry" indicate the number of carry bits in each of the constructions and (to their right) their totals. With respect to the *sum_carry* cost function, bases B_1 and B_2 are optimal for S, with a total of $9 + 2 = 11$ bits while B_3 involves $9 + 5 = 14$ bits. The algorithm might as well return B_1.

The following example shows that when searching for an optimal base with respect to the *sum_carry* cost function, one must consider also non-prime bases.

Example 10. Consider again the Pseudo Boolean constraint $\psi = 2x_1 + 2x_2 + 2x_3 + 2x_4 + 5x_5 + 18x_6 \geq 23$ from Section 3. The encoding with respect to

$B_1 = \langle 2, 3, 3 \rangle$

S	1's	2's	6's	18's	
1	1	0	0	0	
3	1	1	0	0	
4	0	2	0	0	
8	0	1	1	0	
18	0	0	0	1	
18	0	0	0	1	
sum	2	4	1	2	9
carry	0	1	1	0	2
comp	1	9	1	1	12

$B_2 = \langle 3, 2, 3 \rangle$

S	1's	3's	6's	18's	
1	1	0	0	0	
3	0	1	0	0	
4	1	1	0	0	
8	2	0	1	0	
18	0	0	0	1	
18	0	0	0	1	
sum	4	2	1	2	9
carry	0	1	1	0	2
comp	5	3	1	1	10

$B_3 = \langle 2, 2, 2, 2 \rangle$

S	1's	2's	4's	8's	16's	
1	1	0	0	0	0	
3	1	1	0	0	0	
4	0	0	1	0	0	
8	0	0	0	1	0	
18	0	1	0	0	1	
18	0	1	0	0	1	
sum	2	3	1	1	2	9
carry	0	1	2	1	1	5
comp	1	5	3	1	3	13

Fig. 1. Number of inputs/carries/comparators when encoding $S = \{1, 3, 4, 8, 18, 18\}$ and three bases $B_1 = \langle 2, 3, 3 \rangle$, $B_2 = \langle 3, 2, 3 \rangle$, and $B_3 = \langle 2, 2, 2, 2 \rangle$.

$B_1 = \langle 2, 3, 3 \rangle$ results in 4 sorting networks with 10 inputs from the coefficients and 2 carries. So a total of 12 bits. The encoding with respect to $B_2 = \langle 2, 9 \rangle$ is smaller. It has the same 10 inputs from the coefficients but no carry bits. Base B_2 is optimal and non-prime.

We consider a third cost function which we call the *num_comp* cost function. Sorting networks are constructed from "comparators" [10] and in the encoding each comparator is modeled using six CNF clauses. This function counts the number of comparators in the construction. Let $f(n)$ denote the number of comparators in an $n \times n$ sorting network. For small values of $0 \leq n \leq 8$, $f(n)$ takes the values $0, 0, 1, 3, 5, 9, 12, 16$ and 19 respectively which correspond to the sizes of the optimal networks of these sizes [10]. For larger values, the construction uses Batcher's odd-even sorting networks [4] for which $f(n) = n \cdot \lceil \log_2 n \rceil \cdot (\lceil \log_2 n \rceil - 1)/4 + n - 1$.

Definition 11 (cost function: *num_comp*). *Consider the same setting as in Definition 8. Then,*

$$num\_comp(S_{(B)}) = \sum_{j=0}^{k} f(s_j + c_j)$$

Example 12. Consider again the setting of Example 9. In Figure 1 the rows labeled "comp" indicate the number of comparators in each of the sorting networks and their totals. The construction with the minimal number of comparators is that obtained with respect to the base $B_2 = \langle 3, 2, 3 \rangle$ with 10 comparators.

It is interesting to remark the following relationship between the three cost functions: The *sum_digits* function is the most "abstract". It is only based on the representation of numbers in a mixed radix base. The *sum_carry* function considers also properties of addition in mixed-radix bases (resulting in the carry bits). Finally, the *num_comp* function considers also implementation details of the odd-even sorting networks applied in the underlying MiniSat$^+$ construction. In Section 8 we evaluate how the alternative choices for a cost function influence the size and quality of the encodings obtained with respect to corresponding optimal bases.

5 Optimal Base Search I: Heuristic Pruning

This section introduces a simple, heuristic-based, depth-first, tree search algorithm to solve the optimal base problem. The search space is the domain of non-redundant bases as presented in Definition 1. The starting point is the brute force algorithm applied in MiniSat$^+$. For a sequence of integers S, MiniSat$^+$ applies a depth-first traversal of $Base_p^{17}(S)$ to find the base with the optimal value for the cost function $cost_S(B) = sum\_carry(S_{(B)})$.

Our first contribution is to introduce a heuristic function and to identify branches in the search space which can be pruned early on in the search. Each tree node B encountered during the traversal is inspected to check if given the best node encountered so far, $bestB$, it is possible to determine that all descendants of B are guaranteed to be less optimal than $bestB$. In this case, the subtree rooted at B may be pruned. The resulting algorithm improves on the one of MiniSat$^+$ and provides the basis for the further improvements introduced in Sections 6 and 7. We need first a definition.

Definition 13 (base extension, partial cost, and admissible heuristic). *Let $S \in ms(\mathbb{N})$, $B, B' \in Base(S)$, and $cost_S$ a cost function. We say that: (1) B' extends B, denoted $B' \succ B$, if B is a prefix of B', (2) $\partial cost_S$ is a partial cost function for $cost_S$ if $\forall B' \succ B.\ cost_S(B') \geq \partial cost_S(B)$, and (3) h_S is an admissible heuristic function for $cost_S$ and $\partial cost_S$ if $\forall B' \succ B.\ cost_S(B') \geq \partial cost_S(B') + h_S(B') \geq \partial cost_S(B) + h_S(B)$.*

The intuition is that $\partial cost_S(B)$ signifies a part of the cost of B which will be a part of the cost of any extension of B, and that $h_S(B)$ is an under-approximation on the additional cost of extending B (in any way) given the partial cost of B. We denote $cost^\alpha_S(B) = \partial cost_S(B) + h_S(B)$. If $\partial cost_S$ is a partial cost function and h_S is an admissible heuristic function, then $cost^\alpha_S(B)$ is an under-approximation of $cost_S(B)$. The next lemma provides the basis for heuristic pruning using the three cost functions introduced above.

Lemma 14. *The following are admissible heuristics for the cases when:*

1. $cost_S(B) = sum\_digits(S_{(B)})$: $\partial cost_S(B) = cost_S(B) - \sum msd(S_{(B)})$.
2. $cost_S(B) = sum\_carry(S_{(B)})$: $\partial cost_S(B) = cost_S(B) - \sum msd(S_{(B)})$.
3. $cost_S(B) = num\_comp(S_{(B)})$: $\partial cost_S(B) = cost_S(B) - f(s_{|B|} + c_{|B|})$.

In the first two settings we take $h_S(B) = \left|\{\ x \in S \,|\, x \geq \prod B\ \}\right|$.
In the case of num_comp we take the trivial heuristic estimate $h_S(B) = 0$

The algorithm, which we call `dfsHP` for depth-first search with heuristic pruning, is now stated as Figure 2 where the input to the algorithm is a `multiset` of integers `S` and the output is an optimal base. The algorithm applies a depth-first traversal of $Base(S)$ in search of an optimal base. We assume given: a cost function $cost_S$, a partial cost function $\partial cost_S$ and an admissible heuristic h_S. We denote $cost^\alpha_S(B) = \partial cost_S(B) + h_S(B)$. The abstract data type `base` has two operations: `extend(int)` and `extenders(multiset)`. For a base `B` and an

```
/*input*/  multiset S
/*init*/   base bestB = ⟨2, 2, ..., 2⟩
/*dfs*/    depth-first traverse Base(S)
           at each node B, for the next value p ∈ B.extenders(S) do
                base newB = B.extend(p)
                if (cost^α_S(newB) > cost_S(bestB)) prune
                else if (cost_S(newB) < cost_S(bestB)) bestB = newB
/*output*/ return bestB;
```

Fig. 2. `dfsHP`: depth-first search for an optimal base with heuristic pruning

integer `p`, `B.extend(p)` is the base obtained by extending `B` by `p`. For a multiset `S`, `B.extenders(S)` is the set of integer values `p` by which `B` can be extended to a non-redundant base for `S`, i.e., such that $\prod$ `B.extend(p)` $\leq max($`S`$)$. The definition of this operation may have additional arguments to indicate if we seek a prime base or one containing elements no larger than ℓ.

Initialization (/*init*/ in the figure) assigns to the variable `bestB` a finite binary base of size $\lfloor \log_2(max(S)) \rfloor$. This variable will always denote the best base encountered so far (or the initial finite binary base). Throughout the traversal, when visiting a node `newB` we first check if the subtree rooted at `newB` should be pruned. If this is not the case, then we check if a better "best base so far" has been found. Once the entire (with pruning) search space has been considered, the optimal base is in `bestB`.

To establish a bound on the complexity of the algorithm, denote the number of different integers in S by s and $m = max(S)$. The algorithm has space complexity $O(\log(m))$, for the depth first search on a tree with height bound by $\log(m)$ (an element of $Base(S)$ will have at most $\log_2(m)$ elements). For each base considered during the traversal, we have to calculate $cost_S$ which incurs a cost of $O(s)$. To see why, consider that when extending a base B by a new element giving base B', the first columns of $S_{(B')}$ are the same as those in $S_{(B)}$ (and thus also the costs incurred by them). Only the cost incurred by the most significant digit column of $S_{(B)}$ needs to be recomputed for $S_{(B')}$ due to base extension of B to B'. Performing the computation for this column, we compute a new digit for the s different values in S. Finally, by Lemma 7, there are $O(m^{2.73})$ bases and therefore, the total runtime is $O(s * m^{2.73})$. Given that $s \leq m$, we can conclude that runtime is bounded by $O(m^{3.73})$.

6 Optimal Base Search II: Branch and Bound

In this section we further improve the search algorithm for an optimal base. The search algorithm is, as before, a traversal of the search space using the same partial cost and heuristic functions as before to prune the tree. The difference is that instead of a depth first search, we maintain a priority queue of nodes for expansion and apply a best-first, branch and bound search strategy.

Figure 3 illustrates our enhanced search algorithm. We call it `B&B`. The abstract data type `priority_queue` maintains bases prioritized by the value of

```
base findBase(multiset S)
/*1*/   base bestB = ⟨2, 2, ..., 2⟩; priority_queue Q = { ⟨ ⟩ };
/*2*/   while (Q ≠ {} && cost^α_S(Q.peek()) < cost_S(bestB))
/*3*/     base B = Q.popMin();
/*4*/     foreach (p ∈ B.extenders(S))
/*5*/       base newB = B.extend(p);
/*6*/       if (cost^α_S(newB) ≤ cost_S(bestB))
/*7*/         { Q.push(newB); if (cost_S(newB) < cost_S(bestB)) bestB = newB; }
/*8*/   return bestB;
```

Fig. 3. Algorithm B&B: best-first, branch and bound

$cost^{\alpha}_S$. Operations `popMin()`, `push(base)` and `peek()` (peeks at the minimal entry) are the usual. The reason to box the text "`priority_queue`" in the figure will become apparent in the next section.

On line /*1*/ in the figure, we initialize the variable `bestB` to a finite binary base of size $\lfloor \log_2(max(S)) \rfloor$ (same as in Figure 2) and initialize the queue to contain the root of the search space (the empty base). As long as there are still nodes to be expanded in the queue that are potentially interesting (line /*2*/), we select (at line /*3*/) the best candidate base `B` from the frontier of the tree in construction for further expansion. Now the search tree is expanded for each of the relevant integers (calculated at line /*4*/). For each child `newB` of `B` (line /*5*/), we check if pruning at `newB` should occur (line /*6*/) and if not we check if a better bound has been found (line /*7*/) Finally, when the loop terminates, we have found the optimal base and return it (line /*8*/).

7 Optimal Base Search III: Search Modulo Product

This section introduces an abstraction on the search space, classifying bases according to their product. Instead of maintaining (during the search) a priority queue of all bases (nodes) that still need to be explored, we maintain a special priority queue in which there will only ever be at most one base with the same product. So, the queue will never contain two different bases B_1 and B_2 such that $\prod B_1 = \prod B_2$. In case a second base, with the same product as one already in, is inserted to the queue, then only the base with the minimal value of $cost^{\alpha}_S$ is maintained on the queue. We call this type of priority queue a *hashed priority queue* because it can conveniently be implemented as a hash table.

The intuition comes from a study of the *sum_digits* cost function for which we can prove the following **Property 1** on bases: Consider two bases B_1 and B_2 such that $\prod B_1 = \prod B_2$ and such that $cost^{\alpha}_S(B_1) \leq cost^{\alpha}_S(B_2)$. Then for any extension of B_1 and of B_2 by the same sequence C, $cost_S(B_1C) \leq cost_S(B_2C)$. In particular, if one of B_1 or B_2 can be extended to an optimal base, then B_1 can. A direct implication is that when maintaining the frontier of the search space as a priority queue, we only need one representative of the class of bases which have the same product (the one with the minimal value of $cost^{\alpha}_S$).

A second **Property 2** is more subtle and true for any cost function that has the first property: Assume that in the algorithm described as Figure 3 we at some stage remove a base B_1 from the priority queue. This implies that if in the future we encounter any base B_2 such that $\prod B_1 = \prod B_2$, then we can be sure that $cost_S(B_1) \leq cost_S(B_2)$ and immediately prune the search tree from B_2.

Our third and final algorithm, which we call `hashB&B` (best-first, branch and bound, with hash priority queue) is identical to the algorithm presented in Figure 3, except that the the boxed priority queue introduced at line /*1*/ is replaced by a `hash_priority_queue`.

The abstract data type `hash_priority_queue` maintains bases prioritized by the value of $cost^{\alpha}_S$. Operations `popMin()` and `peek()` are as usual. Operation `push(`B_1`)` works as follows: (a) if there is no base B_2 in the queue such that $\prod B_1 = \prod B_2$, then add B_1. Otherwise, (b) if $cost^{\alpha}_S(B_2) \leq cost^{\alpha}_S(B_1)$ then do not add B_1. Otherwise, (c) remove B_2 from the queue and add B_1.

Theorem 15.
*(1) The sum_digits cost function satisfies **Property 1**; and (2) the `hashB&B` algorithm finds an optimal base for any cost function which satisfies **Property 1**.*

We conjecture that the other cost functions do not satisfy **Property 1**, and hence cannot guarantee that the `hashB&B` algorithm always finds an optimal base. However, in our extensive experimentation, all bases found (when searching for an optimal prime base) are indeed optimal.

A direct implication of the above improvements is that we can now provide a tighter bound on the complexity of the search algorithm. Let us denote the number of different integers in S by s and $m = max(S)$. First note that in the worst case the hashed priority queue will contain m elements (one for each possible value of a base product, which is never more than m). Assuming that we use a Fibonacci Heap, we have a $O(\log(m))$ cost (amortized) per `popMin()` operation and in total a $O(m * \log(m))$ cost for popping elements off the queue during the search for an optimal base.

Now focus on the cost of operations performed when extracting a base B from the queue. Denoting $\prod B = q$, B has at most m/q children (integers which extend it). For each child we have to calculate $cost_S$ which incurs a cost of $O(s)$ and possibly to insert it to the queue. Pushing an element onto a hashed priority queue (in all three cases) is a constant time operation (amortized), and hence the total cost for dealing with a child is $O(s)$.

Finally, consider the total number of children created during the search which corresponds to the following sum:

$$O(\sum_{q=1}^{m} m/q) = O(m \sum_{q=1}^{m} 1/q) = O(m * \log(m))$$

So, in total we get $O(m * \log(m)) + O(m * \log(m) * s) \leq O(m^2 * \log(m))$. When we restrict the extenders to be prime numbers then we can further improve this bound to $O(m^2 * \log(\log(m)))$ by reasoning about the density of the primes. A proof can be found in [8].

8 Experiments

Experiments are performed using an extension to MiniSat$^+$ [9] where the only change to the tool is to plug in our optimal base algorithm. The reader is invited to experiment with the implementation via its web interface.[4] All experiments are performed on a Quad-Opteron 848 at 2.2 GHz, 16 GB RAM, running Linux.

Our benchmark suite originates from 1945 Pseudo-Boolean Evaluation [11] instances from the years 2006–2009 containing a total of 74,442,661 individual Pseudo-Boolean constraints. After normalizing and removing constraints with $\{0, 1\}$ coefficients we are left with 115,891 different optimal base problems where the maximal coefficient is $2^{31} - 1$. We then focus on 734 PB instances where at least one optimal base problem from the instance yields a base with an element that is non-prime or greater than 17. When solving PB instances, in all experiments, a 30 minute timeout is imposed as in the Pseudo-Boolean competitions. When solving an optimal base problem, a 10 minute timeout is applied.

Experiment 1 (Impact of optimal bases): The first experiment illustrates the advantage in searching for an optimal base for Pseudo-Boolean solving. We compare sizes and solving times when encoding w.r.t. the binary base vs. w.r.t. an optimal base (using the `hashB&B` algorithm with the *num_comp* cost function). Encoding w.r.t. the binary base, we solve 435 PB instances (within the time limit) with an average time of 146 seconds and average CNF size of 1.2 million clauses. Using an optimal base, we solve 445 instances with an average time of 108 seconds, and average CNF size of 0.8 million clauses.

Experiment 2 (Base search time): Here we focus on the search time for an optimal base in six configurations using the *sum_carry* cost function. Configurations `M17`, `dfsHP17`, and `B&B17`, are respectively, the MiniSat$^+$ implementation, our `dfsHP` and our `B&B` algorithms, all three searching for an optimal base from $Base_p^{17}$, i.e., with prime elements up to 17. Configurations `hashB&B1,000,000`, `hashB&B10,000`, and `hashB&B17` are our `hashB&B` algorithm searching for a base from $Base_p^{\ell}$ with bounds of $\ell = 1{,}000{,}000$, $\ell = 10{,}000$, and $\ell = 17$, respectively.

Results are summarized in Fig. 4 which is obtained as follows. We cluster the optimal base problems according to the values $\lceil \log_{1.9745} M \rceil$ where M is the maximal coefficient in a problem. Then, for each cluster we take the average runtime for the problems in the cluster. The value 1.9745 is chosen to minimize the standard deviation from the averages (over all clusters). These are the points on the graphs. Configuration `M17` times out on 28 problems. For `dfsHP17`, the maximal search time is 200 seconds. Configuration `B&B17` times out for 1 problem. The `hashB&B` configurations have maximal runtimes of 350 seconds, 14 seconds and 0.16 seconds, respectively for the bounds 1,000,000, 10,000 and 17.

Fig. 4 shows that: (left) even with primes up to 1,000,000, `hashB&B` is faster than the algorithm from MiniSat$^+$ with the limit of 17; and (right) even with primes up to 10,000, the search time using `hashB&B` is essentially negligible.

[4] `http://aprove.informatik.rwth-aachen.de/forms/unified_form_PBB.asp`

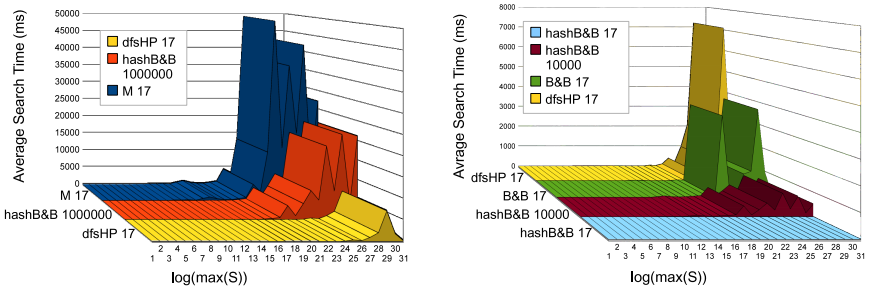


Fig. 4. Experiment 2: the 3 slowest configurations (left) (from back to front) `M17`(blue), `hashB&B1,000,000`(orange) and `dfsHP17`(yellow). The 4 fastest configurations (right) (from back to front) `dfsHP17`(yellow), `B&B17`(green), `hashB&B10,000`(brown) and `hashB&B17`(azure). Note the change of scale for the y-axis with 50k ms on the left and 8k ms on the right. Configuration `dfsHP17` (yellow) is lowest on left and highest on right, setting the reference point to compare the two graphs.

Experiment 3 (Impact on PB solving): Fig. 5 illustrates the influence of improved base search on SAT solving for PB Constraints. Both graphs depict the number of instances solved (the x-axis) within a time limit (the y-axis). On the left, total solving time (with base search), and on the right, SAT solving time only.

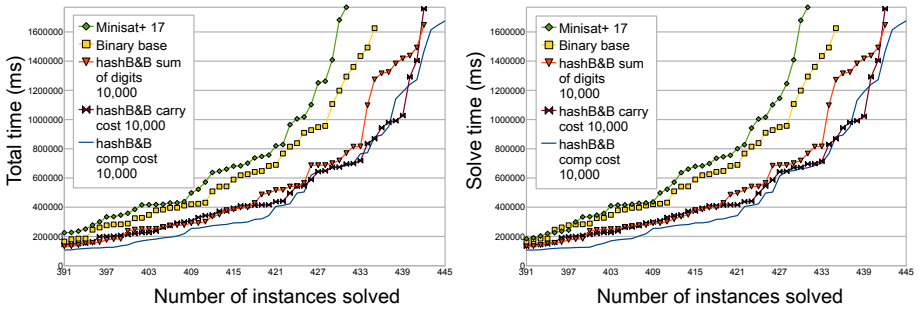


Fig. 5. Experiment 3: total times (left), solving times (right)

Both graphs consider the 734 instances of interest and compare SAT solving times with bases found using five configurations. The first is MiniSat$^+$with configuration `M17`, the second is with respect to the binary base, the third to fifth are `hashB&B` searching for bases from $Base_p^{10,000}(S)$ with cost functions: *sum_digits*, *sum_carry*, and *num_comp*, respectively. The average total/solve run-times (in sec) are 150/140, 146/146, 122/121, 116/115 and 108/107 (left to right). The total number of instances solved are 431, 435, 442, 442 and 445 (left to right). The average CNF sizes (in millions of clauses) for the entire test set/the set where all algorithms solved/the set where no algorithm solved are 7.7/1.0/18, 9.5/1.2/23, 8.4/1.1/20, 7.2/0.8/17 and 7.2/0.8/17 (left to right).

The graphs of Fig. 5 and average solving times clearly show: **(1)** SAT solving time dominates base finding time, **(2)** MiniSat$^+$ is outperformed by the trivial binary base, **(3)** total solving times with our algorithms are faster than with the binary base, and **(4)** the most specific cost function (comparator cost) outperforms the other cost functions both in solving time and size. Finally, note that sum of digits with its nice mathematical properties, simplicity, and application independence solves as many instances as cost carry.

Experiment 4 (Impact of high prime factors): This experiment is about the effects of restricting the maximal prime value in a base (i.e. the value $\ell = 17$ of MiniSat$^+$). An analysis of the our benchmark suite indicates that coefficients with small prime factors are overrepresented. To introduce instances where coefficients have larger prime factors we select 43 instances from the suite and multiply their coefficients to introduce the prime factor 31 raised to the power $i \in \{0, \ldots, 5\}$. We also introduce a slack variable to avoid gcd-based simplification. This gives us a collection of 258 new instances. We used the `B&B` algorithm with the *sum_carry* cost function applying the limit $\ell = 17$ (as in MiniSat$^+$) and $\ell = 31$. Results indicate that for $\ell = 31$, both CNF size and SAT-solving time are independent of the factor 31^i introduced for $i > 0$. However, for $\ell = 17$, both measures increase as the power i increases. Results on CNF sizes are reported in Fig. 6 which plots for 4 different settings the number of instances encoded (x-axis) within a CNF with that many clauses (y-axis).

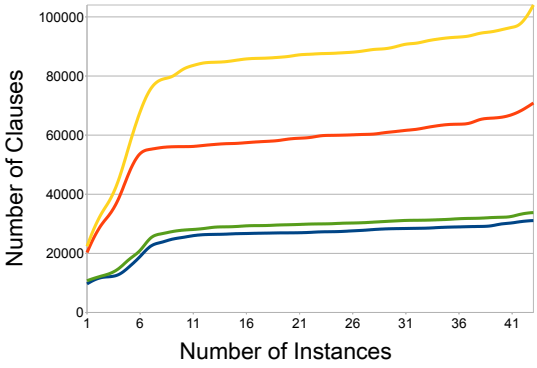


Fig. 6. Experiment 4: Number (x-axis) of instances encoded within number of clauses (y-axis) on 4 configurations. From top line to bottom: (yellow) $\ell = 17$, $i = 5$, (red) $\ell = 17$, $i = 2$, (green) $\ell = 31$, $i = 5$, and (blue) $\ell \in \{17, 31\}$, $i = 0$.

9 Related Work

Recent work [2] encodes Pseudo-Boolean constraints via "totalizers" similar to sorting networks, determined by the representation of the coefficients in an underlying base. Here the authors choose the standard base 2 representation of numbers. It is straightforward to generalize their approach for an arbitrary mixed base, and our algorithm is directly applicable. In [12] the author considers the *sum_digits* cost function and analyzes the size of representing the natural numbers up to n with (a particular class of) mixed radix bases. Our Lemma 6 may lead to a contribution in that context.

10 Conclusion

It has been recognized now for some years that decomposing the coefficients in a Pseudo-Boolean constraint with respect to a mixed radix base can lead to smaller SAT encodings. However, it remained an open problem to determine if it is feasible to find such an optimal base for constraints with large coefficients. In lack of a better solution, the implementation in the MiniSat$^+$ tool applies a brute force search considering prime base elements less than 17.

To close this open problem, we first formalize the optimal base problem and then significantly improve the search algorithm currently applied in MiniSat$^+$. Our algorithm scales and easily finds optimal bases with elements up to 1,000,000. We also illustrate that, for the measure of optimality applied in MiniSat$^+$, one must consider also non-prime base elements. However, choosing the more simple *sum_digits* measure, it is sufficient to restrict the search to prime bases.

With the implementation of our search algorithm it is possible, for the first time, to study the influence of basing SAT encodings on optimal bases. We show that for a wide range of benchmarks, MiniSat$^+$ does actually find an optimal base consisting of elements less than 17. We also show that many Pseudo-Boolean instances have optimal bases with larger elements and that this does influence the subsequent CNF sizes and SAT solving times, especially when coefficients contain larger prime factors.

Acknowledgement We thank Daniel Berend and Carmel Domshlak for useful discussions.

References

1. O. Bailleux, Y. Boufkhad, and O. Roussel. A translation of pseudo boolean constraints to SAT. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 2(1-4):191–200, 2006.
2. O. Bailleux, Y. Boufkhad, and O. Roussel. New encodings of pseudo-boolean constraints into CNF. In *Proc. Theory and Applications of Satisfiability Testing (SAT '09)*, pages 181–194, 2009.
3. P. Barth. *Logic-based 0-1 constraint programming.* Kluwer Academic Publishers, Norwell, MA, USA, 1996.
4. K. E. Batcher. Sorting networks and their applications. In *AFIPS Spring Joint Computing Conference*, volume 32 of *AFIPS Conference Proceedings*, pages 307–314. Thomson Book Company, Washington D.C., 1968.
5. R. E. Bixby, E. A. Boyd, and R. R. Indovina. MIPLIB: A test set of mixed integer programming problems. *SIAM News*, 25:16, 1992.
6. R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Deciding CLU logic formulas via boolean and pseudo-boolean encodings. In *Proc. Intl. Workshop on Constraints in Formal Verification (CFV '02)*, 2002.
7. B. Chor, P. Lemke, and Z. Mador. On the number of ordered factorizations of natural numbers. *Discrete Mathematics*, 214(1-3):123–133, 2000.
8. M. Codish, Y. Fekete, C. Fuhs, and P. Schneider-Kamp. Optimal Base Encodings for Pseudo-Boolean Constraints. Technical Report, `arXiv:1007.4935 [cs.DM]`, available from: `http://arxiv.org/abs/1007.4935`.
9. N. Eén and N. Sörensson. Translating pseudo-boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 2(1-4):1–26, 2006.
10. D. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching.* Addison-Wesley, 1973.
11. V. M. Manquinho and O. Roussel. The first evaluation of Pseudo-Boolean solvers (PB'05). *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 2(1-4):103–143, 2006.
12. N. Sidorov. Sum-of-digits function for certain nonstationary bases. *Journal of Mathematical Sciences*, 96(5):3609–3615, 1999.