Rewriting Optimization Statements in Answer-Set Programs*

Jori Bomanson[1], Martin Gebser[2], and Tomi Janhunen[3]

1 **HIIT and Department of Computer Science, Aalto University, Espoo, Finland**
`jori.bomanson@aalto.fi`
2 **Department of Computer Science, University of Potsdam, Potsdam, Germany**
`gebser@cs.uni-potsdam.de`
3 **HIIT and Department of Computer Science, Aalto University, Espoo, Finland**
`tomi.janhunen@aalto.fi`

Abstract

Constraints on *Pseudo-Boolean* (PB) expressions can be translated into Conjunctive Normal Form (CNF) using several known translations. In *Answer-Set Programming* (ASP), analogous expressions appear in weight rules and *optimization statements*. Previously, we have translated weight rules into normal rules, using normalizations designed in accord with existing CNF encodings. In this work, we rededicate such designs to *rewrite* optimization statements in ASP. In this context, a rewrite of an optimization statement is a replacement accompanied by a set of normal rules that together replicate the original meaning. The goal is partially the same as in translating PB constraints or weight rules: to introduce new meaningful auxiliary atoms that may help a solver in the search for (optimal) solutions. In addition to adapting previous translations, we present selective rewriting techniques in order to meet the above goal while using only a limited amount of new rules and atoms. We experimentally evaluate these methods in preprocessing ASP optimization statements and then searching for optimal answer sets. The results exhibit significant advances in terms of numbers of optimally solved instances, reductions in search conflicts, and shortened computation times. By appropriate choices of rewriting techniques, improvements are made on instances involving both small and large weights. In particular, we show that selective rewriting is paramount on benchmarks involving large weights.

1998 ACM Subject Classification I.2.3 Deduction and Theorem Proving

Keywords and phrases Answer-Set Programming, Pseudo-Boolean optimization, Translation methods

Digital Object Identifier 10.4230/OASIcs.ICLP.2016.5

1 Introduction

Answer-Set Programming (ASP) is a declarative programming paradigm suited to solving computationally challenging search problems [13] by encoding them as *answer-set programs*, commonly consisting of *normal*, *cardinality*, and *weight rules*, as well as *optimization statements* [31]. The latter three relate to linear *Pseudo-Boolean* (PB) constraints [21, 30] and PB optimization statements. Rules restrict the acceptable combinations of truth values for the atoms they contain. The role of a weight rule, or a PB constraint, is to check a bound on a weighted sum of literals, whereas an optimization statement aims at minimizing such a

* This work was funded by the Academy of Finland (251170), DFG (SCHA 550/9), as well as DAAD and the Academy of Finland (57071677/279121).

Technical Communications of the 32nd International Conference on Logic Programming (ICLP 2016).
Editors: Manuel Carro, Andy King, Neda Saeedloei, and Marina De Vos; Article No. 5; pp. 5:1–5:15

Open Access Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

sum. Solutions to search problems can be cast to models, called *answer sets*, and computed by employing ASP *grounders* and *solvers* [4, 17, 22, 27, 31].

In both ASP and PB solving, support for non-standard rules or constraints can be implemented by translating them into simpler logical primitives prior to solving. The performance implications of such *translation-based approaches* in comparison to *native* solving techniques are mixed: both improvements and deteriorations have been observed [2, 10]. A promising direction in recent research [3, 2] is to translate only important constraints or parts of constraints during search. In previous work, we have evaluated the feasibility of translating cardinality rules [11] and weight rules [10] into normal rules, in a process that we call *normalization*. The explored techniques build on methods successfully applied in PB solving [21, 6, 5, 1]. The main observation with relevance to this work is that normalization commonly reduces the number of search conflicts at the cost of increased instance sizes and more time spent between conflicts.

In this paper, we introduce ways in which the primitives developed for cardinality and weight rule normalization can be used to rewrite optimization statements. Indeed, typical normalization or translation methods encode sums of input weights in some number system, such as unary or mixed-radix numbers, for which the comparison to a bound is straightforward. When dealing with optimization statements, the encodings of sums can be applied to the statements while forgoing comparisons. For illustration, suppose we intend to minimize the sum $a+b$ of two atoms. Then, we may equally well minimize the sum $c+d$ of two new atoms defined by sorting a and b via the rules "c :- a. c :- b. d :- a, b." This modification preserves the answer sets of a program regarding the original atoms and the respective optimization values. As in weight rule normalization, the purpose is to enhance solving performance by supplementing problem instances with structure in the form of auxiliary atoms. The atoms are defined in intuitively meaningful ways and provide new opportunities for ASP solvers to learn *no-goods* [22]. In addition, we develop selective rewriting techniques that partition input optimization statements and then rewrite some or all of the parts in separation. As a consequence, the size increase due to rewriting is mitigated, concerning the introduced auxiliary atoms and normal rules, and we study the tradeoff between the costs and benefits of rewriting in terms of solving performance.

The paper is organized as follows. Section 2 introduces basic notations, answer-set programs, and simple optimization rewriting techniques. More elaborate techniques to rewrite optimization statements are presented in Section 3. In Section 4, we experimentally study the performance implications of applying these techniques. Related work is discussed in Section 5, and Section 6 concludes the paper.

2 Preliminaries and Basic Techniques

The following subsections introduce matrix-based expressions used to represent optimization statements and simple techniques for rewriting them.

2.1 Pseudo-Boolean Expressions

A *positive literal* is a propositional *atom* a, and not a is a *negative literal*. We write $(n \times 1)$-matrices, or (column) vectors, as $\mathbf{v} = [v_1; \ldots; v_n]$ and refer to them by symbols like $\mathbf{b}, \mathbf{d}, \mathbf{w}, \boldsymbol{\pi}$ when the elements v_i are nonnegative integers, by $\mathbf{1}$ when $v_1 = \cdots = v_n = 1$, and by $\boldsymbol{h}, \boldsymbol{l}, \boldsymbol{p}, \boldsymbol{q}, \boldsymbol{r}, \boldsymbol{s}, \boldsymbol{t}$ when $v_1, \ldots, v_n$ are literals. The *concatenation* of vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ is $[\mathbf{v}_1; \mathbf{v}_2]$. We also use $(m \times n)$-matrices W of nonnegative integers w_{ij}, where i is the row and j the column. For the result $\mathrm{W} = \mathrm{AB}$ of multiplying $(m \times k)$- and $(k \times n)$-matrices

with elements a_{ij} and b_{ij}, respectively, we have $w_{ij} = \sum_{l=1}^{k} a_{il}b_{lj}$. In the *transpose* W^{T}, each w_{ij} is swapped with w_{ji}. We define a *Pseudo-Boolean* (PB) *expression* e to be a linear combination of nonnegative *weights* $\mathbf{w} = [w_1; \ldots; w_n]$ and literals $\boldsymbol{l} = [l_1; \ldots; l_n]$:

$$e = \mathbf{w}^{\mathrm{T}}\boldsymbol{l} = [w_1 \cdots w_n] \begin{bmatrix} l_1 \\ \vdots \\ l_n \end{bmatrix} = w_1 l_1 + \cdots + w_n l_n.$$

Let $\mathrm{A}(e) = \mathrm{A}(\boldsymbol{l}) = \{a \mid 1 \leq i \leq n, l_i = a \text{ or } l_i = \text{not } a\}$ denote the set of atoms in e and $\boldsymbol{l}$. An *interpretation* I is a set of atoms distinguishing true atoms $a \in I$ and false atoms $a \notin I$. A vector like $\boldsymbol{l}$ *evaluates* to $\boldsymbol{l}(I) = [b_1; \ldots; b_n]$ at I, where for $1 \leq i \leq n$, we have $b_i = 1$ iff $l_i = a$ and $a \in I$, or $l_i = \text{not } a$ and $a \notin I$, and $b_i = 0$ otherwise. An expression like e *evaluates* to $e(I) = (\mathbf{w}^{\mathrm{T}}\boldsymbol{l})(I) = \mathbf{w}^{\mathrm{T}}(\boldsymbol{l}(I))$ at I. We extend this notation to sums of expressions by letting $(e_1 + \cdots + e_m)(I) = e_1(I) + \cdots + e_m(I)$. Two expressions e_1 and e_2 are *equivalent*, denoted by $e_1 \equiv e_2$, iff $e_1(I) = e_2(I)$ for each $I \subseteq \mathrm{A}(e_1) \cup \mathrm{A}(e_2)$.

2.2 Answer-Set Programs

We consider (ground) answer-set *programs* P, defined as sets of (normal) *rules*, which are triples $r = (a, B, C)$ of a *head* atom a and sets of *positive body* atoms B and *negative body* atoms C. An *optimization program* O is a program-expression pair (P, e), written in the *ASP-Core-2* input language format [14] using the forms

$$a \text{ :- } b_1, \ldots, b_k, \text{ not } c_1, \ldots, \text{ not } c_m. \tag{1}$$

$$\text{:}\sim l_1.\ [w_1, 1] \quad \ldots \quad \text{:}\sim l_n.\ [w_n, n] \tag{2}$$

$$\#\text{minimize } \{w_1, 1 : l_1; \ldots; w_n, n : l_n\}. \tag{3}$$

for rules $(a, \{b_1, \ldots, b_k\}, \{c_1, \ldots, c_m\})$ in (1), and *weak constraints* in (2) or *optimization statements* in (3) for the expression $e = w_1 l_1 + \cdots + w_n l_n$.

Let $\mathrm{A}(P) = \bigcup_{(a,B,C)\in P}(\{a\} \cup B \cup C)$, $\mathrm{H}(O) = \mathrm{H}(P) = \{a \mid (a, B, C) \in P\}$, and $\mathrm{A}(O) = \mathrm{A}(P) \cup \mathrm{A}(e)$ denote the sets of atoms occurring in P, as heads in P, or in $O = (P, e)$, respectively. An interpretation I *satisfies* a rule $r = (a, B, C)$ iff $B \subseteq I$ and $C \cap I = \emptyset$ imply $a \in I$. The *reduct* of P with respect to I is $P^I = \{(a, B, \emptyset) \mid (a, B, C) \in P, C \cap I = \emptyset\}$. The set $\mathrm{SM}(P)$ of *stable models* of P, also called *answer sets* of P, is the set of all interpretations $M \subseteq \mathrm{A}(P)$ that are subset-minimal among the interpretations satisfying every rule $r \in P^M$. A stable model M of P is *optimal* iff $e(M) = \min\{e(N) \mid N \in \mathrm{SM}(P)\}$.

Our goal is to rewrite optimization statements while preserving the stable models of a program and associated optimization values. To this end, we utilize notions for comparing the joint parts of optimization programs [26]. Two sets S_1 and S_2 of interpretations are *visibly equal* with respect to a set V of *visible* atoms iff there is a bijection $f : S_1 \to S_2$ such that, for each $I \in S_1$, we have $I \cap V = f(I) \cap V$.

For two sets V_1 and V_2 of atoms, a program P *realizes* a function $f : 2^{V_1} \to 2^{V_2}$ iff for each $I \subseteq V_1$, there is exactly one $M \in \mathrm{SM}(P \cup \{a. \mid a \in I\})$ and $f(I) = M \cap V_2$. An optimization program $O = (P, e')$ is an *optimization rewrite* of an expression e with respect to a set V of visible atoms iff P realizes a function $f : 2^V \to 2^{\mathrm{A}(e')}$ such that, for each $I \subseteq V$, we have $e(I) = e'(f(I))$. In this case, we also say that e is *rewritable* as (P, e') with respect to V.

To decompose optimization rewrites, we say that a set V of atoms and a sequence $P_1, \ldots, P_m$ of programs *fit* iff $(V \cup \bigcup_{j=1}^{i-1} \mathrm{A}(P_j)) \cap \mathrm{H}(P_i) = \emptyset$ for each $1 \leq i \leq m$. Programs that fit preserve the definitions of atoms in V and the programs preceding them.

▶ **Proposition 1.** *Let $O = (P, e)$ be an optimization program, and e be rewritable as (P', e') with respect to $\mathrm{A}(O)$ such that $\mathrm{A}(O)$ and P' fit. Then, there is a bijection $f : \mathrm{SM}(P) \to \mathrm{SM}(P \cup P')$ such that*

1. *$\mathrm{SM}(P)$ and $\mathrm{SM}(P \cup P')$ are visibly equal with respect to $\mathrm{A}(O)$ via f, and*
2. *$e(M) = e'(f(M))$ for each $M \in \mathrm{SM}(P)$.*

2.3 Optimization Rewrites for Small Weights

In this subsection, we examine simple, yet effective rewriting techniques applicable to optimization statements with small weights. To begin with, we define building blocks for sorting operations on vectors of literals. Intuitively, a vector $\boldsymbol{s}$ of literals encodes the value $(\mathbf{1}^{\mathrm{T}}\boldsymbol{s})(I)$ at an interpretation I. When $\boldsymbol{s}$ is sorted, it represents this value as a *unary* number. To obtain such numbers, vectors of literals can be recursively sorted and added up via merging. These operations permute truth values, and hence preserve the encoded values.

▶ **Definition 2.** A vector $\boldsymbol{s}$ of literals is *sorted at* an interpretation I iff the weights in $\boldsymbol{s}(I)$ are monotonically decreasing, and *sorted under* a set S of interpretations iff $\boldsymbol{s}$ is sorted at each $I \in S$.

▶ **Definition 3.** Let $\boldsymbol{t} = [\boldsymbol{h}_1; \boldsymbol{h}_2]$ and $\boldsymbol{s}$ be vectors of literals having the same length, and P be a program realizing a function $f : 2^{\mathrm{A}(\boldsymbol{t})} \to 2^{\mathrm{A}(\boldsymbol{s})}$. Then, P is

1. a *sorting program* with input $\boldsymbol{t}$ and output $\boldsymbol{s}$ iff for each $I \subseteq \mathrm{A}(\boldsymbol{t})$, $\boldsymbol{s}$ is sorted at $f(I)$ and $(\mathbf{1}^{\mathrm{T}}\boldsymbol{t})(I) = (\mathbf{1}^{\mathrm{T}}\boldsymbol{s})(f(I))$;
2. a *merging program* with inputs $\boldsymbol{h}_1, \boldsymbol{h}_2$ and output $\boldsymbol{s}$ iff for each $I \subseteq \mathrm{A}(\boldsymbol{t})$ at which $\boldsymbol{h}_1$ and $\boldsymbol{h}_2$ are sorted, $\boldsymbol{s}$ is sorted at $f(I)$ and $(\mathbf{1}^{\mathrm{T}}[\boldsymbol{h}_1; \boldsymbol{h}_2])(I) = (\mathbf{1}^{\mathrm{T}}\boldsymbol{s})(f(I))$.

Moreover, for any program $P' \supseteq P$, we assume that $(\mathrm{A}(P) \cup \mathrm{A}(\boldsymbol{s})) \cap \mathrm{H}(P' \setminus P) \subseteq \mathrm{A}(\boldsymbol{t})$.

▶ **Example 4.** A sorting program P with input $\boldsymbol{t} = [t_1; t_2]$ and output $\boldsymbol{s} = [s_1; s_2]$ such that $\mathrm{A}(\boldsymbol{t}) \cap \mathrm{A}(\boldsymbol{s}) = \{t_1, t_2\} \cap \{s_1, s_2\} = \emptyset$, which yields an optimization rewrite $(P, \mathbf{1}^{\mathrm{T}}\boldsymbol{s})$ of $\mathbf{1}^{\mathrm{T}}\boldsymbol{t}$ with respect to $\mathrm{A}(\boldsymbol{t}) = \{t_1, t_2\}$, is given by

$$s_1 \coloneq t_1. \quad s_1 \coloneq t_2. \quad s_2 \coloneq t_1,\ t_2.$$

This program can serve as a base case in recursive constructions of larger sorting programs, such as the following. Given vectors $\boldsymbol{h}_1$, $\boldsymbol{h}_2$, $\boldsymbol{s}_1$, $\boldsymbol{s}_2$ and $\boldsymbol{s}$ of literals having appropriate lengths, a sorting program with input $\boldsymbol{t} = [\boldsymbol{h}_1; \boldsymbol{h}_2]$ and output $\boldsymbol{s}$ is recursively obtained as the union of (i) a sorting program P_1 with input $\boldsymbol{h}_1$ and output $\boldsymbol{s}_1$, (ii) a sorting program P_2 with input $\boldsymbol{h}_2$ and output $\boldsymbol{s}_2$, and (iii) a merging program P_3 with inputs $\boldsymbol{s}_1, \boldsymbol{s}_2$ and output $\boldsymbol{s}$, assuming that $\mathrm{A}(\boldsymbol{t})$ and P_1, P_2, P_3 fit.

Note that, by definition, sorting programs are merging programs, and both lend themselves to rewriting expressions with unit weights. Moreover, such *optimization rewriting* can be applied to an arbitrary expression $\mathbf{w}^{\mathrm{T}}\boldsymbol{l}$ after *flattening* it into the form $\mathbf{1}^{\mathrm{T}}\boldsymbol{t}$, where $\boldsymbol{t}$ is any vector such that $\mathbf{1}^{\mathrm{T}}\boldsymbol{t} \equiv \mathbf{w}^{\mathrm{T}}\boldsymbol{l}$. For example, $e = 2a + 4b + 3c + 3d + e + 4f$ is reproduced by picking $\boldsymbol{t} = [a; a; b; b; b; b; c; c; c; d; d; d; e; f; f; f; f]$. Rewrites based on flattening can, however, become impractically large when there are literals with large non-unit weights. To alleviate this problem, we consider selective rewriting techniques in Section 3.3, and alternative ways to handle non-unit weights in Section 3.4.

3 More Elaborate Rewriting Techniques

In this section, we present techniques for rewriting optimization statements that build on those presented in Section 2.3. They are based on a process in which we rewrite optimization statements in parts using simple techniques and then form new substitute optimization statements from the outputs. Before going into the details, let us illustrate the basic idea.

▶ **Example 5.** Consider the minimization statement

$$\#\text{minimize}\ \{5,1:a;\ 10,2:b;\ 15,3:c\}.$$

The statement encodes the expression $5a+10b+15c$. To deal with the non-unit weights, we can flatten it into $5a+5b+5b+5c+5c+5c$ and rewrite it using a single sorting program with input $[a;\ b;\ b;\ c;\ c;\ c]$. On the other hand, we obtain a more concise rewrite by modifying the expression into $5a+5c+10b+10c$, sorting the parts $[a;\ c]$ and $[b;\ c]$ into vectors of auxiliary atoms $[d;\ e]$ and $[f;\ g]$, and minimizing the expression $5d+5e+10f+10g$:

$$d \text{ :- } a. \quad d \text{ :- } c. \quad e \text{ :- } a, c.$$
$$f \text{ :- } b. \quad f \text{ :- } c. \quad g \text{ :- } b, c.$$
$$\#\text{minimize}\ \{5,1:d;\ 5,2:e;\ 10,3:f;\ 10,4:g\}.$$

3.1 Mixed-radix Bases and Decomposition

We define a *mixed-radix base* to be any pair $(\mathbf{b},\boldsymbol{\pi})$ of *radices* $\mathbf{b}=[b_1;\ \ldots;\ b_k]$ and *place values* $\boldsymbol{\pi}=[\pi_1;\ \ldots;\ \pi_k]$ such that $b_k=\infty$ and, for each $1\leq i\leq k$, we have $\pi_i=\prod_{j=1}^{i-1}b_i$. Examples of mixed-radix bases include the usual base for counting seconds, minutes, hours, and days, $([60;\ 60;\ 24;\ \infty],[1;\ 60;\ 3600;\ 86400])$, and the finite-length binary base $([2;\ \ldots;\ 2;\ \infty],[1;\ 2;\ 4;\ \ldots;\ 2^{k-1}])$ for any $k\geq 1$. In a given base $(\mathbf{b},\boldsymbol{\pi})$, every nonnegative integer d has a *mixed-radix decomposition* with *digits* $\mathbf{d}=[d_1;\ \ldots;\ d_k]$ such that $\mathbf{d}^{\mathrm{T}}\boldsymbol{\pi}=d$, and exactly one decomposition $\mathbf{d}$ of d satisfies $d_i<b_i$ for all $1\leq i<k$. We say that d_i is the ith *least* or the $(k+1-i)$th *most significant* digit of d in base $(\mathbf{b},\boldsymbol{\pi})$. More generally, the mixed-radix decomposition of an $(n\times 1)$-vector $\mathbf{w}$ of weights is a $(k\times n)$-matrix W such that $\mathrm{W}^{\mathrm{T}}\boldsymbol{\pi}=\mathbf{w}$. By these definitions, the ith row gives the ith least significant digits of all weights, and the jth column gives the digits of the jth weight.

▶ **Example 6.** In base $(\mathbf{b},\boldsymbol{\pi})=([3;\ 2;\ 2;\ \infty],\ [1;\ 3;\ 6;\ 12])$, we may decompose weights $\mathbf{w}=\begin{bmatrix}21\\1\\3\\5\end{bmatrix}$ into a matrix $\mathrm{W}=\begin{bmatrix}0&1&0&2\\1&0&1&1\\1&0&0&0\\1&0&0&0\end{bmatrix}$ such that $\mathrm{W}^{\mathrm{T}}\boldsymbol{\pi}=\begin{bmatrix}0&1&1&1\\1&0&0&0\\0&1&0&0\\2&1&0&0\end{bmatrix}\begin{bmatrix}1\\3\\6\\12\end{bmatrix}=\begin{bmatrix}21\\1\\3\\5\end{bmatrix}=\mathbf{w}$.

3.2 Selecting Mixed-radix Bases

Given a vector $\mathbf{w}=[w_1;\ \ldots;\ w_n]$ of weights, our goal is to pick a base $(\mathbf{b},\boldsymbol{\pi})$ with some number k of radices that yields a decomposition matrix W with small digits w_{ij} such that $\mathrm{W}^{\mathrm{T}}\boldsymbol{\pi}=\mathbf{w}$. In view of sorting programs, introduced in Section 2.3, whose sizes are of order $\mathrm{c}_{\mathrm{sort}}(n)=n(\log n)^2$, the aim is to minimize the size needed for sorting every row:

$$\mathrm{c}(\mathbf{b},\mathbf{w})=\sum_{i=1}^{k}\mathrm{c}_{\mathrm{sort}}\left(\sum_{j=1}^{n}w_{ij}\right).$$

To this end, we use a greedy heuristic algorithm: for each $i=1,2,\ldots,$ let $\pi_i=\prod_{j=1}^{i-1}b_j$ and pick radix b_i as the product of the least prime p that minimizes $\mathrm{c}([p;\ 2;\ \ldots;\ 2],$

$[\lfloor w_1/\pi_i \rfloor; \ldots; \lfloor w_n/\pi_i \rfloor])$ and the greatest common divisor of $\{\lfloor w_j/(\pi_i p) \rfloor \mid 1 \leq j \leq n\}$, defined here to be infinite for $\{0\}$, and stop at $b_i = \infty$. We note that complete optimization procedures were proposed for finding optimal bases in translating PB constraints [18].

3.3 Selective Optimization Rewriting

In the following, we define ways to carry out partial optimization rewriting. The goal is to reduce the needed size while retaining as much of the benefits of rewriting as possible.

By additivity, sums of expressions can be rewritten term-by-term. Recall that, given an expression $e = \mathbf{w}^{\mathrm{T}}\boldsymbol{l}$, we may decompose $\mathbf{w}$ in any mixed-radix base to obtain a matrix W. Then, $\mathbf{w}^{\mathrm{T}} = (\mathrm{W}^{\mathrm{T}}\boldsymbol{\pi})^{\mathrm{T}} = \boldsymbol{\pi}^{\mathrm{T}}\mathrm{W}$ yields $e = \boldsymbol{\pi}^{\mathrm{T}}\mathrm{W}\boldsymbol{l}$, and any sum such that $\mathrm{W}_1 + \cdots + \mathrm{W}_m = \mathrm{W}$ carries over to a sum reproducing the expression $e = \boldsymbol{\pi}^{\mathrm{T}}(\sum_{i=1}^{m} \mathrm{W}_i)\boldsymbol{l} = \sum_{i=1}^{m} \boldsymbol{\pi}^{\mathrm{T}}\mathrm{W}_i\boldsymbol{l}$.

▶ **Lemma 7.** *Let* $\mathrm{W} = \mathrm{W}_1 + \cdots + \mathrm{W}_m$ *be a mixed-radix decomposition such that* $\mathrm{W}^{\mathrm{T}}\boldsymbol{\pi} = \mathbf{w}$ *for an expression* $e = \mathbf{w}^{\mathrm{T}}\boldsymbol{l}$*. For* $1 \leq i \leq m$*, let* $\boldsymbol{\pi}^{\mathrm{T}}\mathrm{W}_i\boldsymbol{l}$ *be rewritable as* $O_i = (P_i, e_i)$ *with respect to* $\mathrm{A}(e)$ *such that* $\mathrm{A}(e)$ *and* $O_1, \ldots, O_{i-1}, O_{i+1}, \ldots, O_m, O_i$ *fit. Then,* e *is rewritable as* $(\bigcup_{i=1}^{m} P_i, \sum_{i=1}^{m} e_i)$ *with respect to* $\mathrm{A}(e)$*.*

This opens up selective rewriting strategies. To this end, suppose there are k radices in a base $(\mathbf{b}, \boldsymbol{\pi})$, and let $\mathbf{w} = [w_1; \ldots; w_n]$, so that W is a $(k \times n)$-matrix. For any $m \in \{k, n\}$ and $S \subseteq \{1, \ldots, m\}$, let I_S denote the symmetric $(m \times m)$-*selection* matrix having the value 1 in row i and column i iff $i \in S$, and 0 otherwise. Intuitively, when fixing some $S \subseteq \{1, \ldots, n\}$, the $(k \times n)$-matrix WI_S selects weights w_i for all $i \in S$, while columns $j \notin S$ are set to zero. Similarly, when we fix some $S \subseteq \{1, \ldots, k\}$, the $(k \times n)$-matrix $\mathrm{I}_S\mathrm{W}$ captures the ith significant digits for all $i \in S$, and rows $j \notin S$ are set to zero.

▶ **Example 8.** Given $\mathrm{W} = \left[\begin{smallmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{smallmatrix}\right]$ and $S = \{1, 2\}$, we have $\mathrm{I}_S = \left[\begin{smallmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{smallmatrix}\right]$, $\mathrm{WI}_S = \left[\begin{smallmatrix} 0 & 1 & 0 \\ 3 & 4 & 0 \\ 6 & 7 & 0 \end{smallmatrix}\right]$, and $\mathrm{I}_S\mathrm{W} = \left[\begin{smallmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 0 & 0 & 0 \end{smallmatrix}\right]$.

By means of selection matrices, we can conveniently partition W in either dimension, along its columns or rows, respectively. Namely, given a partition $S_1, \ldots, S_l$ of $\{1, \ldots, m\}$ for $m \in \{k, n\}$, we have that $\sum_{i=1}^{l} \mathrm{I}_{S_i}$ is the identity matrix. In view of Lemma 7, $\sum_{i=1}^{l} \mathrm{WI}_{S_i} = \mathrm{W}$, if $m = n$, and $\sum_{i=1}^{l} \mathrm{I}_{S_i}\mathrm{W} = \mathrm{W}$, if $m = k$, thus yield literal-wise or significance-wise optimization rewrites, respectively. For example, lines (a) in Figure 1 draw the partition $\{1, 2, 3\}, \{4, 5, 6\}, \{7, 8, 9\}$ applied to $m = n = 9$ literals, limiting the size of parts to $t = 3$. We below focus on particular cases of such *matrix partition rewrites*.

To begin with, we may rewrite some *bounded* number t of weighted literals in *equal-weight chunks* by forming a partition $S_1, \ldots, S_l$ based on (maximal) sets S_i such that $|S_i| \leq t$ and $|\{w_i \mid i \in S_i\}| = 1$, so that each chunk consists of up to t literals of the same weight. Second, we may let $l = k$ and pick $S_i = \{i\}$ for each $1 \leq i \leq k$ to rewrite an expression in *digit-wise* layers of the form $\boldsymbol{\pi}^{\mathrm{T}}(\mathrm{I}_{S_i}\mathrm{W})\boldsymbol{l}$. Third, we may drop a number t of least significant digits from each weight, such as those above line (b) in Figure 1, and rewrite the *globally* most significant digits only. This amounts to using quotients due to division by π_{t+1}, and can be formalized by applying the previous technique to a base $([\pi_{t+1}; \infty], [1; \pi_{t+1}])$, where only the quotient weights $\lfloor w_i/\pi_{t+1} \rfloor$ are rewritten for $1 \leq i \leq n$.

Matrix partition rewrites not a priori referring to particular digits include the *literal-wise* approach indicated by lines (a) in Figure 1. Another strategy is to pick, for each weight, a number t of its *locally* most significant digits starting from the most significant nonzero digit. To express this as a matrix partition rewrite, let W be a $(k \times n)$-matrix as before. Then, define an equally-sized matrix W_2 by mapping the elements $w_1, \ldots, w_k$

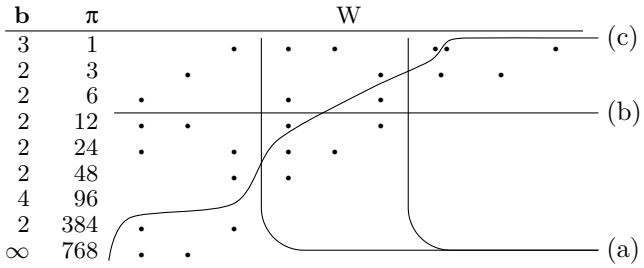

Figure 1 A mixed-radix decomposition matrix expressed using dots for digit 1 and pairs of dots for 2. The lines represent partitions for matrix partition rewrites. Columns separated by lines (a) denote a literal-wise partition, using the parameter value $t = 3$. Digits below lines (b) and (c) represent the globally or locally most significant digits to be rewritten, based on $t = 3$ or $t = 2$, respectively.

of each column in W to elements $v_1, \ldots, v_k$ of a respective column in W_2 such that $v_i = \max\{0, \min\{w_i, t - \sum_{j=i+1}^{l} u_j\}\}$ for each $1 \leq i \leq k$, where $[u_1; \ldots; u_l] = [b_1; \ldots; b_{l-1}; w_l]$, $w_l \neq 0$, and $w_{l+1} = \cdots = w_k = 0$. This yields $\mathrm{W} = \mathrm{W}_1 + \mathrm{W}_2$ for $\mathrm{W}_1 = \mathrm{W} - \mathrm{W}_2$, so that, by Lemma 7, $\boldsymbol{\pi}^{\mathrm{T}}\mathrm{W}_1\boldsymbol{l}$ and $\boldsymbol{\pi}^{\mathrm{T}}\mathrm{W}_2\boldsymbol{l}$ can be rewritten separately. Similar to rewriting the globally most significant digits only, we may rewrite the locally more significant part W_2, as located below line (c) in Figure 1 for $t = 2$, but not the rest.

3.4 Optimization Rewrites for Large Weights

We build on sorting and merging programs to devise optimization rewrites applicable to expressions containing large non-unit weights. To this end, we begin by defining an abstract class of rewrites that encode unary numbers as sorted vectors of literals. This allows us to compose rewrites from building blocks that rely on sorted inputs to produce sorted outputs.

▶ **Definition 9.** Let $O = (P, d + \alpha_1\mathbf{1}^{\mathrm{T}}\boldsymbol{s}_1 + \cdots + \alpha_m\mathbf{1}^{\mathrm{T}}\boldsymbol{s}_m)$ be an optimization rewrite of an expression e such that P realizes $f : 2^{\mathrm{A}(e)} \to 2^{\mathrm{A}(d+\alpha_1\mathbf{1}^{\mathrm{T}}\boldsymbol{s}_1+\cdots+\alpha_m\mathbf{1}^{\mathrm{T}}\boldsymbol{s}_m)}$. Then, O is a *multi-unary rewrite* of e with the set $\{\boldsymbol{s}_1, \ldots, \boldsymbol{s}_m\}$ of vectors as its output iff $\boldsymbol{s}_i$ is sorted under the image of f for all $1 \leq i \leq m$.

Recall the strategy from Section 3.3 to rewrite an expression in digit-wise layers based on a mixed-radix decomposition. Such a *digit-wise* rewrite can be realized by flattening along with sorting programs applied to the layers in parallel. In this process, the radix b_i for a layer i limits the number of (flattened) inputs to a corresponding sorting program.

▶ **Proposition 10.** *Let* W *be a mixed-radix decomposition in a base* $(\mathbf{b}, \boldsymbol{\pi})$ *with place values* $\boldsymbol{\pi} = [\pi_1; \ldots; \pi_k]$ *such that* $\mathrm{W}^{\mathrm{T}}\boldsymbol{\pi} = \mathbf{w}$ *for an expression* $e = \mathbf{w}^{\mathrm{T}}\boldsymbol{l}$*. For* $1 \leq i \leq k$*, let* $\mathbf{w}_i$ *be the* i*th row of* W*,* $\boldsymbol{t}_i$ *be some vector such that* $\mathbf{1}^{\mathrm{T}}\boldsymbol{t}_i \equiv \mathbf{w}_i^{\mathrm{T}}\boldsymbol{l}$*, and* P_i *be a sorting program with input* $\boldsymbol{t}_i$ *and output* $\boldsymbol{s}_i$ *such that* $\mathrm{A}(e)$ *and*

$$P_1, \ldots, P_{i-1}, P_{i+1}, \ldots, P_k, P_i \cup \{(a, \emptyset, \emptyset) \mid a \in (\mathrm{A}(P_i) \cup \mathrm{A}(\boldsymbol{s}_i)) \cap (\mathrm{A}(e) \setminus \mathrm{A}(\boldsymbol{t}_i))\}$$

fit. Then, $(\bigcup_{i=1}^{k} P_i, \sum_{i=1}^{k} \pi_i\mathbf{1}^{\mathrm{T}}\boldsymbol{s}_i)$ *is a multi-unary rewrite of* e *with output* $\{\boldsymbol{s}_1, \ldots, \boldsymbol{s}_k\}$*.*

The above conditions that $\mathrm{A}(e)$ and sorting programs fit (in any order) as well as that atoms from $\mathrm{A}(e)$ are used as inputs in $\boldsymbol{t}_i$ only make sure that the sorting programs define disjoint atoms and otherwise evaluate nothing but their inputs. While such conditions are

easy to establish, in practice, different sorting programs may share common substructures based on the same inputs. In fact, a scheme for optimizing the layout of sorting programs towards structure sharing is given in [10].

Digit-wise rewrites based on sorting programs yield non-unique mixed-radix decompositions of the sum of input weights. For example, given $\mathbf{b} = [6; \infty]$, $\boldsymbol{\pi} = [1; 6]$, and $e = 5a + 5b + 10c + d$, the sorting programs from Proposition 10 realize a function $f : 2^{\{a,b,c,d\}} \to 2^{\{s_{1,1},\dots,s_{1,15},s_{2,1}\}}$ leading to an output expression e' of the form $s_{1,1} + \dots + s_{1,15} + 6s_{2,1}$. Then, the sum 10, associated with both $I_1 = \{a, b\}$ and $I_2 = \{c\}$, is mapped to $f(I_1) = \{s_{1,1}, \dots, s_{1,10}\}$ or $f(I_2) = \{s_{1,1}, \dots, s_{1,4}, s_{2,1}\}$, respectively, where $e'(f(I_1)) = e'(f(I_2)) = 10$. In terms of ASP solving, this means that a bound on the optimization value is not captured by a single no-good. Instead, several representations of the same value may be produced during search. An encoding of a *unique* mixed-radix decomposition can be built by combining sorting programs with *deferred carry propagation*, utilizing merging programs, as introduced in Section 2.3, to express addition along with division of unary numbers by constants. To begin with, we formalize the role of a merging program in this context.

▶ **Lemma 11.** *Let $O = (P, d + \alpha\mathbf{1}^{\mathrm{T}}\boldsymbol{h}_1 + \alpha\mathbf{1}^{\mathrm{T}}\boldsymbol{h}_2)$ be a multi-unary rewrite of an expression e with output $\{\boldsymbol{h}_1, \boldsymbol{h}_2\}$, and P' be a merging program with inputs $\boldsymbol{h}_1, \boldsymbol{h}_2$ and output $\boldsymbol{s}$ such that $\mathrm{A}(O)$ and $P' \cup \{(a, \emptyset, \emptyset) \mid a \in (\mathrm{A}(P') \cup \mathrm{A}(\boldsymbol{s})) \cap (\mathrm{A}(e) \setminus \mathrm{A}([\boldsymbol{h}_1; \boldsymbol{h}_2]))\}$ fit. Then, $(P \cup P', d + \alpha\mathbf{1}^{\mathrm{T}}\boldsymbol{s})$ is a multi-unary rewrite of e with output $\{\boldsymbol{s}\}$.*

The purpose of merging programs is to map the sum of digits in one layer and a corresponding carry from less significant layers to a unary number, which can in turn provide a carry to the next layer. To obtain such carries, we make use of division. Namely, given a vector $\boldsymbol{s} = [s_1; \dots; s_n]$ and a positive integer m, we define the *quotient* of $\boldsymbol{s}$ divided by m as $[s_m; s_{2m}; \dots; s_{\lfloor n/m \rfloor m}]$, which contains every mth literal of $\boldsymbol{s}$. A respective residue, also represented as a unary number, is produced by a program as follows.

▶ **Definition 12.** Let $\boldsymbol{s} = [s_1; \dots; s_n]$ and $\boldsymbol{r} = [r_1; \dots; r_k]$ be vectors of literals such that $k = \min\{n, m-1\}$ for some positive integer m. A program P is a *residue program* modulo m with input $\boldsymbol{s}$ and output $\boldsymbol{r}$ iff P realizes a function $f : 2^{\mathrm{A}(\boldsymbol{s})} \to 2^{\mathrm{A}(\boldsymbol{r})}$ such that, for each $I \subseteq \mathrm{A}(\boldsymbol{s})$ at which $\boldsymbol{s}$ is sorted, $\boldsymbol{r}$ is sorted at $f(I)$ and $(\mathbf{1}^{\mathrm{T}}\boldsymbol{r})(f(I)) = (\mathbf{1}^{\mathrm{T}}\boldsymbol{s})(I) \bmod m$. Moreover, for any program $P' \supseteq P$, we assume that $(\mathrm{A}(P) \cup \mathrm{A}(\boldsymbol{r})) \cap \mathrm{H}(P' \setminus P) \subseteq \mathrm{A}(\boldsymbol{s})$.

▶ **Example 13.** A residue program modulo m with input $\boldsymbol{s} = [s_1; \dots; s_n]$ and output $\boldsymbol{r} = [r_1; \dots; r_k]$ for $k = \min\{n, m-1\}$, generalizing the design displayed on the left of Figure 2, is given by

$$\begin{array}{lll} r_j \coloneq s_{qm+j}, \text{ not } s_{(q+1)m}. & \text{for } 0 \leq q < \lfloor n/m \rfloor & \text{and } 1 \leq j < m, \\ r_j \coloneq s_{qm+j}. & \text{for } q = \lfloor n/m \rfloor & \text{and } 1 \leq j \leq n - qm. \end{array}$$

▶ **Lemma 14.** *Let $O = (P, d + \alpha\mathbf{1}^{\mathrm{T}}\boldsymbol{s})$ be a multi-unary rewrite of an expression e with output $\{\boldsymbol{s}\}$, and P' be a residue program modulo m with input $\boldsymbol{s}$ and output $\boldsymbol{r}$ such that $\mathrm{A}(O)$ and $P' \cup \{(a, \emptyset, \emptyset) \mid a \in (\mathrm{A}(P') \cup \mathrm{A}(\boldsymbol{r})) \cap (\mathrm{A}(e) \setminus \mathrm{A}(\boldsymbol{s}))\}$ fit. Then, $(P \cup P', d + m\alpha\mathbf{1}^{\mathrm{T}}\boldsymbol{q} + \alpha\mathbf{1}^{\mathrm{T}}\boldsymbol{r})$ is a multi-unary rewrite of e with output $\{\boldsymbol{q}, \boldsymbol{r}\}$, where $\boldsymbol{q}$ is the quotient of $\boldsymbol{s}$ divided by m.*

We are now ready to augment digit-wise rewrites according to Proposition 10 with carry propagation, and call the resulting scheme (unique) *mixed-radix rewrite*. Such a rewrite resembles (parts of) the CNF encoding of PB expressions given in [21], when built from Batcher's odd-even sorting and merging networks [8].

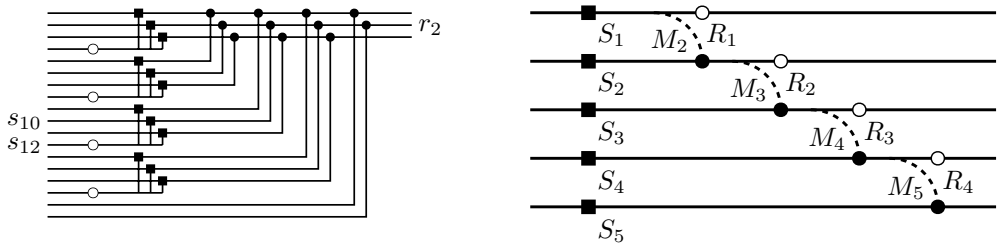

Figure 2 The residue program from Example 13 for $n = 18$ and $m = 4$ (left), with ○, ■, and ● standing for logical *not*s, *and*s, and *or*s, and the layout of a mixed-radix rewrite as in Proposition 15 in a base of length $k = 5$ (right), with ○, ■, and ● standing for R_i, S_i, and M_i, and dashed lines for quotients $\boldsymbol{q}_i$.

▶ **Proposition 15.** *Let* W *be a mixed-radix decomposition in a base* $(\mathbf{b}, \boldsymbol{\pi})$ *with radices* $\mathbf{b} = [b_1; \ldots; b_k]$ *and place values* $\boldsymbol{\pi} = [\pi_1; \ldots; \pi_k]$ *such that* $\mathrm{W}^{\mathrm{T}}\boldsymbol{\pi} = \mathbf{w}$ *for an expression* $e = \mathbf{w}^{\mathrm{T}}\boldsymbol{l}$*. For* $1 \leq i \leq k$*, let* $\mathbf{w}_i$ *be the* i*th row of* W*,* $\boldsymbol{t}_i$ *be some vector such that* $\mathbf{1}^{\mathrm{T}}\boldsymbol{t}_i \equiv \mathbf{w}_i^{\mathrm{T}}\boldsymbol{l}$*, and*

1. S_i *be a sorting program with input* $\boldsymbol{t}_i$ *and output* $\boldsymbol{h}_i$*,*
2. M_i *be a merging program with inputs* $\boldsymbol{h}_i, \boldsymbol{q}_i$ *and output* $\boldsymbol{s}_i$*, where* $\boldsymbol{q}_i$ *is the quotient of* $\boldsymbol{s}_{i-1}$ *divided by* b_{i-1}*, provided that* $1 < i$*,*
3. R_i *be a residue program modulo* b_i *with input* $\boldsymbol{s}_i$ *and output* $\boldsymbol{r}_i$*, provided that* $i < k$*,*

such that $\mathrm{A}(e)$ *and* $S'_1, \ldots, S'_{i-1}, S'_{i+1}, \ldots, S'_k, S'_i, M'_1, R'_1, \ldots, M'_k, R'_k$ *fit, where* $M_1 = R_k = \emptyset$*,* $\boldsymbol{s}_1 = \boldsymbol{h}_1$*,* $\boldsymbol{r}_k = \boldsymbol{s}_k$*, and* $P' = P \cup \{(a, \emptyset, \emptyset) \mid a \in (\mathrm{A}(P) \cup V_2) \cap (\mathrm{A}(e) \setminus V_1)\}$ *for any program* P *with atoms* V_1 *and* V_2 *in its input(s) or output, respectively. Then,* $(\bigcup_{i=1}^{k}(S_i \cup M_i \cup R_i), \sum_{i=1}^{k} \pi_i \mathbf{1}^{\mathrm{T}}\boldsymbol{r}_i)$ *is a multi-unary rewrite of* e *with output* $\{\boldsymbol{r}_1, \ldots, \boldsymbol{r}_k\}$*.*

The principal design of such a mixed-radix rewrite is visualized on the right of Figure 2. Using sorting and merging programs according to [8], $\mathrm{c}_{\mathrm{sort}}(n) = n(\log n)^2$ and $\mathrm{c}_{\mathrm{merge}}(n) = n \log n$ limit the size of each sorting or merging program, respectively, needed to rewrite a mixed-radix decomposition of $\mathbf{w} = [w_1; \ldots; w_n]$ in a binary base, while each residue program (modulo 2) requires linear size. As $k = \lceil \log \max\{w_1, \ldots, w_n\} \rceil$ bounds the number of programs, the resulting size is of the order $kn(\log n)^2$, which matches the CNF encoding of PB expressions given in [21].

4 Experiments

For evaluating the effect of optimization rewriting, we implemented the rewriting strategies described above in the tool LP2NORMAL (2.27),[1] and ran the ASP solver CLASP (3.1.4) [22], using its "trendy" configuration for a single thread per run, on a cluster of Linux machines equipped with Intel Xeon E5-4650 2.70GHz processors. All of the applied optimization rewrites are primarily based on sorting programs, built from (normal) ASP encodings of Batcher's odd-even merging networks [8, 11], or alternatively from merging programs that do not introduce auxiliary atoms whenever the sum of required atoms and rules is reduced in this way. Moreover, each merging program is enhanced by (redundant) integrity constraints asserting the implication from a consecutive output atom to its predecessor, groups of sorting programs are compressed by means of structure sharing [10], and rewritings are pruned by

[1] Available with benchmarks at `http://research.ics.aalto.fi/software/asp`.

Table 1 Impact of optimization rewriting on solving performance.

Connected Still-Life

	cons	time	conf	22	28	55	15
–	3.6	3.4	7.9	O	S	S	S
64	4.1	**1.0**	**5.3**	**O**	**O**	**O**	S
so	4.2	1.2	5.4	O	O	S	S

Crossing Minimization

	cons	time	conf	50	21	1	5	8
–	3.8	2.9	7.6	O	S	S	S	S
64	4.5	**0.8**	**4.8**	**O**	**O**	S	**O**	S
so	4.6	1.4	5.2	O	O	O	S	S

Maximal Clique

	cons	time	conf	51	92	10	33
–	5.9	2.9	6.6	O	S	S	S
64	6.0	**1.3**	**5.2**	**O**	**O**	**O**	S
so	6.1	1.7	**5.2**	O	O	S	S

Timetabling

	cons	time	conf	24	12	1	2	1	12	1	2	2
–	5.0	2.1	6.1	O	S	O	O	O	S	S	S	S
64	6.2	**1.9**	**5.0**	**O**	**O**	**O**	**O**	S	S	S	S	M
so	6.9	2.8	5.4	O	O	S	T	S	S	T	M	M

Bayes Alarm

	cons	time	conf	5	3	1	1	22
–	4.3	1.0	5.7	O	O	O	S	S
l1	5.3	0.6	4.7	O	O	O	S	S
l2	5.5	**0.3**	4.4	**O**	**O**	**O**	**O**	S
l3	5.6	**0.3**	**4.1**	**O**	**O**	**O**	**O**	S
g7	6.5	2.2	5.1	O	O	S	S	S
mr	6.8	2.5	5.2	O	S	S	S	S

Timetabling

	cons	time	conf	24	1	12	1	1	1	11	1	1	4
–	5.0	**2.6**	6.6	O	O	S	O	O	O	S	S	S	S
l1	6.5	2.9	**6.1**	O	S	O	S	O	O	S	S	S	M
l2	6.6	3.1	6.2	**O**	**O**	**O**	**O**	**O**	T	S	S	T	M
l3	6.7	3.1	6.4	O	O	O	O	S	T	S	S	T	M
g7	5.4	2.7	6.7	O	O	S	O	O	O	S	S	S	S
mr	6.9	2.9	6.4	O	O	O	S	T	T	S	T	T	M

Bayes Water

	cons	time	conf	15	3	1	4	4
–	3.0	2.8	7.4	O	S	S	S	S
l1	4.6	2.0	6.2	O	O	S	S	S
l2	4.8	1.8	5.8	O	O	O	S	S
l3	4.9	**1.6**	5.5	**O**	**O**	**O**	**O**	S
g7	5.5	2.1	**5.2**	O	O	O	S	S
mr	5.8	2.3	**5.2**	O	O	O	S	S

Markov Network

	cons	time	conf	19	4	2	1	1	1	2	42
–	4.1	2.0	6.6	**O**	**O**	**O**	S	S	S	**O**	S
l1	4.9	2.0	6.1	O	O	O	O	S	S	S	S
l2	5.0	**1.8**	**5.8**	**O**	**O**	**O**	**O**	S	**O**	S	S
l3	5.2	2.0	**5.8**	**O**	**O**	**O**	**O**	**O**	S	S	S
g7	5.7	2.7	6.1	O	O	S	S	S	S	S	S
mr	6.1	3.1	6.2	O	S	S	S	S	S	S	S

Bayes Hailfinder

	cons	time	conf	31	1	3	10	2	1	1	2
–	4.0	**1.4**	5.9	O	O	O	S	S	S	S	S
l1	5.3	1.6	**5.0**	O	O	O	O	S	S	S	S
l2	5.4	1.7	**5.0**	**O**	**O**	**O**	**O**	**O**	S	**O**	S
l3	5.5	1.9	5.1	**O**	**O**	**O**	**O**	**O**	**O**	S	S
g7	6.2	3.2	5.4	O	S	S	S	S	S	S	S
mr	6.5	3.0	5.3	O	O	S	S	S	S	S	S

dropping rules not needed to represent optimization values below or near the value of the first stable model found by CLASP in a (short) trial run skipping optimization.

Table 1 provides experimental results for six benchmark classes. Columns headed by numbers partition the instances of a class based on solving performance: an entry "O" expresses that optima were found and proven for the respective number of instances within 10,800s time and 16GB memory limit per run; "S" means that some solutions have been obtained, yet not proven optimal; "T" marks that no solution was reported in time; and "M" indicates aborts due to memory excess. For each class, columns "cons", "time", and "conf" give the decimal logarithms of the averages of numbers of constraints, seconds of CPU time, and conflicts reported by CLASP with respect to rewriting strategies indicated in rows. The constraints are averaged over instances on which no rewriting strategy under consideration aborted due to memory excess, and the time and conflicts are averaged over instances solved optimally with respect to all strategies. That is, accumulated runtimes and conflicts refer to

the instances in a corresponding column consisting of "O" entries only, while runs without proven optima are not included. Smallest CPU times and numbers of conflicts are highlighted in boldface, and likewise the "O" entries of rewriting strategies leading to most optimally solved instances for a class.

The four benchmark classes in the upper part of Table 1, Connected Still-Life, Crossing Minimization, and Maximal Clique from the ASP Competition [15, 23] along with Curriculum-based Course Timetabling [7, 12], involve optimization statements with unit weights $w_i = 1$ or few groups of non-unit weights, respectively, in case of Timetabling. Hence, these classes lend themselves to sorting inputs in digit-wise layers, as described in Sections 2.3 and 3.3, where unit weights yield a single layer, so that sorting programs produce a unique representation of their sum as a unary number. This strategy is denoted by "so", its bounded application to equal-weight chunks of up to $t = 64$ literals by "64", and no rewriting at all by "–".

Comparing these three approaches, we observe that the bounded strategy dominates in terms of optimally solved instances as well as runtimes and conflicts over the subsets of instances solved optimally with respect to all three strategies. The edge over plain CLASP without rewriting is particularly remarkable and amounts to 83 more optimally solved instances for Connected Still-Life, 26 for Crossing Minimization, 102 for Maximal Clique, and still 11 for Timetabling, considering that mixed-radix decompositions obtained via digit-wise sorting (without carry propagation) are not necessarily unique for the latter class. While the unbounded strategy also yields substantial improvements relative to plain CLASP, it incurs a significantly larger size increase that does not pay off by reducing search conflicts any further than the bounded approach, and thus does not lead to more optimally solved instances either. That is, the introduction of sorting programs and atoms for bounded unary numbers, capturing parts of optimization statements in separation, already suffices to counteract combinatorial explosion due to optimization statements to the extent feasible.

The five benchmark suites in the lower part of Table 1 stem from three classes, Bayesian Network Learning [20, 24] with samples from three data sets, Markov Network Learning [19, 25], and Curriculum-based Course Timetabling again. The corresponding instances feature non-unit weights amenable to mixed-radix rewrites, as presented in Section 3.4, which yield a unique mixed-radix decomposition of the sum of input weights. We denote this approach by "mr", and in addition consider selective strategies based on matrix partitioning according to Section 3.3: limiting mixed-radix rewrites to a number t of locally most significant digits per weight, indicated by "l1" for $t = 1$, "l2" for $t = 2$, and "l3" for $t = 3$, as well as "g7" dropping the $t = 7$ least significant digits to rewrite the globally most significant digits of each weight only. The baseline of plain CLASP without rewriting is again marked by "–".

Regarding these approaches, we observe that the strategies denoted by "l2" and "l3", focusing on locally most significant digits, constitute a good tradeoff between size increase and reduction of conflicts, which in turn leads to more optimally instances solved than other rewriting strategies and plain CLASP. In fact, full rewriting "mr" blows up size more than (additionally) facilitating search, the global strategy "g7" is not flexible enough to encompass all diverging non-unit weights in an optimization statement, and plain CLASP cannot draw on rewrites to learn more effective no-goods. This becomes particularly apparent on the Bayes Hailfinder instances, where "l2" and "l3" yield 13 more optimally solved instances than plain CLASP, 16 more than "mr", and 17 more than "g7". For the other classes, Markov Network and Timetabling, the distinction is less clear and amounts to singular instances separating the local strategies "l2" and "l3" from each other as well as plain CLASP or full mixed-radix rewriting "mr", respectively.

In comparison to the bounded digit-wise sorting approach denoted by "64", also applied to Timetabling in the upper part of Table 1, the best-performing strategy "l2" based on

mixed-radix rewrites leads to the same number of optimally solved instances, yet incurring two timeouts more. The latter observation indicates that the search of CLASP does not truly benefit from the unique representation of a sum of weights in relation to just producing unary numbers capturing sums of digits. We conjecture that this is due to the non-monotonic character of residue programs, while sorting and merging programs map inputs to outputs in a monotonic fashion. This suggests trying alternative rewriting approaches that avoid residue programs while dealing with diverging non-unit weights, and such methods are future work.

5 Related Work

In previous work, we have addressed the normalization of cardinality rules [11] and weight rules [10], and this paper extends the investigation of rewriting techniques to optimization statements. While normalization allows for completely eliminating cardinality and weight rules, optimization rewriting maps one expression to another, where the introduced atoms and rules add structure that provides new opportunities for ASP solvers to learn no-goods, which may benefit solving performance.

Mixed-radix rewrites, as presented in Section 3.4, resemble the CNF encoding of PB expressions given in [21]. More recent translation methods [6, 10, 29] use so-called tares to simplify bound checking for mixed-radix decompositions of PB constraints, while tares are not meaningful for optimization statements that lack fixed bounds. The hybrid CNF encoding of cardinality constraints in [9] compensates weak propagation by (small) partial building blocks, which is comparable to the selective rewriting techniques in Section 3.3. Dynamic approaches to limit the size of CNF encodings of PB expressions, complementing selective rewriting strategies, include conflict-directed lazy decomposition [3], where digit-wise rewriting is performed selectively during search. A related strategy [2] consists of fully rewriting expressions deemed relevant in PB solving. Moreover, CNF encodings of PB optimization statements can be simplified when creating them incrementally during search [28]. In contrast to such dynamic approaches, we aim at preprocessing ASP optimization statements prior to solving. As a consequence, our rewriting techniques can be flexibly combined with different optimization strategies [22, 4].

6 Conclusions

Our work extends the scope of normalization methods, as originally devised for cardinality and weight rules, by developing rewriting techniques for optimization statements as well. In this context, sorting programs serve as basic building blocks for representing sums of unit weights or digits as unary numbers. When dealing with non-unit weights, merging unary numbers amounts to carry propagation, so that residues yield a unique mixed-radix decomposition of the sum of input weights. Our rewriting strategies can be applied selectively based on partitioning a vector of weights or a corresponding matrix, respectively. Such partial optimization rewrites allow for reducing the size needed to augment answer-set programs with additional structure in order to enhance the performance of ASP solvers.

Experiments with the ASP solver CLASP showed substantially improved robustness of its model-guided optimization strategy, used by default, due to optimization rewriting. This particularly applies to benchmarks involving unit weights or moderately many groups of non-unit weights, respectively. Sorting here effectively counteracts the combinatorial explosion faced without rewriting, as no-goods over unary numbers capture much larger classes of interpretations than those stemming from optimization statements over comparably

specific atoms. Mixed-radix decompositions and carry propagation helped to improve solving performance for benchmarks with non-unit weights as well, yet not by the same amount as sorting equal weights, provided equal weights exist. We conjecture that this observation is related to the non-monotonic character of residue programs, and investigating alternative approaches avoiding them is part of future work. The latter also includes in-depth experiments with core-guided optimization strategies, which in preliminary tests seemed unimpaired by rewriting, neither positively nor negatively. Finally, our experiments indicated that selective rewriting techniques require significantly less size than full rewriting to reduce search conflicts equally well, so that predefining effective adaptive selection strategies is of interest.

References

1 Ignasi Abío, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Valentin Mayer-Eichberger. A new look at BDDs for Pseudo-Boolean constraints. *Journal of Artificial Intelligence Research*, 45:443–480, 2012. doi:10.1613/jair.3653.

2 Ignasi Abío, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Peter J. Stuckey. To encode or to propagate? The best choice for each constraint in SAT. In *Proceedings of CP 2013*, volume 8124 of *LNCS*, pages 97–106. Springer, 2013. doi:10.1007/978-3-642-40627-0_10.

3 Ignasi Abío and Peter J. Stuckey. Conflict directed lazy decomposition. In *Proceedings of CP 2012*, volume 7514 of *LNCS*, pages 70–85. Springer, 2012. doi:10.1007/978-3-642-33558-7_8.

4 Mario Alviano, Carmine Dodaro, Nicola Leone, and Francesco Ricca. Advances in WASP. In Calimeri et al. [16], pages 40–54. doi:10.1007/978-3-319-23264-5_5.

5 Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Cardinality networks: A theoretical and empirical study. *Constraints*, 16(2):195–221, 2011. doi:10.1007/s10601-010-9105-0.

6 Olivier Bailleux, Yacine Boufkhad, and Olivier Roussel. New encodings of Pseudo-Boolean constraints into CNF. In *Proceedings of SAT 2009*, volume 5584 of *LNCS*, pages 181–194. Springer, 2009. doi:10.1007/978-3-642-02777-2_19.

7 Mutsunori Banbara, Takehide Soh, Naoyuki Tamura, Katsumi Inoue, and Torsten Schaub. Answer set programming as a modeling language for course timetabling. *Theory and Practice of Logic Programming*, 13(4-5):783–798, 2013. doi:10.1017/S1471068413000495.

8 Kenneth E. Batcher. Sorting networks and their applications. In *Proceedings of AFIPS 1968*, pages 307–314. ACM, 1968. doi:10.1145/1468075.1468121.

9 Yael Ben-Haim, Alexander Ivrii, Oded Margalit, and Arie Matsliah. Perfect hashing and CNF encodings of cardinality constraints. In *Proceedings of SAT 2012*, volume 7317 of *LNCS*, pages 397–409. Springer, 2012. doi:10.1007/978-3-642-31612-8_30.

10 Jori Bomanson, Martin Gebser, and Tomi Janhunen. Improving the normalization of weight rules in answer set programs. In *Proceedings of JELIA 2014*, volume 8761 of *LNCS*, pages 166–180. Springer, 2014. doi:10.1007/978-3-319-11558-0_12.

11 Jori Bomanson and Tomi Janhunen. Normalizing cardinality rules using merging and sorting constructions. In *Proceedings of LPNMR 2013*, volume 8148 of *LNCS*, pages 187–199. Springer, 2013. doi:10.1007/978-3-642-40564-8_19.

12 Alex Bonutti, Fabio De Cesco, Luca Di Gaspero, and Andrea Schaerf. Benchmarking curriculum-based course timetabling: Formulations, data formats, instances, validation, visualization, and results. *Annals of Operations Research*, 194(1):59–70, 2012. doi:10.1007/s10479-010-0707-0.

13 Gerhard Brewka, Thomas Eiter, and Mirosław Truszczyński. Answer set programming at a glance. *Communications of the ACM*, 54(12):92–103, 2011. doi:10.1145/2043174.2043195.

14 Francesco Calimeri, Wolfgang Faber, Martin Gebser, Giovambattista Ianni, Roland Kaminski, Thomas Krennwallner, Nicola Leone, Francesco Ricca, and Torsten Schaub. ASP-Core-2: Input language format. Available at https://www.mat.unical.it/aspcomp2013/ASPStandardization/, 2012.

15 Francesco Calimeri, Martin Gebser, Marco Maratea, and Francesco Ricca. Design and results of the fifth answer set programming competition. *Artificial Intelligence*, 231:151–181, 2016. doi:10.1016/j.artint.2015.09.008.

16 Francesco Calimeri, Giovambattista Ianni, and Miroslaw Truszczynski, editors. *Proceedings of LPNMR 2015*, volume 9345 of *LNCS*. Springer, 2015. doi:10.1007/978-3-319-23264-5.

17 Broes De Cat, Bart Bogaerts, Maurice Bruynooghe, Gerda Janssens, and Marc Denecker. Predicate logic as a modelling language: The IDP system. Available at https://arxiv.org/abs/1401.6312, 2016.

18 Michael Codish, Yoav Fekete, Carsten Fuhs, and Peter Schneider-Kamp. Optimal base encodings for Pseudo-Boolean constraints. In *Proceedings of TACAS 2011*, volume 6605 of *LNCS*, pages 189–204. Springer, 2011. doi:10.1007/978-3-642-19835-9_16.

19 Jukka Corander, Tomi Janhunen, Jussi Rintanen, Henrik J. Nyman, and Johan Pensar. Learning chordal Markov networks by constraint satisfaction. In *Proceedings of NIPS 2014*, pages 1349–1357. NIPS Foundation, 2013.

20 James Cussens. Bayesian network learning with cutting planes. In *Proceedings of UAI 2011*, pages 153–160. AUAI, 2011.

21 Niklas Eén and Niklas Sörensson. Translating Pseudo-Boolean constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1-4):1–26, 2006.

22 Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, 187-188:52–89, 2012. doi:10.1016/j.artint.2012.04.001.

23 Martin Gebser, Marco Maratea, and Francesco Ricca. The design of the sixth answer set programming competition. In Calimeri et al. [16], pages 531–544. doi:10.1007/978-3-319-23264-5_44.

24 Tommi S. Jaakkola, David A. Sontag, Amir Globerson, and Marina Meila. Learning Bayesian network structure using LP relaxations. In *Proceedings of AISTATS 2010*, pages 358–365. JMLR Proceedings, 2010.

25 Tomi Janhunen, Martin Gebser, Jussi Rintanen, Henrik J. Nyman, Johan Pensar, and Jukka Corander. Learning discrete decomposable graphical models via constraint optimization. *Statistics and Computing*, online access, 2015. doi:10.1007/s11222-015-9611-4.

26 Tomi Janhunen and Ilkka Niemelä. Applying visible strong equivalence in answer-set program transformations. In *Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, volume 7265 of *LNCS*, pages 363–379. Springer, 2012. doi:10.1007/978-3-642-30743-0_24.

27 Nicola Leone, Gerald Pfeifer, Wolfgang Faber, Thomas Eiter, Georg Gottlob, Simona Perri, and Francesco Scarcello. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic*, 7(3):499–562, 2006. doi:10.1145/1149114.1149117.

28 Panagiotis Manolios and Vasilis Papavasileiou. Pseudo-Boolean solving by incremental translation to SAT. In *Proceedings of FMCAD 2011*, pages 41–45. FMCAD Inc., 2011.

29 Norbert Manthey, Tobias Philipp, and Peter Steinke. A more compact translation of Pseudo-Boolean constraints into CNF such that generalized arc consistency is maintained. In

Proceedings of KI 2014, volume 8736 of *LNCS*, pages 123–134. Springer, 2014. doi:10.1007/978-3-319-11206-0_13.

30 Olivier Roussel and Vasco M. Manquinho. Pseudo-Boolean and cardinality constraints. In *Handbook of Satisfiability*, pages 695–733. IOS, 2009. doi:10.3233/978-1-58603-929-5-695.

31 Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002. doi:10.1016/S0004-3702(02)00187-X.