An Efficient Incremental Algorithm for Solving Systems of Linear Diophantine Equations

Evelyne Contejean
Laboratoire de Recherche en Informatique
URA 410 du CNRS
Bâtiment 490, 91 405 Orsay Cedex, France
&
Max-Planck-Institut für Informatik
Im Stadtwald
D-6600 Saarbrücken, Germany

and

Hervé Devie
Laboratoire de Recherche en Informatique
URA 410 du CNRS
Bâtiment 490, 91 405 Orsay Cedex, France

Abstract

In this paper, we describe an algorithm for solving systems of linear Diophantine equations based on a generalization of an algorithm for solving one equation due to Fortenbacher [3]. It can solve a system as a whole, or be used incrementally when the system is a sequential accumulation of several subsystems. The proof of termination of the algorithm is difficult, whereas the proofs of completeness and correctness are straightforward generalizations of Fortenbacher's proof.

1 Introduction

Linear Diophantine equations occur frequently in mathematics and computer science, namely in the decision procedure of the accessibility property for Petri-Nets, in associative-commutative unification [1], in constrained logic programming, in the vectorization of FORTRAN programs, *etc.*

By definition, a Diophantine equation has the form $Q = 0$, where Q is a

polynomial with integer coefficients. The equation is homogeneous if Q has no constant and linear if all monomial have the form kx_i where $k \in \mathbb{Z}$. A solution is any vector of natural numbers which equates Q to 0. The set $\mathcal{S}$ of solutions of S is a (generally infinite) additive monoïd, which can always be represented by a finite set $\mathcal{B}$ called a *basis*. By definition, the basis is a minimal subset of $\mathcal{S}$ with respect to the inclusion ordering such that every solution in $\mathcal{S}$ can be expressed as a linear combination of the solutions in $\mathcal{B}$, by using natural numbers as coefficients.
It can be easily shown that $\mathcal{B}$ is the subset of $\mathcal{S}$ made of all solutions in $\mathcal{S}$ different from the trivial solution $(0, \ldots, 0)$ which are minimal with respect to the following ordering $\gg$ on tuples of natural numbers :

$$(a_1, \ldots, a_q) \gg (b_1, \ldots, b_q) \quad \text{if} \quad \begin{array}{l} \forall i \in [1..q], a_i \geq b_i \\ \text{and } \exists i \in [1..q], a_i \neq b_i \end{array}$$

For this reason, $\mathcal{B}$ is called the set of *minimal solutions* of S.

The case where the system S is reduced to a single equation has been thoroughly investigated during the last decade. Two main approaches can be distinguished: the first searches the space of p-tuples within bounds on minimal solutions obtained by Huet [10] and refined by Lambert [11]. All tuples below these bounds must be checked. Due to Fortenbacher, the second searches the space of p-tuples while trying to minimize the absolute value of Q. Unfortunately neither method is well-suited to solve simultaneously several Diophantine equations.

Let S be the system $Q = 0 \wedge S'$ of n equations, and $\mathcal{B}_Q$ and $\mathcal{B}'$ the respective sets of minimal solutions of $Q = 0$ and S'. Since solutions of S are linear combinations of solutions in $\mathcal{B}_Q = \{s_1, \ldots, s_k\}$ we can substitute such combinations $c_1 s_1 + \ldots + c_k s_k$ in S' resulting in a new system S'' of $(n-1)$ equations whose variables are the coefficients $c_1, \ldots, c_k$. Solving S'' yields a basis $\mathcal{B}''$ for the solutions of S''. Substituting back in the combinations $c_1 s_1 + \ldots + c_k s_k$ yields a set $\mathcal{B}$ of solutions for S. Finally, as the set of minimal solutions of S is included in $\mathcal{B}$, $\mathcal{B}$ must be cleaned up. This method, that we call Gaussian elimination, solves linear Diophantine equations one at a time and, consequently, it can use either one of Huet's and Fortenbacher's algorithms. However, it has three severe drawbacks : first, the algorithm for solving one Diophantine equation is called as many times as there are equations in S; second, the number of variables in the new system S'' is equal to the cardinality of $\mathcal{B}_Q$, an exponential of the value of the coefficients of the monomials in Q in the worst case, which can quickly lead to untractable computations; third, the values of the solutions in the basis themselves grow, which causes the values of the coefficients in the successive systems to grow; fourth, $\mathcal{B}$ contains some useless (i.e. non minimal) solutions. In practice, the second and third drawbacks forbid solving systems containing too many equations [See the example in section 11.4].

Based on a simple geometric interpretation of Fortenbacher's method, our algorithm was the first to solve a system as a whole [4, 5]. Pottier then described a variation of our algorithm. Two other completely different algorithms were also later proposed by Domenjoud [7] and Pottier [12].

Mathematical puzzles often lead to solving linear Diophantine systems. Gardner suggested once the following [9]: 5 sailors and a monkey escape from a naufrage and reach an Island with coconuts. Before dawn, they gather a few of them and decide to sleep first and share the next day. At night however, a first of them awakes, counts the nuts, makes five parts, gives the only remaining nut to the monkey, saves his share away and sleeps back in. The other four in turn wake up and do the same. When they all wake up in the morning, they again count the nuts, divide them into five parts take their share and give the last remaining nut to the monkey. How many nuts were there at the beginning? Let x_i be the number of nuts taken away by the ith sailor, x_6 be the number of nuts obtained by each of them at the last sharing, and x_0 the total number of nuts. These integer variables are solutions of the following linear Diophantine system:

$$\left\{ \begin{array}{rcl} x_0 & = & 5x_1 + 1 \\ 4x_1 & = & 5x_2 + 1 \\ 4x_2 & = & 5x_3 + 1 \\ 4x_3 & = & 5x_4 + 1 \\ 4x_4 & = & 5x_5 + 1 \\ 4x_5 & = & 5x_6 + 1 \end{array} \right.$$

Since these equations are not homogeneous, the set of solutions is obtained by adding a minimal solution of the above system to an arbitrary solution of the homogeneous part of the system. We will see that our algorithm can handle non-homogeneous systems in a straightforward way. In the above case, our algorithm finds the general solution of the above system ($x_0 = -4 + k * 5^6$) in 197 seconds. On the other hand, solving this system by using Gaussian elimination is simply impossible for the reasons indicated above. This ability of our method to solve non-homogeneous systems is part of the flexibility of our algorithm: we will describe variants for solving linear Diophantine systems together with various ordering constraints. In addition, these algorithms can all be used within constraint logic programming applications, since they are *incremental*: solving the system $S_1 \wedge S_2$ can be done in two steps: solve S_1 first, then use the minimal solutions of S_1 to solve S_2 with a modified version of our algorithm. Of course, this modified version does not use Gaussian elimination.

The paper is organized as follows: Section 2 introduces the notations. The geometric interpretation in a one-dimension space of Fortenbacher's algorithm for solving one equation is described in section 3. Its generalization to a p-dimension space is discussed in section 4. The dimension p turns out to be the number of equations to be solved. This yields a new algorithm which solves systems of linear Diophantine equations by using computations on vectors. The proof of the basic version of the algorithm is given in section 5, and several improvements are discussed in section 6. The cases of non-homogeneous systems and more general constrained systems are presented in sections 7 and 8 respectively. The incremental version of the algorithm is given in section 9, which yields an incremental satisfiability test described in section 10. Finally, a short comparison with Domenjoud's and Pottier's algorithms is done in section 11.

2 Basic notations

An homogeneous linear Diophantine system with p equations and q variables may be written

$$\left\{\begin{array}{ccccccc} a_{11}x_1 & +\ldots+ & a_{1j}x_j & +\ldots+ & a_{1q}x_q & = & 0 \\ \vdots & & \vdots & & \vdots & & \vdots \\ a_{i1}x_1 & +\ldots+ & a_{ij}x_j & +\ldots+ & a_{iq}x_q & = & 0 \\ \vdots & & \vdots & & \vdots & & \vdots \\ a_{p1}x_1 & +\ldots+ & a_{pj}x_j & +\ldots+ & a_{pq}x_q & = & 0 \end{array}\right.$$

where $a_{ij\ (1\leq i\leq p, 1\leq j\leq q)}$ is an integer. We denote by $e_{j\ (1\leq j\leq q)}$ the j^{th} vector in the canonical basis of N^q : $e_j = (\underbrace{0,\ldots,0}_{j-1\ times},1,0,\ldots,0)$. Let a be the linear mapping whose matrix in the canonical basis $(e_1,\ldots,e_q)$ is $(a_{ij})_{(1\leq i\leq p, 1\leq j\leq q)}$. The j^{th} column vector of the matrix, that is $a(e_j)$ is called the j^{th} *basic default vector* or *default vector of* x_j. Thus, for a tuple $x=(x_1,\ldots,x_q)$ of N^q, $a(x)$ is the vector

$$a(x) = \left(\begin{array}{ccc} a_{11}x_1 & +\ldots+ & a_{1q}x_q \\ \vdots & & \vdots \\ a_{p1}x_1 & +\ldots+ & a_{pq}x_q \end{array}\right) = x_1 a(e_1) + \cdots + x_q a(e_q) \quad (*)$$

In the following, we will freely refer to this definition of $a(x)$ as a sum of default vectors. $a(x)$ is therefore a vector, the vector $\vec{OM}$, where O is the origin of the p-dimension space. Note that the tuple x can be seen as a mechanism for selecting particular default vectors (with possibly repetitions), and we will sometimes identify x with the vectors it selects. We will also use $\cdot$ for the scalar product and $\|\ \|$ for the Euclidean norm.

3 A geometric interpretation of Fortenbacher's algorithm

Consider a single equation $a_1x_1+\ldots+a_qx_q=0$. Fortenbacher's idea is to search for the minimal solutions starting from the tuples in the canonical

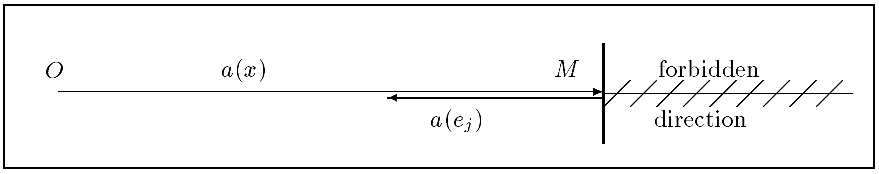


Figure 1: A geometric interpretation of Fortenbacher's restriction.

basis of N^q. Suppose that the current q-tuple $(x_1, \ldots, x_q)$ is not yet a solution. It can be non deterministically increased component by component until it becomes a solution or greater than a solution. It turns out that the following restriction can drastically decrease the search space without loosing completeness:

if $a_1x_1 + \ldots + a_qx_q < 0$, then increase by 1 some x_j such that $a_j > 0$;
if $a_1x_1 + \ldots + a_qx_q > 0$, then increase by 1 some x_j such that $a_j < 0$.

Fortenbacher's restriction can be expressed by the following simple condition:

$$(C_1) \qquad \text{increase by 1 some } \; x_j \; \text{ satisfying } \; a(x) \times a(e_j) < 0$$

whose geometric interpretation is displayed below:

Fortenbacher's method consists in choosing between different possibilities the vectors $a(e_j)$ heading to the "right direction", that is, the origin. The notion of "right direction" is easy in the case of one equation because vectors are in $\mathbb{Z}$. But, for a system of several equations, the image of a tuple by a lies in $\mathbb{Z}^p$ and the notion of "right direction" must be made precise.

4 A generalization of Fortenbacher's algorithm

In the case of an equation as well as in the case of a system of equations, the value of $a(x)$ given in (*) shows that any solution of the system can be seen as a collection of default vectors whose sum is $\vec{0}$. Choosing an arbitrary order among these vectors allows to construct a sequence of default vectors starting from the origin and returning to the origin. The basic idea underlying the algorithm is therefore a stepwise construction of such sequences, starting from the empty sequence, and such that new default vectors are non-deterministically added until a solution is eventually found or no minimal solution can be obtained. Different sequences may however correspond to the same solution (up to permutation of vectors in the sequence). Fortenbacher's restriction enables to restrict the search and therefore to eliminate most but not all redundant sequences. We generalize

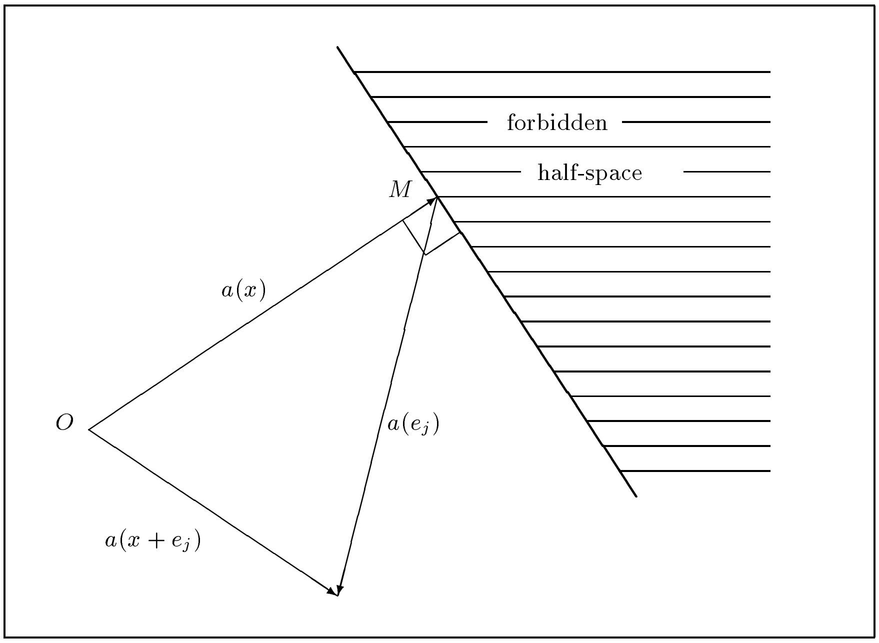


Figure 2: A generalization of Fortenbacher's restriction.

Fortenbacher's restriction for systems of equations by forbidding at each step in the construction of a sequence the half-space which does not contain the origin, as shown in figure 2.

Formally, a tuple x (denoting a sequence of default vectors), whose sum $a(x)$ is non-null can be increased by 1 on its j^{th} component provided $a(x + e_j) = a(x) + a(e_j)$ lies in the half-space containing the origin and delimited by the affine hyperplan perpendicular to the vector $a(x)$ at its extremity when originating from the origin O of the space.

Mathematically, this constraint can be expressed by the following simple condition:

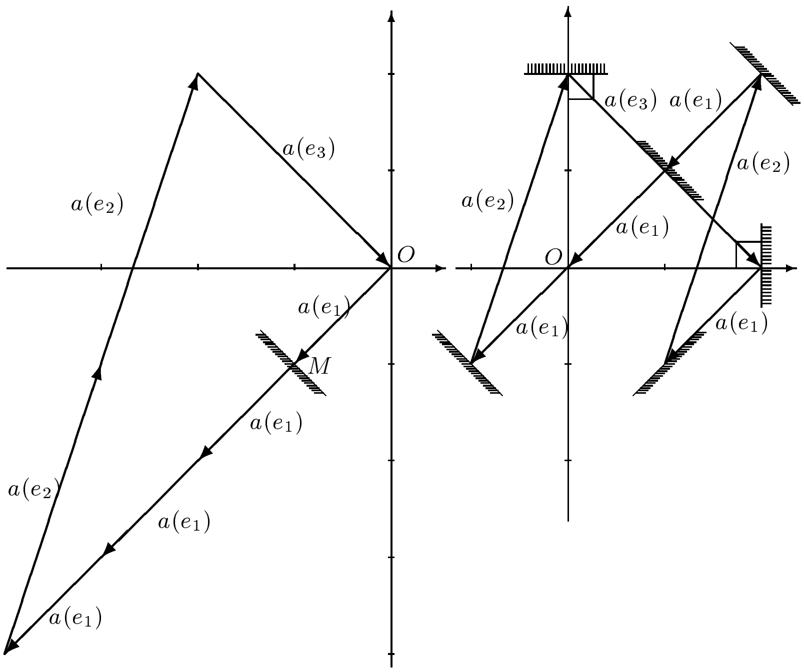
(C_p) increase by 1 some x_j satisfying $a(x) \cdot a(e_j) < 0$.

In case of a single equation ($p = 1$), the scalar product $\cdot$ is reduced to the standard product $\times$ on integers, so that the condition (C_1) is actually a particular case of (C_p).

Such a restriction reduces the search space without losing any minimal solution, since every sequence of vectors which corresponds to a solution can be rearranged in a sequence satisfying the restriction as examplified on figure 3.

Let us now give a first (naïve) version of our algorithm for computing the basis $\mathcal{B}$ of a linear Diophantine system defined by its matrix a.

Let $S = \begin{cases} - x_1 + x_2 + 2x_3 - 3x_4 = 0 \\ - x_1 + 3x_2 - 2x_3 - x_4 = 0 \end{cases}$



The sequence of default vectors displayed on the left, that is $a(e_1) + a(e_1) + a(e_1) + a(e_1) + a(e_2) + a(e_2) + a(e_3)$, corresponds to the solution $(4, 2, 1, 0)$, but is obtained by a sequence which does not satisfy the condition (C_2). This sequence can be rearranged so as to fulfill the restriction, yielding the new sequence $a(e_1) + a(e_2) + a(e_3) + a(e_1) + a(e_2) + a(e_1) + a(e_1)$ displayed on the right part of the figure.

Figure 3: Completeness.

input: $a(x) = 0$, a linear system of linear Diophantine equations.

$$
\begin{array}{l}
\mathcal{P} := \{e_1, \ldots, e_q\} \\
\mathcal{B} := \emptyset \\
\textbf{while } \mathcal{P} \neq \emptyset \textbf{ do} \\
\quad \mathcal{B} := \mathcal{B} \cup \{x \in \mathcal{P} \mid a(x) = 0\} \\
\quad \mathcal{Q} := \{x \in \mathcal{P} \setminus \mathcal{B} \mid \forall s \in \mathcal{B} \;\; x \not\gg s\} \\
\quad \mathcal{P} := \{x + e_i \mid x \in \mathcal{Q}, \;\; a(x) \cdot a(e_i) < 0\} \\
\textbf{end}
\end{array}
$$

output: $\mathcal{B}$ the basis of the set of solutions of $a(x) = 0$.

Let n be the n-th execution of the while loop. The algorithm can be seen as constructing a directed acyclic graph such that $\mathcal{P}_n$ is the set of nodes at depth $n-1$, $\mathcal{B}_n$ is the subset of $\mathcal{P}_n$ whose elements are minimal solutions, and $\mathcal{Q}_n$ is the set of internal nodes at depth $n-1$:

$$
\begin{array}{rlcl}
 & \mathcal{P}_1 & = & \{e_j \mid j \in [1..q]\} \\
\forall n \geq 1, & \mathcal{P}_{n+1} & = & \{x + e_j \mid x \in \mathcal{Q}_n, a(x) \cdot a(e_j) < 0\} \\
 & \mathcal{B}_n & = & \{x \in \mathcal{P}_n \mid a(x) = 0\} \\
 & \mathcal{Q}_n & = & \{x \in \mathcal{P}_n \backslash \mathcal{B}_n \mid \forall s \in \bigcup_{k<n} \mathcal{B}_k, x \not\gg s\}
\end{array}
$$

The algorithm stops as soon as $\mathcal{P}_n$ is empty and returns the set $\mathcal{B} = \bigcup_{k<n} \mathcal{B}_k$.

Example 1
Let us consider the following example (see figure 4):

$$
S \left\{ \begin{array}{rrrrrrrcl}
- & x_1 & + & x_2 & + & 2x_3 & - & 3x_4 & = & 0 \\
- & x_1 & + & 3x_2 & - & 2x_3 & - & x_4 & = & 0
\end{array} \right.
$$

According to the description of the algorithm, we obtain the dag drawn on figure 4, and for this example the basis $\mathcal{B}$ is therefore equal to

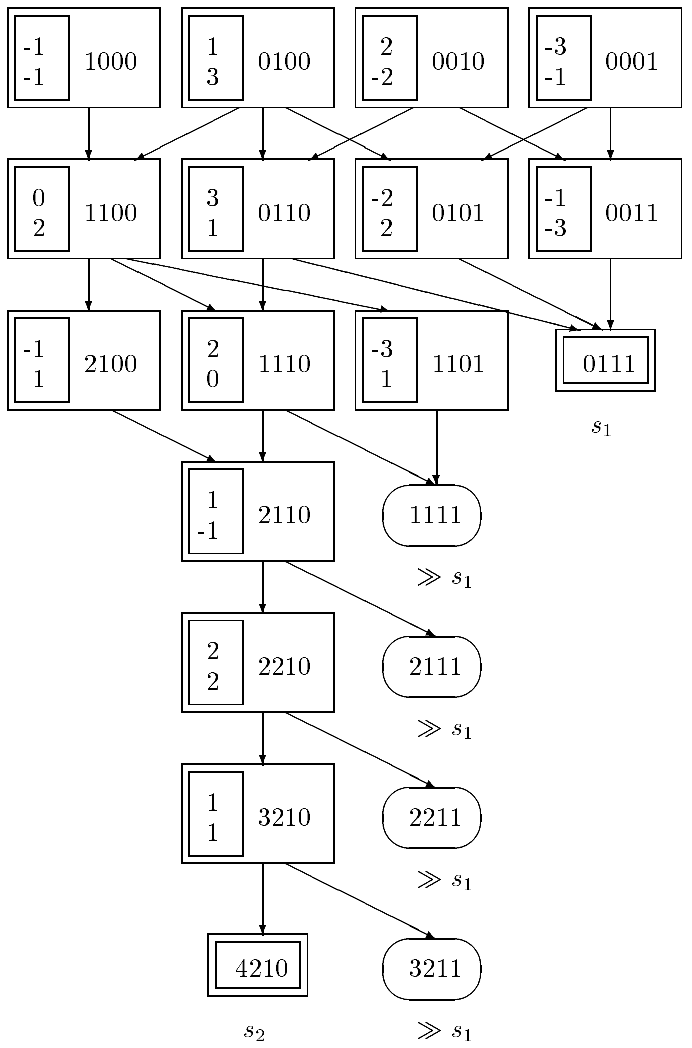
$$\mathcal{B} = \{(0,1,1,1); (4,2,1,0)\}$$

5 Proof of the algorithm

On the contrary to Fortenbacher's algorithm, the proof is quite delicate, and involves techniques from real analysis.

Proposition 1 (Completeness)
Every minimal solution of S is computed, i.e. $\mathcal{B} \subseteq \bigcup_{n \geq 1} \mathcal{B}_n$.

Proof. - The key to the proof is the following remark: from the extremity M of $a(x)$, where x is an arbitrary tuple in N^q, all sequences of



There are only two kinds of leaves in this example: s_1 and s_2 are solutions, the others are greater than a solution. The default vector of each quadruple is written at its left.

Figure 4: The dag developed by the algorithm for the example 1

default vectors returning to the origin do contain a vector heading to the "right" direction. Since addition of vectors is associative and commutative, such a vector can always be applied first, as done by our algorithm.

Formally, let s be such a solution. The algorithm constructs a sequence $v_1, \ldots, v_n$ such that

$$\begin{cases} \forall i \in [1..n-1], & v_i \in \mathcal{Q}_i \text{ and } s \gg v_i \ ; \\ & v_{i+1} = v_i + e_{j_i}, (j_i \in [1..q]) \ ; \\ v_n = s \ . & \end{cases}$$

For v_1, every e_j such that $s \gg e_j$ can be chosen. Such tuples e_j do exist since s is not null, and belong to $\mathcal{P}_1$, hence to $\mathcal{Q}_1$ since s is minimal.
Suppose that we have got $v_1, \ldots, v_i$. $s \gg v_i$, hence there exists s_i such that $s = v_i + s_i$.

$$0 = \|a(s)\|^2 = \underbrace{\|a(v_i)\|^2}_{>0} + \underbrace{\|a(s_i)\|^2}_{>0} + 2\, a(v_i) \cdot a(s_i)$$

hence $a(v_i) \cdot a(s_i) < 0$. Consequently, there exists e_{j_i} $(s \gg e_{j_i})$ such that $a(v_i) \cdot a(e_{j_i}) < 0$. Let us take $v_{i+1} = v_i + e_{j_i}$. If v_{i+1} is equal to s, we are done. Otherwise, v_{i+1} is neither a solution, nor greater than a solution (since it is smaller than a minimal solution) so it belongs to $\mathcal{Q}_{i+1}$. That $s \gg v_{i+1}$ is straightforward.
Finally, we have proved that s belongs to $\mathcal{B}_n$. $\square$

Proposition 2 (Soundness)
Every computed solution is minimal, i.e. $\bigcup_{n\geq 1} \mathcal{B}_n \subseteq \mathcal{B}$.

Proof. - Let s be in some $\mathcal{B}_n$. It is obvious that s is a solution; let us suppose that s is not minimal. Then $s = s_1 + s_2$ where s_1 and s_2 are (non-null) solutions smaller than s. The tuple x immediately above s in the dag obviously belongs to $\mathcal{Q}_{n-1}$; Clearly, it must be greater than (or equal to) s_1 or s_2, hence greater than or equal to an already computed minimal solution (because of completeness). Both cases yield a contradiction. $\square$

The ability to compute directly minimal solutions is a key advantage of this algorithm over the "classical method" based on Gaussian elimination which requires an expensive process for eliminating non-minimal solutions.

Termination is the hard part of the proof. It is based upon the following lemma which follows from the use of the condition (C_p) :

Lemma 1
Let $(v_n)_{n\in N}$ *be an infinite sequence satisfying the constraint* (C_p). *Then*

$$\lim_{n\to\infty} \frac{\|a(v_n)\|}{n} = 0.$$

Proof. - We prove by induction on n that $\|a(v_n)\| \leq C\sqrt{n}$, where $C = max_{1\leq j\leq q}\|a(e_j)\|$. It is obvious for $n = 1$. Let us assume it is true for $n \geq 1$ and let v_{n+1} be in $\mathcal{P}_{n+1}$. v_{n+1} can be written as $v_n + e_j$ with $v_n \in \mathcal{P}_n$ and $j \in [1..q]$ such that $a(v_n) \cdot a(e_j) < 0$. We get:

$$\|a(v_{n+1})\|^2 = \underbrace{\|a(v_n)\|^2}_{\leq n\,C^2} + \underbrace{\|a(e_j)\|^2}_{\leq C^2} + \underbrace{2\,a(v_n) \cdot a(e_j)}_{<0} \leq (n+1)\,C^2$$

Hence

$$\forall n \geq 1,\ 0 \leq \|a(\frac{v_n}{n})\| = \frac{1}{n}\,\|a(v_n)\| \leq \frac{1}{n}\,C\sqrt{n} = \frac{C}{\sqrt{n}}$$

□

It is worth noting that any restriction implying the above lemma yields termination, since the forthcoming proof does only rely on the lemma.

Proposition 3 (Termination)
There is no infinite sequence $(v_n)_{n\in N}$ satisfying lemma 1 and such that for all n, v_n is not greater than a solution of $a(x) = 0$, i.e. $\exists n \geq 1, \mathcal{P}_n = \emptyset$.

Proof. - Assume there exists an infinite sequence $(v_n)_{n\geq 1}$ satisfying lemma 1. We will prove that there exists a solution smaller than v_n for some n. As usually, for all $n \geq 1$, $v_{n+1} = v_n + e_j$ for some j in $[1..q]$. The tuple v_n may be written $(v_{1n}, \ldots, v_{qn})$; for all j in $[1..q]$, v_{jn} is a natural number, and $\sum_{j=1}^{q} v_{jn} = n$.

We first prove the existence of a solution in strictly positive real numbers. For this purpose, let us consider the sequence $(u_n)_{n\geq 1}$ defined by :

$$\forall n \geq 1,\ u_n = \frac{v_n}{n}$$

Since the sequence (v_n) satisfies lemma 1,

$$\lim_{n\to\infty} \|a(u_n)\| = 0\ .$$

On the other hand, for all $n \geq 1$, u_n takes its value in $[0,1]^q$ (where $[0,1]$ is the real interval). Since $[0,1]^q$ is a compact subset of R^q, the sequence $(u_n)_{n\geq 1}$ has an adherence value $l = (l_1, \ldots, l_q)$, limit of a subsequence $(u_{\varphi(n)})_{n\geq 1}$ (where φ is an increasing mapping on N).
Using now the continuity of the mapping $v \mapsto \|a(v)\|$,

$$\|a(l)\| = \lim_{n\to\infty} \|a(u_{\varphi(n)})\| = 0\ .$$

Hence $a(l) = 0$, and l is a tuple in $[0,1]^q$ solution of S in R^q. Note that $\sum_{j=1}^{q} l_j = 1$, hence l is not the null tuple. This ends up the first step of the proof.

Starting now from the solution l, our goal is to construct a solution in $\mathbb{N}^q$ smaller than v_n for all n greater than some n_0. Using continuity arguments, we will first construct a solution l' in $\mathbb{Q}^q$. Using standard techniques in arithmetic, l' will then be transformed into a solution $l'' \in \mathbb{N}^q$. Let us now proceed with the first of these two steps. We have established that every component l_j in l is positive and that at least one of them is non negative. Let k be the number of non zero components in l. Then $1 \leq k \leq q$ and by reindexing the components $(l_j)_{1 \leq j \leq q}$ of l if necessary, we can suppose that these k components are $l_1, \ldots, l_k$. Let us furthermore notice that, as l_j (where $1 \leq j \leq k$) is non null, the sequence $(v_{jn})_{n \geq 1}$ takes infinitely large values.
The reals $l_1, \ldots, l_k$ generate a vector space on $\mathbb{Q}$ whose dimension is m $(1 \leq m \leq k)$. Let $(l_1, \ldots, l_m)$ be a basis of this vector space (after reindexing again the components $(l_j)_{1 \leq j \leq q}$ if necessary). Then the vectors $l_{m+1}, \ldots, l_k$ can be decomposed on this basis :

$$\forall r \in [m+1..k],\ \forall j \in [1..m],\ \exists \alpha_{jr} \in \mathbb{Q},\ l_r = \sum_{j=1}^{m} \alpha_{jr} l_j \ .$$

Therefore :

$$0 = a(l) = \sum_{j=1}^{k} l_j\, a(e_j) = \sum_{j=1}^{m} l_j\, (a(e_j) + \sum_{r=m+1}^{k} \alpha_{jr}\, a(e_r)) \ .$$

The vectorial equation above stands for the p equations :

$$\left\{ \begin{array}{rcl} \sum_{j=1}^{m} l_j\, (a_{1j} + \sum_{r=m+1}^{k} \alpha_{jr}\, a_{1r}) & = & 0 \\ \vdots & & \vdots \\ \sum_{j=1}^{m} l_j\, (a_{ij} + \sum_{r=m+1}^{k} \alpha_{jr}\, a_{ir}) & = & 0 \\ \vdots & & \vdots \\ \sum_{j=1}^{m} l_j\, (a_{pj} + \sum_{r=m+1}^{k} \alpha_{jr}\, a_{pr}) & = & 0\ . \end{array} \right.$$

Note that for all (i,j) in $[1..p] \times [1..q]$, $a_{ij} + \sum_{r=m+1}^{k} \alpha_{jr}\, a_{ir}$ is a rational number. Since the reals $l_1, \ldots, l_m$ are lineary independent on $\mathbb{Q}$, these $p \times q$ rational coefficients are null. Hence:

$$\begin{array}{rcll} \forall (z_1, \ldots, z_m) \in \mathbb{R}^m,\quad 0 & = & \displaystyle\sum_{j=1}^{m} z_j\, (a(e_j) + \sum_{r=m+1}^{k} \alpha_{jr}\, a(e_r)) & \\ & = & \displaystyle\sum_{j=1}^{m} z_j\, a(e_j) + \sum_{r=m+1}^{k} (\sum_{j=1}^{m} \alpha_{jr} z_j)\, a(e_r) & (*) \end{array}$$

Now, for all r in $[m+1..k]$, $\sum_{j=1}^{m} \alpha_{jr} l_j = l_r > 0$. By a continuity argument, there exist rational and non negative numbers, w_j , $1 \leq j \leq m$, which are close enough to the respective l_j so that :

$$\forall r \in [m+1..k],\ \sum_{j=1}^{m} \alpha_{jr} w_j > 0\ .$$

As a particular case of (*), we have :

$$0 = \sum_{j=1}^{m} w_j\, a(e_j) + \sum_{r=m+1}^{k} (\sum_{j=1}^{m} \alpha_{jr} w_j)\, a(e_r)\ .$$

Consequently, the following tuple :

$$l' = (\underbrace{w_1, \ldots, w_m}_{m}, \underbrace{\sum_{j=1}^{m} \alpha_{j\ m+1} w_j, \ldots, \sum_{j=1}^{m} \alpha_{jk} w_j}_{k-m}, \underbrace{0, \ldots, 0}_{q-k})$$

is a solution of S whose components are in $\mathbb{Q}$, are positive, and are not all null.
We can now proceed with the last step: multiplying the components of l' by the least common multiplicator of the denominators yields a solution l'' in $\mathbb{N}^q$.

As the first k components of v_n are increasing with n without bound, the following property is satisfied :

$$\exists n_0 > 1, \forall n > n_0, \forall j \in [1..k], v_{jn} > l''_j.$$

Hence :

$$\forall n > n_0, v_n \gg l''.$$

Therefore, v_{n_0+1} cannot have been generated by the algorithm, a contradiction. Hence the algorithm terminates. □

Remarks. -

- The previous proof is highly depending on Lemma 1. Whichever algorithm satisfying the property that $a(v_n)$ is in the limit infinitely smaller than n will terminate. We have not explored this direction any further, but Pottier has found another condition ensuring this property [12].

- Another characteristic of this proof is that it does not provide bounds for the solutions, since it works by contradiction. Here is the main difference with Fortenbacher's algorithm for which there exists a very simple bound on the length of the computation. Namely, for an equation $a_1x_1 + \ldots + a_qx_q = 0$, the depth of the dag is bounded by $max_{a_j \geq 0}(a_j) + max_{a_j < 0}(-a_j)$. A bound has been found too in the case of two equations by Romeuf [13], and recently Pottier has presented a new exponential bound for the general case [12].

From the previous propositions, we can derive the following main result :

Theorem 1
The algorithm computes the basis $\mathcal{B}$ of S in finite time, i.e.

$$\exists n_0, \mathcal{P}_{n_0} = \emptyset \quad \wedge \quad \mathcal{B} = \bigcup_{1 \leq n < n_0} \mathcal{B}_n \ .$$

6 Improvements

6.1 From dags to forests

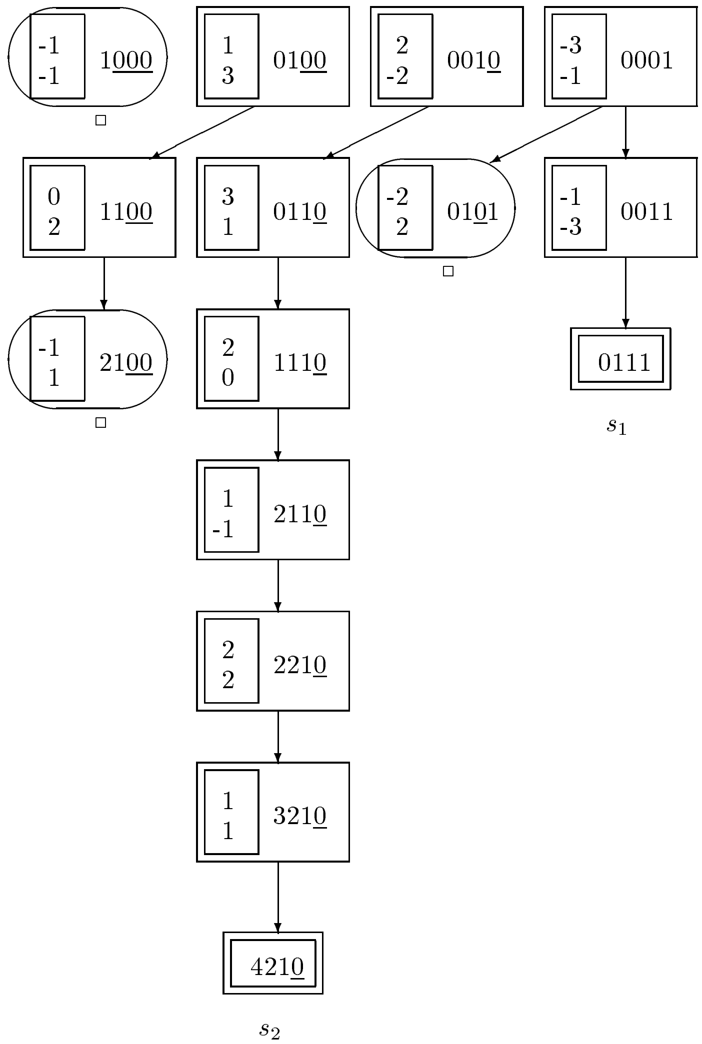
The algorithm we have given deals with sets $\mathcal{P}_n$ of tuples x such that $\sum_{j=1}^{q} x_j = n$. But such a tuple x in $\mathcal{P}_n$ may appear as successor of several different tuples in $\mathcal{P}_{n-1}$, hence the underlying structure is actually a dag, and not a forest.
In order to avoid such redundancy, we need an ordering $\succ$ on the successors of a node in order to avoid having different paths leading to a same node. What is required for $\succ$ is to provide a total ordering $\succ_x$ on the successors of each node x (i.e. a locally total ordering on a subset of the variables in S). Suppose now that a node x has two successors $x + e_{j_1}$ and $x + e_{j_2}$ such that $x_{j_1} \succ x_{j_2}$: then we may forbid in the subdag originating from the node labeled by $x + e_{j_2}$ to add the vector e_{j_1}; the j_1-th component of the tuple becomes *frozen* in this subdag. This will of course reduce the search, and some nodes of the previous algorithm may become leaves.

Several orderings may be chosen. The simplest one is obtained by arbitrarily ordering the q unknowns, for example : $x_1 \succ x_2 \succ \ldots \succ x_q$. This particular ordering is denoted by $\succ_q$. Such an ordering is not node-dependent and was already used by Fortenbacher in order to improve his own algorithm.
Another interesting one, denoted by $\succ_{\| \ \|}$ is based on the idea of a minimal norm :

$$\begin{aligned} x_{j_1} \succ_{\| \ \|} x_{j_2} \text{ iff } & \|a(x + e_{j_1})\| > \|a(x + e_{j_2})\| \\ & \text{or } \|a(x + e_{j_1})\| = \|a(x + e_{j_2})\| \text{ and } j_1 > j_2 \ . \end{aligned}$$

This ordering may provide shorter forests than the previous one : an intuitive reason is that the computed paths stay as close as possible to the origin. Moreover, it is almost independent of the ordering on the variables.

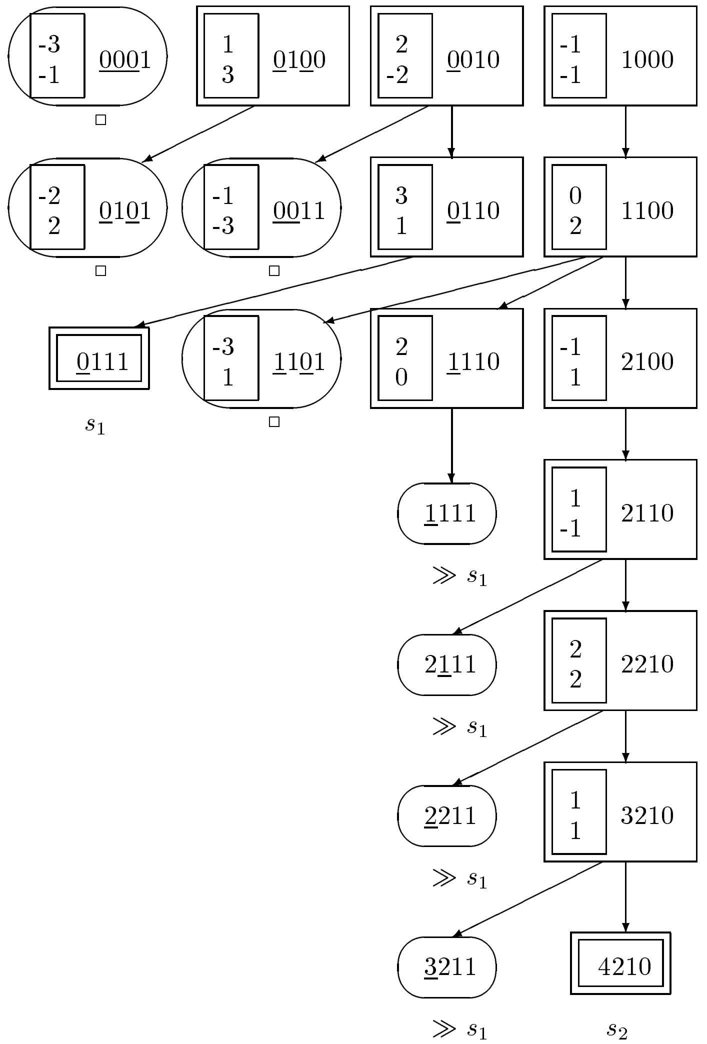
Example 2
We consider the same example as in Example 1. According to the previous description, the forest associated with the ordering $\succ_4$ is represented on figure 5, whereas the forest associated with the ordering $\succ_{\| \ \|}$ is represented on figure 6.



number of nodes	:	15
number of arrows	:	11
number of minimality tests	:	0

The nodes labeled by $(1,1,0,1)$, $(1,1,1,2)$ and $(1,1,2,2)$, which appeared in the dag of figure 4, do not exist any longer in the forest above. The underlined components are those which are frozen by the $\succ_4$ ordering. The symbol $\square$ appears below a node without successors.

Figure 5: The dag developed by the ordered version of the algorithm with the $\succ_4$ ordering on the system S of the example 1



number of nodes	:	20
number of arrows	:	16
number of minimality tests	:	4

The underlined components are those who are frozen by the $\succ_{\|\ \|}$ ordering.

Figure 6: The dag developed by the ordered version of the algorithm with the $\succ_{\|\ \|}$ ordering on the system S of example 1

The ordered version of the algorithm is of course terminating and sound, since it does less computation than the first version. We now prove that it is still complete:

Proposition 4 (Completeness A) *The ordered version of the algorithm is complete.*

Proof. - Let s be a minimal solution of S. We have shown in Proposition 1 that there exists at least a finite sequence $e = (e_{j_1}, \ldots, e_{j_n})$ leading to s with the first algorithm, i.e. such that $\sum_{k=1}^{l} e_{j_k} = v_l \in \mathcal{Q}_l$ and $\sum_{k=1}^{n} e_{j_n} = s$. Let us now consider the set of these sequences and order it in the following way. Let e and e' two such sequences and l the greatest index (eventually 0) satisfying: $\forall k \leq l \; e_{j_k} = e'_{j_k}$; $e \rhd e'$ iff $x_{j_{l+1}} \succ_{v_{j_l}} x'_{j_{l+1}}$. It clearly appears that the smaller sequence w.r.t. $\succ$ is the one computed by the new version of the algorithm. □

6.2 From a forest to a stack

The forests computed by the new version of the algorithm owns an interesting topological property, if all successors of a node are written from left to right according to the ordering $\succ$.

Proposition 5 *If a tuple x in the forest is greater than a solution s, then the leaf labeled by s is on the right of the leaf labeled by x.*

Proof. - Suppose that there is some solution s in the forest on the left side of x. It will be shown that such a solution cannot be smaller than x w.r.t. to the ordering $\gg$. Let x' be the last common ancestor of x and s (eventually $(0, \ldots .0)$).

$$x = x' + w_1$$
$$s = x' + w_2$$

With the previous convention, since s is on the left side of x, w_2 contains some e_{j_0} forbidden in w_1. Hence the j_0 component of s is greater than that of x. Hence, s cannot be smaller than x w.r.t. to $\gg$. □

As a consequence, it is enough to check whether a tuple is greater than the solutions on its right side.
And indeed, there is a way for enumerating solutions from right to left, by using a stack. Such a stack searches the forest depth first, but from right to left.

The initial stack is filled in with $e_1, \ldots, e_q$, the roots of the dag (e_1 at the bottom, e_q at the top). The top is poped and its successors are pushed onto the stack with respect to the ordering $\succ$, the greatest one being pushed first. The forest has been completely searched once the stack is empty.

Note moreover that the size of the stack is bounded by q. This results immediately from the following straightforward property : a tuple at level

j ($1 \leq j \leq q$) in the stack has exactly $j-1$ frozen components. As a consequence we obtain an extremely space-efficient algorithm. This stack-version, with the simple ordering $\succ_q$, is described below:

input: $a(x) = 0$, a system of linear homogeneous Diophantine equations.

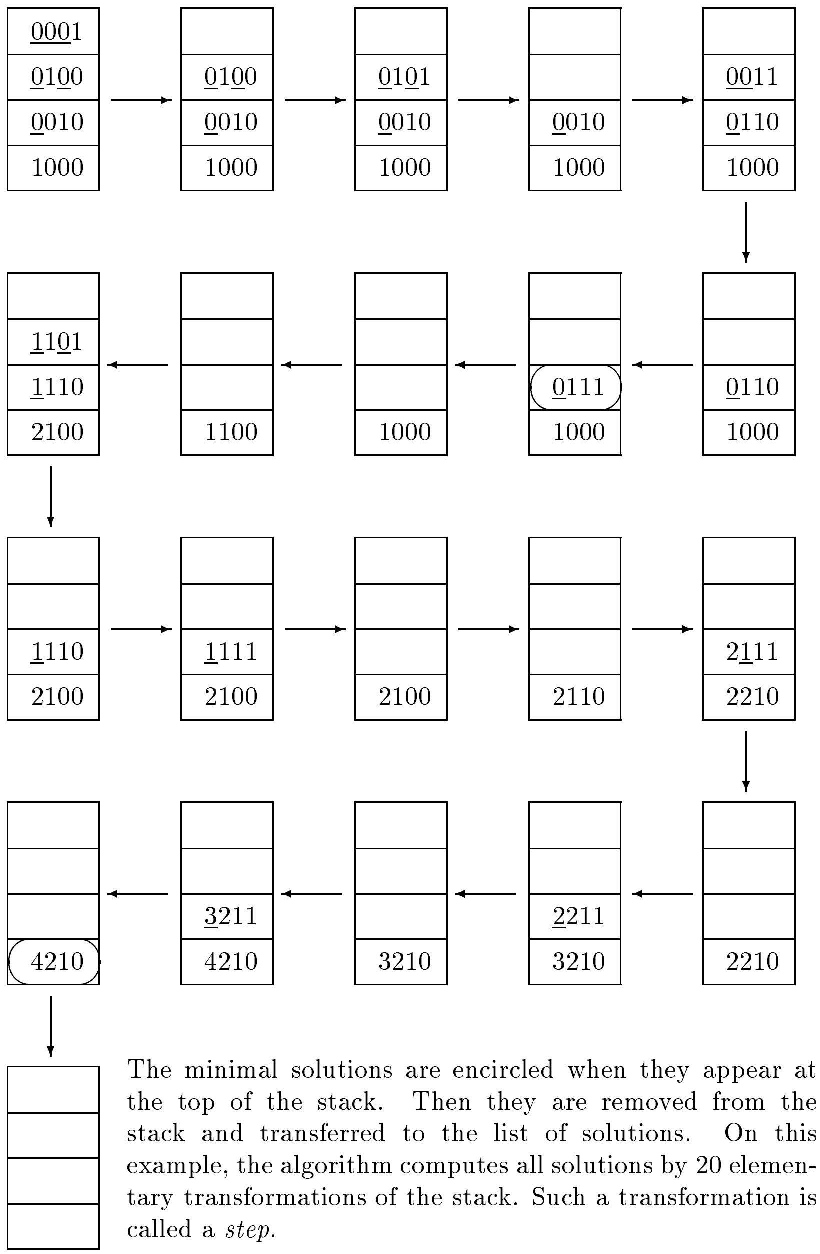
```
P := Push(0⃗, emptystack)
B := emptylist
for i := 1 to q do
Frozen[1,i] := False
endfor
while P ≠ emptystack do
t := top(P)
P := pop(P)
n := height(P) − 1
if a(t) = 0⃗ then B := cons(t, B)
    else
       for i := 1 to q do
       F[i] := Frozen[n + 1, i]
       endfor
       for i := 1 to q do
           if F[i] = False and a(t) · a(e_i) < 0 and Min(t + e_i, B)
            then P := Push(t + e_i, P)
                 n := n + 1
                 for j := 1 to q do
                 Frozen[n, j] := F[j]
                 endfor
                 F[i] := True
           endif
       endfor
   endif
endwhile

Min(B, t)
if B = emptylist then Min(B, t) := True
else Min(B, t) := not (car(B) ≫ t) and Min(cdr(B), t)
```

output: $\mathcal{B}$ the basis of the set of solutions of $a(x) = 0$.

Example 3

We compute the minimal solutions for the system $\mathcal{S}$ of the first example again, but with the stack-version of the algorithm (see the figure 7).



The minimal solutions are encircled when they appear at the top of the stack. Then they are removed from the stack and transferred to the list of solutions. On this example, the algorithm computes all solutions by 20 elementary transformations of the stack. Such a transformation is called a *step*.

Figure 7: The sequence of values of the stack for the system S of example 1 with the ordering $\succ_{\|\ \|}$.

7 Non Homogeneous Diophantine Systems

Our method can be easily modified to cope with non homogeneous Diophantine systems. Let us now consider the following system S'

$$\begin{cases} a_{11}x_1 & +\ldots+ & a_{1j}x_j & +\ldots+ & a_{1q}x_q & = & b_1 \\ \vdots & & \vdots & & \vdots & & \vdots \\ a_{i1}x_1 & +\ldots+ & a_{ij}x_j & +\ldots+ & a_{iq}x_q & = & b_i \\ \vdots & & \vdots & & \vdots & & \vdots \\ a_{p1}x_1 & +\ldots+ & a_{pj}x_j & +\ldots+ & a_{pq}x_q & = & b_p, \end{cases}$$

where $b = (b_1, \ldots, b_p)$ is a vector of integers. The set $\mathcal{S}$ of solutions of S' is no longer an additive monoid. It can be characterized by two sets (instead of one as before) $\mathcal{M}_0$ and $\mathcal{M}_1$. $\mathcal{M}_0$ is the set of minimal solutions of S, the homogeneous system associated with S', and $\mathcal{M}_1$ is the set of minimal solutions of S'. Now every solution is the sum of an element of $\mathcal{M}_1$ and an arbitrary number of elements of $\mathcal{M}_0$, which we write $\mathcal{S} = \mathcal{M}_1 + \mathcal{M}_0^*$.
It is very easy to turn S' into an homogeneous system S'_0 by introducing an extra variable x_0:

$$S'_0 \begin{cases} - & b_1x_0 & + & a_{11}x_1 & + & \ldots & + & a_{1q}x_q & = & 0 \\ & \vdots & & \vdots & & & & \vdots & & \vdots \\ - & b_px_0 & + & a_{p1}x_1 & + & \ldots & + & a_{pq}x_q & = & 0 \end{cases}$$

It is clear that the sets $\mathcal{M}_0$ and $\mathcal{M}_1$ are projections of some subsets of the set $\mathcal{M}$ associated with S'_0.
$\mathcal{M}_0 = \{x \in N^q \mid (0, x) \in \mathcal{M}\}$
$\mathcal{M}_1 = \{x \in N^q \mid (1, x) \in \mathcal{M}\}$
So, it is sufficient to bound the component associated with x_0 by 1: in a tuple, when this component reaches the value 1, it behaves as if it was frozen. Moreover, there is a very simple way to split $\mathcal{M}$ into $(0, \mathcal{M}_0)$ and $(1, \mathcal{M}_1)$: let x_0 be the greatest variable with respect to the ordering $\succ$; then the algorithm provides first $(0, \mathcal{M}_0)$ and then $(1, \mathcal{M}_1)$.

8 Constrained Diophantine systems

In this section, we consider more general problems, whose one part is a set of linear Diophantine equations, and other part is an ordering constraint. For example, we may need to compute the set of minimal solutions greater (or smaller) than some $x_0 \in N^q$ or the set of minimal solutions "between" x_1 and x_2.
Since our algorithm considers a system as a whole and not as a list of equations, it can be very easily modified in order to compute such sets, without computing useless solutions.

Constraints $s \gg x_1$

Instead of starting with the tuples $e_1, \ldots, e_q$, we simply start with the tuple x_1. The rest of the algorithm remains the same.

It should however be noticed that $\mathcal{M}in\{s \mid As = 0 \ , \ s \gg x_1\}$ does not give us all solutions of $\{s \mid As = 0 \ , \ s \gg x_1\}$. Such a solution may be written as a sum of one tuple in $\mathcal{M}in\{s \mid As = 0 \ , \ s \gg x_1\}$ and an arbitrary number of tuples in $\mathcal{M}in\{s \mid As = 0\}$:

$$\{s \mid As = 0 \ , \ s \gg x_1\} = \mathcal{M}in\{s \mid As = 0 \ , \ s \gg x_1\} + \mathcal{M}in\{s \mid As = 0\}^*$$

Constraints $x_2 \gg s$

We compute $\mathcal{M}in\{s \mid As = 0 \ x_2 \gg s\}$ as in the standard algorithm from the q-tuples $e_1, \ldots, e_q$, but the j-th component of a tuple becomes frozen as soon as it reaches the j-th component of x_2.

If we want to compute $\{s \mid As = 0 \ x_2 \gg s\}$ instead of $\mathcal{M}in\{s \mid As = 0 \ x_2 \gg s\}$, we can remove the minimality test.

Constraints $x_2 \gg s \gg x_1$

The two previous techniques combined give the solution.

Theorem 2
The previous algorithms are sound ,complete and terminating.

The proof follows directly from Proposition 2 and the previous remarks.

Remark. - These algorithms compute the set of minimal solutions provided not all components of the tuples are bounded.

9 Incrementality

In constraint logic programming, the set of constraints is usually increased at each logical inference. Hence, a crucial point is to build incremental algorithms for solving linear systems of Diophantine equations. Our algorithm can be easily adapted to this purpose. Let $S_1 \wedge S_2 \wedge \ldots \wedge S_n$ be a linear system composed of n subsystems. Assume $\mathcal{M}^i$ is the set of minimal solutions of $S_1 \wedge S_2 \wedge \ldots \wedge S_i$. The set $\mathcal{M}^{i+1}$ of minimal solutions of $S_1 \wedge S_2 \wedge \ldots \wedge S_i \wedge S_{i+1}$ can of course be obtained by Gaussian elimination, as explained in the introduction. It would however be more expensive than computing directly $\mathcal{M}^{i+1}$ with our algorithm!

The key point of the incremental version of our algorithm is very simple: it consists in using the elements $m_1, \ldots, m_k$ of $\mathcal{M}^i$ (instead of the canonical tuples) as roots of the forest as well as increments for solving the subsystem S_{i+1} in the forest version of the algorithm. This means that we use the vectors $a(m_i)$ as new default vectors in place of the vectors $a(e_j)$.

The forest that we get is contained in the forest computed by the simple version of the algorithm from the subsystem

$$S_{i+1}(m_1)y_1 + \ldots + S_{i+1}(m_k)y_k = 0$$

where every node is replaced by its image under the linear mapping

$$(y_1, \ldots, y_n) \mapsto y_1 m_1 + \ldots + y_k m_k$$

This mapping is not, in general, compatible with the ordering $\gg$, and the proposition 5 is no longer true.

Hence two different choices can be made: either the forest version, with a minimality check involving all the solutions above a tuple, or the stack version, with a minimality check involving all the solutions at the right of a tuple. Since the proposition 5 is false in the incremental case, the incremental stack version computes always at most as many nodes as the incremental forest version (see the example below).

Example 4 Let us consider the system S of the first example, that we can split into the two subsystems S_1 and S_2:

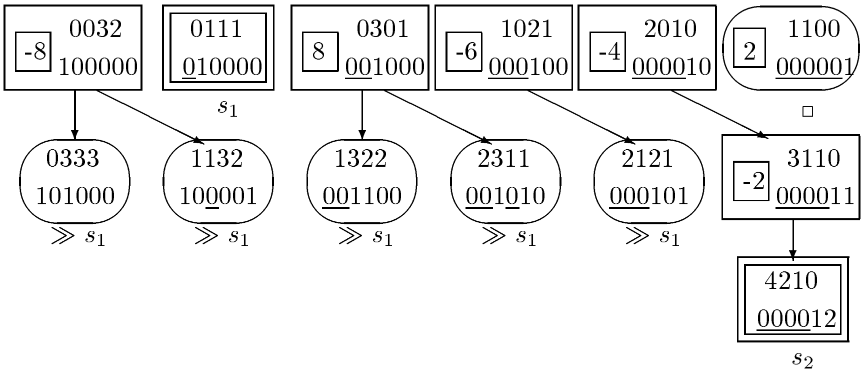
$$\begin{array}{l} S_1 \equiv -x_1 + x_2 + 2x_3 - 3x_4 = 0 \\ S_2 \equiv -x_1 + 3x_2 - 2x_3 - x_4 = 0. \end{array}$$

S_1 is first solved by the standard version of the algorithm which yields

$$\mathcal{M}^1 = \left\{ \begin{array}{c} (0,0,3,2) \\ (0,1,1,1) \\ (0,3,0,1) \\ (1,0,2,1) \\ (2,0,1,0) \\ (1,1,0,0) \end{array} \right\}$$

and then $S_1 \wedge S_2$ is solved by the incremental version, as shown on the figure 8.

The incremental version of the algorithm for solving a system S made of n subsystems $a_1(x) = 0, \ldots, a_n(x) = 0$ of linear Diophantine equations is therefore as follows :



Under each 4-tuple u, there is a 6-tuple $(c_1, \ldots, c_6)$ which corresponds to the coefficients of the m_i in u, *i.e.* such that $\Sigma_{i=1}^{6} c_i m_i = u$. $(c_1, \ldots, c_6)$ is also a node of the forest developed by the algorithm in order to solve the equation $S_2(m_1)y_1 + \ldots + S_2(m_6)y_6 = 0$, that is to say

$$-8y_1 + 0y_2 + 8y_3 - 6y_4 - 4y_5 + 2y_6 = 0$$

as it can be seen below.

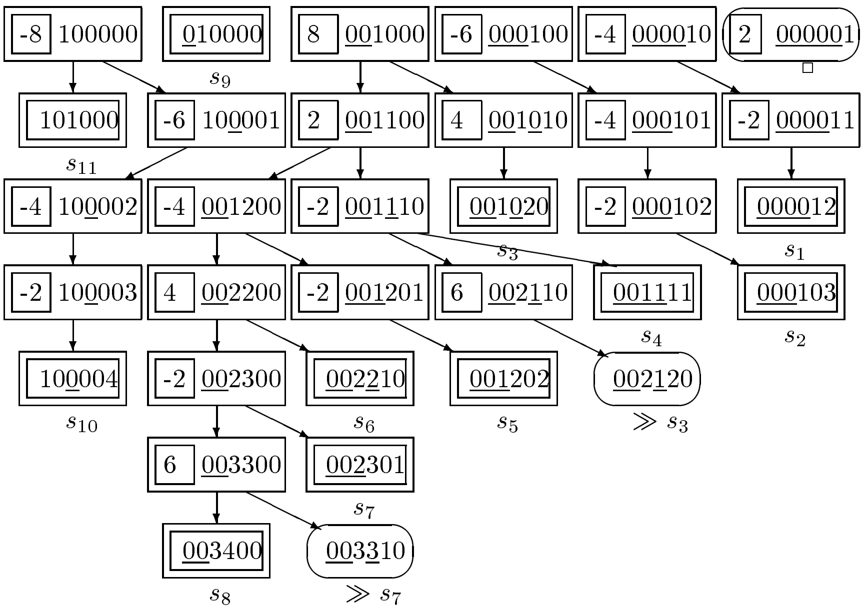


Figure 8: The forest developed by the incremental algorithm for solving S

$$
\begin{array}{l}
\mathcal{P}_1^1 = \{e_j \mid j \in [1..q]\} \ ; \\
\forall k \geq 1,\ \mathcal{P}_{k+1}^1 = \{x + e_j \mid x \in \mathcal{Q}_k^1, a_1(x) \cdot a_1(e_j) < 0\} \ ; \\
\forall k \geq 1,\ \mathcal{B}_k^1 = \{x \in \mathcal{P}_k^1 \mid a_1(x) = 0\} \ ; \\
\forall k \geq 1,\ \mathcal{Q}_k^1 = \{x \in \mathcal{P}_k^1 \backslash \mathcal{B}_k^1 \mid \forall s \in \bigcup_{l<k} \mathcal{B}_l^1, x \not\gg s\} \ ; \\
\mathcal{M}^1 = \bigcup_k \mathcal{B}_k^1 \ ; \\
\forall\ 2 \leq j \leq n,\ \ \mathcal{P}_1^j = \mathcal{M}^{j-1} \ ; \\
\forall\ 2 \leq j \leq n,\ \forall k \geq 1,\ \ \mathcal{B}_k^j = \{x \in \mathcal{P}_k^j \mid a_j(x) = 0\} \ ; \\
\forall\ 2 \leq j \leq n,\ \forall k \geq 1,\ \ \mathcal{Q}_k^j = \{x \in \mathcal{P}_k^j \backslash \mathcal{B}_k^j \mid \forall s \in \bigcup_{l<k} \mathcal{B}_l^j, x \not\gg s\} \ ; \\
\forall\ 2 \leq j \leq n,\ \mathcal{M}^j = \bigcup_k \mathcal{B}_k^j \ .
\end{array}
$$

10 Satisfiability: an incremental test

For some applications (AC-unification), all the minimal solutions for a linear Diophantine system are required, but in constraint logic programming, one may only need a single solution. As explained above, all the constraints are not known when starting the resolution of a problem. Hence a important point is to check for satisfiability incrementally. This is possible with the stack version of our algorithm:

Theorem 3 *Let S be a linear Diophantine system $S_1 \wedge S_2$. Let $\mathcal{S}_t$ be the stack computed by the stack version of the algorithm with the input S_1 when the first solution is encountered. Then S has a solution if and only if the stack version of the algorithm with the input S_2, and starting from the stack $\mathcal{S}_t$, finds a solution.*

Remark. - The proof amounts to consider a particular combination of minimal solutions where the first solution only has a non-null coefficient. The details are left to the reader.

It should be noticed that the above process can be iterated.

11 Comparison with other algorithms

Our algorithm was the first algorithm for solving a system of linear Diophantine equations without using Gaussian elimination. Other algorithms have been later described by Pottier and by Domenjoud. We will give a short description of these algorithms, and make some comparisons.

11.1 Pottier's first algorithm

Pottier has suggested the use of an alternative to our condition (C_p) where the current tuple x is forced to remain as close as possible to the straight line defined by a solution [12]. More precisely $a(x)$ has to belong to an hypercube depending on the system only. Formally Pottier's condition is as follows:

(NC_p) increase by 1 some x_j satisfying
for all l, the l^{th} component of $a(x_1, \ldots, x_{j-1}, x_j + 1, x_{j+1}, \ldots, x_q)$ belongs to the interval $[-\sum_{l=1}^{q} max\{a_{il}, 0\}, \sum_{l=1}^{q} max\{-a_{il}, 0\}]$.

As shown in [12], this condition yields a bound over the sum of the components of a minimal solution:

Theorem 4 (Pottier) *Let $a(x) = 0$ be a system of linear Diophantine equations, and $m = (m_1, \ldots, m_q)$ one of its minimal solutions. Then*

$$\sum_{i=1}^{q} m_i \leq \prod_{i=1}^{p} (\sum_{j=1}^{q} |a_{ij}| + 1)$$

In fact, Pottier gives an even more precise bound depending on the rank of the matrix a.

Pottier's condition and our condition (C_p) are not comparable, since no one implies the other. Hence, there is no systematic inclusion relation between the sets of nodes computed by both algorithms. Note however that our condition is much simpler to express and to compute. In all our practical (hand-made) experiments, our algorithm computed a smaller set of nodes, but both sets were always of the same order of magnitude.

11.2 Pottier's second algorithm

Pottier's second algorithm [12] is derived from the computation of a standard (or Gröbner) basis for a particular ideal associated with any linear Diophantine system. A q-tuple is considered as the exponent of a monomial in $K[Y_1, \ldots, Y_q]$, where K is a field. An algorithm for the computation of a standard basis has a rewrite system as output [2]. All solutions can be obtained as the exponents of some monomials emumerated by using this rewrite system from right to left, and starting from the trivial monomial 1 associated with the trivial solution $\vec{0}$. In order to obtain the minimal solutions, one has to enumerate all the generated tuples below a previously known bound (given for example by the previous method). This ensures termination.

Although elegant and simple, this method leads to intractable computations, hence is outperformed by all the others.

11.3 Domenjoud's algorithm

Domenjoud's algorithm is the juxtaposition of two distinct algorithms [7]. The first one computes all minimal solutions with a minimal support for a given linear Diophantine system. This step is performed by a new method different from Farkas' algorithm used in Petri nets [8]. Then these particular minimal solutions are combined (using rational positive coefficients) so as to provide all minimal solutions. Before applying Domenjoud's algorithm, one has to be sure that the matrix $p \times q$ associated with the linear Diophantine system $a(x) = 0$ has a rank equal to p (no redundant equation), and that $q \leq p + 1$ (at least one non-zero solution in $\mathbb{Z}^q$). There of course always exists such a system which admits the same set of solutions than $a(x) = 0$.

The computation of the set of minimal solutions with a minimal support is based on the following theorem:

Theorem 5 (Domenjoud)
Let s be a minimal solution with a minimal support for the linear Diophantine system
$a(x) = 0$. There exists a subset $\{i_1, \ldots, i_{p+1}\}$ of $\{1, \ldots, q\}$, and a positive integer k such that

$$det \begin{pmatrix} e_{i_1} & \ldots & e_{i_{p+1}} \\ a(e_{i_1}) & \ldots & a(e_{i_{p+1}}) \end{pmatrix} = k\ s$$

Hence a very simple way to obtain all minimal solutions with a minimal support consists in computing the determinant $det \begin{pmatrix} e_{i_1} & \ldots & e_{i_{p+1}} \\ a(e_{i_1}) & \ldots & a(e_{p+1}) \end{pmatrix}$ for every subset $\{i_1, \ldots, i_{p+1}\}$ of $\{1, \ldots, q\}$ and then retaining the non-null vectors for which all components are simultaneously non-negative or non-positive. Such vectors are then divided by the greatest common divisor of their components.

The second step is based on the following result also due to Domenjoud:

Theorem 6 (Domenjoud)
All the minimal solutions of the linear Diophantine system $a(x) = 0$ are linear combinations with positive rational cœfficients of some minimal solutions with minimal support which are linearly independent.

Let α be the vector of positive rational cœfficients of the combination and M be the matrix, the columns of which are the linearly independent minimal solutions with a minimal support of $a(x) = 0$. There exists a matrix U of integers such that $det(U) = \pm 1$, and

$$UM = \begin{pmatrix} T \\ 0 \end{pmatrix}$$

where T is an upper triangular matrix with positive cœfficients on the diagonal. This can be shown by using Gauss' elimination on the matrix M. $M\alpha$ is an integer vector if and only if $T\alpha$ is an integer tuple, that is if and only if $\alpha = T^{-1}X$, where X is an integer vector. As it is sufficient to try some rational cœfficients ranging over the real interval $[0, 1[$ (otherwise, the solutions may be not minimal), α is equal to $T^{-1}X - [T^{-1}X] = \frac{1}{d}[T^{-1}X]_d$, where $d = det(T)$, $[T^{-1}X]$ is the vector built from the integer part of the components of $T^{-1}X$, and $[T^{-1}X]_d$ is the vector of which components are the rests of the Euclidean division of the components of $T^{-1}X$ by d. T and $det(T)$ are known, hence one has to check a finite number of vectors X: vectors the components of which range over the integer interval $\{0, \ldots, d\}$. From this finite set of vectors X, one constructs a finite set of vectors α.

Domenjoud's algorithm and our algorithm are hard to compare formally, since the methods are quite different. On a practical basis, no one is better than the other for all examples. Both algorithms can however be compared with respect to their respective advantages.

Domenjoud's algorithm can be parallelized [7], whereas ours cannot.

On the other hand, our algorithm can be made incremental, whereas Domenjoud's cannot. Moreover, our algorithm allows to control and bound the generated tuples component by component. Hence it is suitable for solving non-homogeneous linear Diophantine systems as well as linear Diophantine equations associated with monotonous linear inequalities. This is an important advantage for logic programming applications.

11.4 Implementation and benchmarks

The implementation of the non incremental stack-version of our algorithm has revealed that checking whether a tuple x is bigger than an already obtained solution may be time-consuming. This leads us to introduce the notion of a positive difference c associated with a tuple x and a solution s:

$$\begin{aligned} c &= (c_1, \ldots, c_q) \\ x &= (x_1, \ldots, x_q) \\ s &= (s_1, \ldots, s_q) \\ where \quad c_j &= max(s_j - xj, 0) \end{aligned}$$

If $x + e_j$ is a successor of x for the algorithm, its positive differences are easy to compute from those of x. When a positive difference of x becomes null, this means that x is greater than the solution it is associated with. The trick is that most of the time, there are less positive differences for a tuple than there are solutions: if two differences are comparable for $\gg$, it is sufficient to keep the smallest one.

We have implemented two different versions of the algorithm, one "with" and the other "without constraints". The one which performs best for the case of a single equation is "with constraints" because of the large number of solutions in the general case, whereas it depends for systems. The following benchmarks for our algorithm were performed on a SUN 4/330 workstation, with a program written in C, whereas Domenjoud's benchmarks come from [7] and were performed on a SUN 4/390 workstation. A bad case for our algorithm is when a system has a few solutions whose size is large. Hence we have tried to run it on some randomly generated systems, and we give the results we got for three such systems. Although the sizes of the minimal solutions are quite large, the algorithm answers in a reasonable time.

System	Gaussian Elimination			Contejean-Devie's Algorithm
	Number of Variables	Number of Solutions	Number of minimal Solutions	
e_1: -4 1 1 1 1 e_2: 1 -4 1 1 1 e_3: 1 1 -4 1 1 e_4: 1 1 1 -4 1	e_1: 5 e'_2: 35 e'_3: 10 e'_4: 3	e_1: 35 e'_2: 215 e'_3: 20 e'_4: 2	e_1: 35 $\{e_1, e_2\}$: 10 $\{e_1, e_2, e_3\}$: 3 $\{e_1, e_2, e_3, e_4\}$: 1	1 solution in 0.017s in 30 steps
e_1: 0 0 0 0 -2 1 1 e_2: 0 0 -4 1 1 1 1 e_3: -6 1 1 1 1 1 1	e_1: 7 e'_2: 7 e'_3: 15	e_1: 7 e'_2: 21 e'_3: 323	e_1: 7 $\{e_1, e_2\}$: 15 $\{e_1, e_2, e_3\}$: 95	95 solutions in 0.650s in 13 193 steps
e_1: 0 0 0 0 -3 1 1 1 e_2: 0 0 -5 1 1 1 1 1 e_3: -7 1 1 1 1 1 1 1	e_1: 8 e'_2: 14 e'_3: 149	e_1: 14 e'_2: 2 015 e'_3: *core*	e_1: 14 $\{e_1, e_2\}$: 149 $\{e_1, e_2, e_3\}$: 11 942	11942 solutions in 127s in 1 971 992 steps

System	Number of Solutions	Domenjoud	Contejean-Devie "with constraints"	
		Time of Computation	Number of Steps	Time of Computation
1 2 -3 -2 -4 2 -1 -3 2 5	10	0.0035s	215	0.017s
10 -7 -8 3 -11 12 -9 -7 3 13	240	0.7850s	65 091	1.967s
0 0 0 0 -2 1 1 0 0 -4 1 1 1 1 -6 1 1 1 1 1 1	95	0.53s	13 193	0.650s
-12 2 2 2 0 3 3 6 -1 -5 0 0 0 0 -6 1 1 1 1 1 1	95	0.58s	262 084	11.000s
-10 0 20 -1 -21 9 1 -17 2 19	0	0.0013s	20 261	0.567s
2 -3 -5 -1 -2 4 -5 10 -3 -7 2 1 3 0 4 -2 -3 -1	47	25.43s	74 089	2.733s

System	Number of Solutions	Maximal Size of Solutions	Average of the Sizes	Number of Steps	Time of Computation
1 -3 9 -1 81 -9 27 81 -9 1 1 -3 9 1 -81	2	8 098	4 051	411 918	11.333s
-10 7 3 -6 90 10 9 8 -45 2 -7 -5 -3 -2 89	12	78 260	25 130	12 138 342	322.767s
-5 -2 1 -1 1 9 62 -5 -3 101 56 -34 -11 67 -98	7	17 006	5 408	10 978 396	295.867s

12 Conclusion

To summarize the main advantages of this new algorithm,

- it allows to solve a system of linear Diophantine equations as a whole rather than by using Gaussian elimination (as a consequence the number of variables does not change during the computation),
- it can handle non-homogeneous linear equations and monotonous inequalities without overhead,
- it can be made incremental,
- it provides an incremental satisfiability test.

Solving a system sequentially, equation by equation, yields as many variables as there are solutions for each single equation built during the computation, and this number can be really huge. Experiments exemplify this remark: each time such an equation is generated, our algorithm outperforms Gaussian elimination. When it does not happen (for example for systems of one equation) all algorithms perform very closely.

The last three advantages are specific to our algorithm, and are very important for an integration in a constraint logic programming language. This integration is currently under way in the CHIP system [6].

Acknowledgement

We thank Mehmet Dincbas for asking us an incremental version of our algorithm, and Jean-Pierre Jouannaud for his support and comments on this work, not to speak about his patience for reading numerous versions of this paper.

References

[1] Alexandre Boudet, Evelyne Contejean, and Hervé Devie. A new AC-unification algorithm with a new algorithm for solving Diophantine equations. In *Proc. 5th IEEE Symp. Logic in Computer Science, Philadelphia*, June 1990.

[2] Bruno Buchberger. A critical pair/completion algorithm for finitely generated ideals in rings. *LNCS*, 1983. Symposium Rekursive Kombinatorik, Munster, Mai 83.

[3] M. Clausen and A. Fortenbacher. Efficient solution of linear diophantine equations. *Journal of Symbolic Computation*, 8(1&2):201–216, 1989.

[4] Evelyne Contejean and Hervé Devie. Solving systems of linear diophantine equations. In *Proc. 3rd Workshop on Unification, Lambrecht, Germany.* University of Kaiserslautern, June 1989.

[5] Evelyne Contejean and Hervé Devie. Résolution de systèmes linéaires d'équations diophantiennes. *Comptes-Rendus de l'Académie des Sciences de Paris*, 313:115–120, 1991. Série I.

[6] M. Dincbas, P. Van Hentenryck, Helmut Simonis, A. Aggoun, T. Graf, and F.Berthier. The constraint logic programming language CHIP. In *Proc. Int. Conf. on Fifth Generation Computer Systems FGCS-88*, pages 693–702, 1988.

[7] Eric Domenjoud. Outils pour la déduction automatique dans les théories associatives-commutatives. Thèse de doctorat de l'université de Nancy I, 1991.

[8] J. Farkas. Theorie der einfachen ungleichungen. *Journal F.d. reine und anweg Mathematik*, 124:1–27, 1902.

[9] M. Gardner. Mathematical games: Tasks you cannot help finishing no matter how hard you try to block finishing them. *Scientific American*, 24(2):12–21, 1983.

[10] Gérard Huet. An algorithm to generate the basis of solutions to homogeneous linear diophantine equations. *Information Processing Letters*, 7(3), April 1978.

[11] J. L. Lambert. Une borne pour les générateurs des solutions entières positives d'une équation diophantienne linéaire. *Comptes Rendus de l'Académie des Sciences de Paris*, 305:39,40, 1987. Série I.

[12] L. Pottier. Minimal solutions of linear diophantine systems : bounds and algorithms. In *Proceedings of the Fourth International Conference*

on Rewriting Techniques and Applications, pages 162–173, Como, Italy, April 1991.

[13] J. F. Romeuf. A polynomial algorithm for solving systems of two linear diophantine equations. Technical report, Laboratoire d'Informatique de Rouen et LITP, France, 1989.