Solving Linear Diophantine Constraints Incrementally

Evelyne Contejean
Max-Planck-Institut für Informatik
Im Stadtwald
D-6600 Saarbrücken, Germany
contejea@mpi-sb.mpg.de

Abstract

In this paper, we show how to handle linear Diophantine constraints incrementally by using several variations of the algorithm by Contejean and Devie (hereafter called **ABCD**) for solving linear Diophantine systems [4, 5]. The basic algorithm is based on a certain enumeration of the potential solutions of a system, and termination is ensured by an adequate restriction on the search. This algorithm generalizes a previous algorithm due to Fortenbacher [2], which was restricted to the case of a single equation. Note that using Fortenbacher's algorithm for solving systems of Diophantine equations by repeatedly applying it to the successive equations is completely unrealistic: the tuple of variables in the solved equation must then be substituted in the rest of the system by a linear combination of the minimal solutions found in which the coefficients stand for new variables. Unfortunately, the number of these minimal solutions is actually exponential in both the number of variables and the value of the coefficients of the equation solved. In contrast, **ABCD** solves systems faster, without any intermediate blow-up, since it considers the system as a whole. Besides, and this is the new feature described in this paper, it can easily tolerate additional constraints such as membership constraints, linear monotonic inequations, and so on. This is due to the enumeration of tuples which allows a componentwise control of potential solutions. This is not the case with others (more recent) algorithms for solving systems of Diophantine equations, which are based on algebraic and combinatorial techniques [7, 17].

Keywords: Linear Diophantine equations, constraints.

1 Introduction

Solving Diophantine equations is a famous problem which has been widely studied starting with the Greeks. It received even more attention after Hilbert listed it as its 10th open problem. This long study resulted in 1970 in the celebrated paper of Matijasevic [15] who proved that no terminating algorithm can decide whether an arbitrary Diophantine equation has a solution or not. Research, however, proceeded actively for solving subcases, especially the subcase of *linear* Diophantine equations [11, 13, 2, 5, 7, 17]. Indeed, solving such linear equations arises in various areas of computer science. In automated deduction, solving linear homogeneous Diophantine equations surfaces in many important unification problems: unification modulo A [14], modulo AC [18, 9, 1], modulo $AC1$ [8], modulo D [3]. In concurrency, solving linear Diophantine equations shows up in the reachability problem for Petri nets [16], more generally for vector addition systems [10, 12], as well as in the vectorization of FORTRAN programs. In constraint logic programming, linear Diophantine constraints (equations, inequations and disequations) are a basic tool for formalizing concrete problems in operational research, such as resource management applications. For this last kind of applications, incrementality is a crucial requirement which is not met (except in a trivial way) by the algorithms described in [7, 17].

This paper is organized as follows: section 2 recalls the principles of **ABCD** for solving systems of homogeneous linear Diophantine equations. The rest of the material is new. Section 3 deals with the linear constraints mentioned above. Section 4 describes an incremental version of the algorithm, which finally gives rise to an incremental satisfiability test in section 5.

2 The Basic Version of ABCD

In this section, we briefly recall the main principle of **ABCD** in the case of systems of homogeneous linear Diophantine equations. In this case, the set of solutions is empty or is an infinite additive semi-group, and can be represented by its finite subset of *minimal solutions* since any solution is a linear combination of such minimal solutions. Hence, any algorithm providing all minimal solutions of a system is an adequate one. Before describing the standard version of the algorithm, we introduce some definitions and notations.

2.1 Definitions

A system S of p homogeneous linear Diophantine equations with q unknowns may be defined by its associated matrix A which has p rows and q columns:

$$A \equiv \begin{pmatrix} a_{11} \ldots a_{1j} \ldots a_{1q} \\ \vdots \quad \vdots \quad \vdots \\ a_{i1} \ldots a_{ij} \ldots a_{iq} \\ \vdots \quad \vdots \quad \vdots \\ a_{p1} \ldots a_{pj} \ldots a_{pq} \end{pmatrix} \quad S \equiv \left\{ \begin{array}{rcl} a_{11}x_1 + \ldots + a_{1j}x_j + \ldots + a_{1q}x_q & = & 0 \\ \vdots \qquad\qquad \vdots \qquad\qquad \vdots & & \vdots \\ a_{i1}x_1 + \ldots + a_{ij}x_j + \ldots + a_{iq}x_q & = & 0 \\ \vdots \qquad\qquad \vdots \qquad\qquad \vdots & & \vdots \\ a_{p1}x_1 + \ldots + a_{pj}x_j + \ldots + a_{pq}x_q & = & 0 \end{array} \right.$$

We denote by e_j the j^{th} vector in the canonical basis of N^q:

$$e_j = (\underbrace{0,\ldots,0}_{j-1 \text{ times}}, 1, \underbrace{0,\ldots,0}_{q-j \text{ times}})^t = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

In the following, for short, we will not distinguish between column and row vectors. A solution of the system S is a tuple v which fulfills the following conditions:
1. v is in N^q, where N is the set of natural numbers,
2. v belongs to the kernel of the linear application defined by A with respect to the canonical basis.

The set of solutions of S is denoted by $\mathcal{S}$. The set of minimal solutions of S with respect to the strict ordering $\gg$ defined by

$$(x_1,\ldots,x_q) \gg (y_1,\ldots,y_q) \text{ if } \quad \forall i \in [1..q] \ \ x_i \geq y_i \quad \text{and} \quad \exists i \in [1..q] \ \ x_i > y_i$$

is denoted by $\mathcal{M}$. All tuples in $\mathcal{S}$ are linear combinations with natural coefficients of tuples in $\mathcal{M}$.

2.2 Fortenbacher's Algorithm

Assume that the system S is reduced to a single equation. It can then be solved by using Fortenbacher's algorithm [2] as follows: a directed acyclic graph rooted at the canonical vectors is searched breadth-first (indeed, the e_j's are the smallest non-trivial potential solutions of S). If a node $(x_1,\ldots,x_q)$ is neither equal to or greater than an already encountered solution, then its sons are generated by increasing by 1 some of its components under the following restriction:
• generate all tuples $(x_1,\ldots,x_{j-1},x_j+1,x_{j+1},\ldots,x_q)$ such that $Ae_j > 0$, when $A(x_1,\ldots,x_q) < 0$, and
• generate all tuples $(x_1,\ldots,x_{j-1},x_j+1,x_{j+1},\ldots,x_q)$ such that $Ae_j < 0$, when $A(x_1,\ldots,x_q) > 0$.
The constructed graph contains all minimal solutions of S, that is the set $\mathcal{M}$, and is finite.

2.3 ABCD

In a joint work with Hervé Devie [4, 5], we have presented a powerful generalization of Fortenbacher's algorithm for the case of an arbitrary number of equations. The first difficulty was to generalize Fortenbacher's mechanism, and this was obtained via a geometric interpretation of the algorithm. The second was to prove that the generated generalized graph is again finite and contains all minimal solutions.

In order to generate the sons of a node in the search space, the two above distinguished cases ($A(x_1, \ldots, x_q) < 0$ and $A(x_1, \ldots, x_q) > 0$) can be packed in the single following condition:

• generate all tuples $(x_1, \ldots, x_{j-1}, x_j + 1, x_{j+1}, \ldots, x_q)$ such that $A(x_1, \ldots, x_q) \times Ae_j < 0$.

The geometric interpretation of this condition is displayed on figure 1.

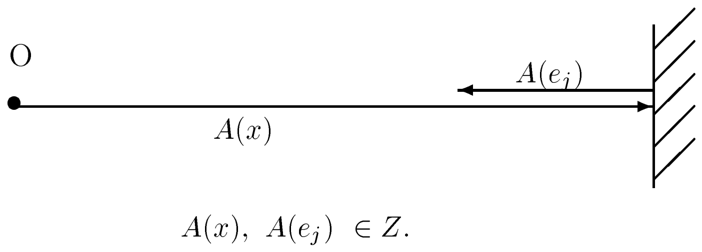


Figure 1: Geometric interpretation of Fortenbacher's restriction.

Now this condition can easily be generalized to a p-dimension space where p stands for the number of equations in S, as displayed on figure 2.

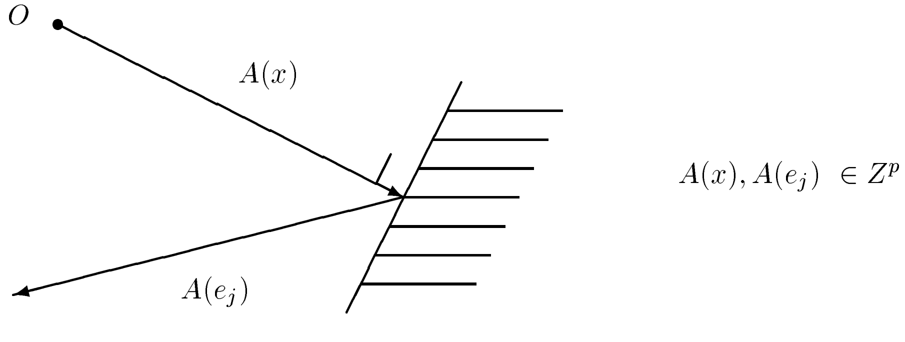


Figure 2: Geometric interpretation of Contejean and Devie's generalized restriction.

ABCD is similar to Fortenbacher's algorithm, except that the condition in order to build the sons of a node x is now:

• generate all tuples $(x_1, \ldots, x_{j-1}, x_j + 1, x_{j+1}, \ldots, x_q)$ such that the extremity of $A(x + e_j)$ lies in the afin half-space containing the origin, and delimited by the hyperplan orthogonal to $A(x)$ which contains the extremity of $A(x)$. This can also be expressed in a very simple analytic way: $A(x_1, \ldots, x_q) \cdot Ae_j < 0$, where $\cdot$ is the scalar product of vectors.

Hence, our algorithm is the following:

- search a directed acyclic graph rooted at $e_1, \ldots, e_q$ breadth-first,
- **if** the node y is equal to or greater than an already encountered solution of $A(x) = 0$, **then** y has no son,
- **otherwise** compute the sons of y by increasing each j-th component of y, yielding $y + e_j$, which satisfies the condition $A(y) \cdot A(e_j) < 0$.

Theorem 2.1 (Contejean and Devie) *Given a system of p homogeneous linear Diophantine equations, the above procedure builds a* finite *directed acyclic graph which contains* all minimal solutions of the system.

The proof of finiteness for the graph is quite complicated and may be found in [1]. It is based on techniques borrowed from real analysis, and consequently, it does not help for studying the complexity of the algorithm. The proof that the graph contains all minimal solutions is an adaptation of Fortenbacher's proof.

Example 2.2 With $S = \left\{ \begin{array}{rcrcrcl} -x_1 & + & 2x_2 & + & x_3 & = & 0 \\ -x_1 & + & x_2 & + & 2x_3 & = & 0 \end{array} \right.$, our algorithm builds the graph drawn on the figure 3.

According to the restriction $A(x) \cdot A(e_j) < 0$, the node $x = (1, 0, 0)$ has only two sons:

$$A(x) \cdot A(e_1) = \begin{pmatrix} -1 \\ -1 \end{pmatrix} \cdot \begin{pmatrix} -1 \\ -1 \end{pmatrix} = (-1 \times -1) + (-1 \times -1) = 2 \geq 0$$

$$A(x) \cdot A(e_2) = \begin{pmatrix} -1 \\ -1 \end{pmatrix} \cdot \begin{pmatrix} 2 \\ 1 \end{pmatrix} = -3 < 0$$

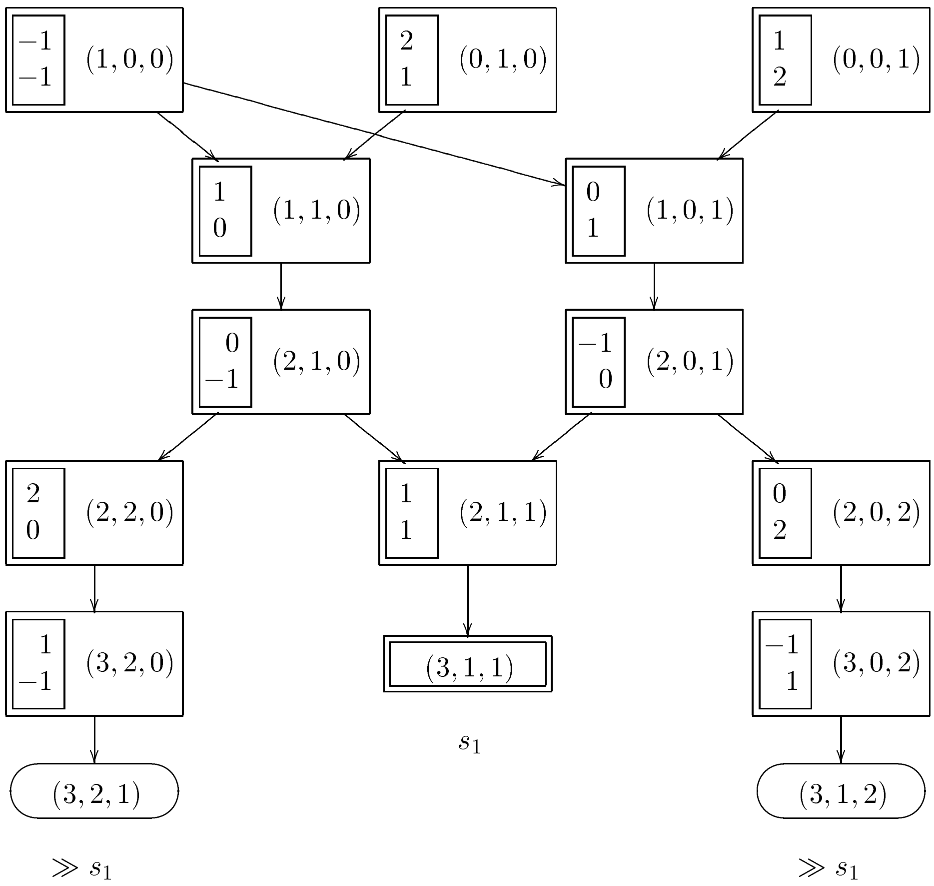
$$A(x) \cdot A(e_2) = \begin{pmatrix} -1 \\ -1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 2 \end{pmatrix} = -3 < 0$$

This graph contains the unique minimal solution $(3, 1, 1)$ of S, and it is finite.

2.4 Refinements

2.4.1 From a graph to a forest

Example 2.2 shows redundancies: some tuples are computed several times, as sons of several distincts fathers. This can be fixed easily by using "frozen components". Assume that for each node x occurring in the graph built by the algorithm, there is a total (arbitrary) ordering $\prec_x$ on its sons. If a node x has two distinct sons $x + e_{j_1}$, and $x + e_{j_2}$ such that $x + e_{j_1} \prec_x x + e_{j_2}$, then the j_1-th component is frozen in the sub-graph rooted at $x + e_{j_2}$: it cannot be increased any more, even if the geometrical condition expressed



Each node contains a triple (x_1, x_2, x_3) of the values of the variables at that node, and when it is relevant, the vector $A(x)$ is written in a box on the left hand side of the node. Solutions are inside double boxes.

Figure 3: A graph built by **ABCD**

by the scalar product is satisfied. This method is still complete, and builds a sub-forest of the original graph.

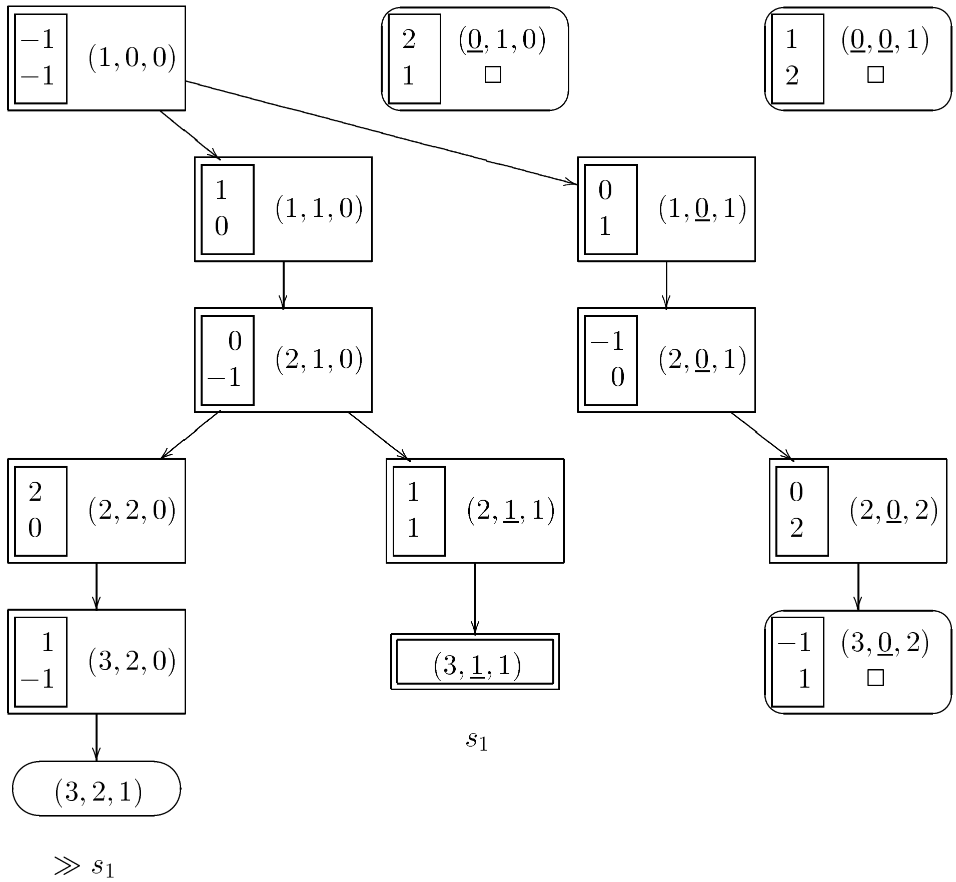
Example 2.3 Applying the above method to the system obtained from our first example, with the ordering $\prec_x$ defined by

$$x + e_{j_1} \prec_x x + e_{j_2} \text{ if } j_1 < j_2$$

results in the forest drawing on figure 4.

2.4.2 From a forest to a stack

The former algorithm builds forests that have an interesting property when the sons of any node x are written from left to right according to the ordering $\prec_x$: if y is a tuple of the forest which is greater than a minimal solution



The underlined components of a tuple are the frozen ones.

Figure 4: A forest built by the ordered version of **ABCD**

of the system s, then s occurs to the left of y in the forest. Such a property enables us to search the forest depth-first, from right to left, which can be done with a stack:

• The initial stack is filled with the e_j's, according to the ordering $\prec_{\vec{0}}$, the smaller one at the bottom, the greater on the top. The higher a tuple is in the stack, the greater is its number of frozen components.
• **if** the top x of the stack has no successor (it is a leaf in the forest-version of the algorithm), **then** it is popped from the stack. Moreover if it is a solution of the system, it is appended to the list of the already computed solutions.
• **otherwise** x is popped from the stack and its successors $x+e_{j_1},\ldots,x+e_{j_k}$ enumerated according to $\prec_x$ are pushed in turn onto the stack.

The forest has been completely searched once the stack is empty. Note that the height of the stack is bounded by q, the number of variables in S, since a tuple occurring at level j in the stack has $j-1$ frozen components.

Example 2.4 The sequence of stacks obtained for our example is drawn on figure 5.

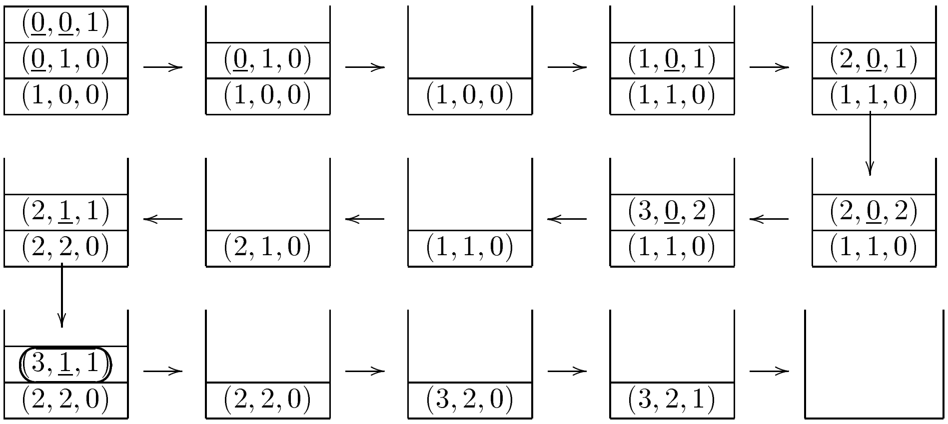


Figure 5: A sequence of stacks built by the stack-version of **ABCD**.

3 Solving linear Diophantine constraints

We now consider additional linear constraints, such as membership constraints and monotonic constraints ($c_1x_1 + c_2x_2 + \ldots + c_qx_q \leq c_{q+1}$, or $c_1x_1 + c_2x_2 + \ldots + c_qx_q \geq c_{q+1}$, with $c_j \in N$). It is well-known that solving $c_1x_1 + c_2x_2 + \ldots + c_qx_q \leq c_{q+1}$ boils down to solving $c_1x_1 + c_2x_2 + \ldots + c_qx_q + x_{q+1} = c_{q+1}$, where x_{q+1} is a variable which does not occur in the problem. Of course, such a method increases the number of variables occurring in the problem, hence the search-space. In contrast, our forthcoming algorithm handles such constraints without adding any new variable. On the other hand, solving systems of heterogeneous linear Diophantine equations will require adding a single new variable for the whole system.

3.1 Heterogeneous linear Diophantine equations

The set of solutions of an heterogeneous linear Diophantine system S can be represented by two finite sets $\mathcal{M}_0$ and $\mathcal{M}_1$, where $\mathcal{M}_0$ is the set of minimal solutions of the homogeneous system associated with S, and $\mathcal{M}_1$ is the set of minimal (with respect to $\gg$) solutions of S. Any solution of S is the sum of a solution in $\mathcal{M}_1$ and a linear combination of the homogeneous solutions in $\mathcal{M}_0$. Solving the heterogeneous system $A\,(x_1, \ldots, x_q)^t = (b_1, \ldots, b_p)^t$ boils

down to solve the homogeneous system $\begin{bmatrix} A & \begin{matrix} -b_1 \\ \vdots \\ -b_p \end{matrix} \end{bmatrix} (x_1, \ldots, x_q, x_{q+1})^t = 0$ and freeze the last component x_{q+1} of each tuple as soon as it is equal to 1. The algorithm returns a set of solutions $\mathcal{M}$ which yields $\mathcal{M}_0$ and $\mathcal{M}_1$ as follows:
• if $(\vec{x}, x_{q+1})$ belongs to $\mathcal{M}$, and x_{q+1} is equal to 0, then $\vec{x}$ belongs to $\mathcal{M}_0$,
• if $(\vec{x}, x_{q+1})$ belongs to $\mathcal{M}$, and x_{q+1} is equal to 1, then $\vec{x}$ belongs to $\mathcal{M}_1$.
Since the last component has to be frozen as soon as its value is equal to 1, there are no other cases.

3.2 Linear monotonic constraints

Assume that the linear system to be solved is built from a non-empty set of linear equations $A(x) = 0$ and monotonic constraints of the form $c_{i1}x_1 + c_{i2}x_2 + \ldots + c_{iq}x_q \leq c_i$ with $c_{ij}, c_i \in N$. These constraints are called monotonic since the linear mapping $(x_1, \ldots, x_q) \mapsto c_{i1}x_1 + c_{i2}x_2 + \ldots + c_{iq}x_q$ is monotonic with respect to the natural ordering $\gg$ on N^q. They are written $C(x) \leq c$. This case generalizes the previous one: indeed, solving $Ax = b$ boils down to solve $Ax - bx_{q+1} = 0 \wedge x_{q+1} \leq 1$. Concerning the representation of the solutions, again, this can be achieved thanks to two finite sets: $\mathcal{M}_0$ and $\mathcal{M}_c$. $\mathcal{M}_0$ is the set of minimal solutions of $A(x) = 0 \wedge C(x) = 0$, whereas $\mathcal{M}_c$ is the set of the solutions of $A(x) = 0 \wedge C(x) \neq 0 \wedge C(x) \leq c$, which are not greater than a solution in $\mathcal{M}_0$. Any solution of $A(x) = 0 \wedge C(x) \leq c$ is the sum of a solution in $\{\vec{0}\} \cup \mathcal{M}_c$ and a linear combination of the homogeneous solutions of $\mathcal{M}_0$. The algorithm for solving $A(x) = 0 \wedge C(x) \leq c$ works as follows:

• Initialize $\mathcal{M}_0$ and $\mathcal{M}_c$ with $\emptyset$ and start searching a directed acyclic graph rooted at $e_1, \ldots, e_q$ breadth-first,
• **if** the node y is not a solution of $C(x) \leq c$, **then** y has no son,
• **if** the node y is a solution of $A(x) = 0 \wedge C(x) \leq c$, **then**

  – **if** y is a solution of $C(x) = 0$, **then** add y to the set $\mathcal{M}_0$. In this case y has no son,
  – **otherwise**, $C(x) \neq 0$, **if** y is not greater or equal to a solution in $\mathcal{M}_0$, add y to the set $\mathcal{M}_c$. In this last case, the sons of y are $y + e_1, y + e_2, \ldots, y + e_q$, otherwise y has no sons.

• **otherwise** compute the sons of y by increasing each j-th component of y, yielding $y + e_j$, which satisfies the condition $A(y) \cdot A(e_j) < 0$.

Theorem 3.1 *The above method builds a finite graph which contains $\mathcal{M}_0$ and $\mathcal{M}_c$, and any solution of $A(x) = 0 \wedge C(x) \leq c$ is composed with a member of $\mathcal{M}_c$ and a linear combination of members of $\mathcal{M}_0$.*

The monotonicity of th mapping $x \mapsto C(x)$ ensures the completeness of the algorithm, and the proof of finiteness of the graph can be adapted from the original one.

3.3 Linear constraints on finite domains

In practice, there are some cases where the constraints have to be solved in finite domains, simply because there are finitely many objects in operational research problems. As the reader should have noticed, we have restricted ourselves to linear monotonic constraints of the form $C(x) \leq c$ in the case of non-finite domains (of course, it is always possible to handle any linear constraints, even non monotonic, by adding new variables). In finite domains, we do not need to worry about the finiteness of the search space anymore. Hence we can handle a wider class of linear constraints, such as $D(x) \geq d$, as well as membership constraints of the form $x_j \in [l_i^1...u_i^1] \cup \ldots \cup [l_i^k...u_i^k]$. Since the search space is finite, any system has a finite number of solutions which can be enumerated as explained below.

Note that a large amount of finite domains can be handled by adding memberships constraints in conjunction with the original ones. These domains may or may not contain the origin.

Assume now that the system is built from a non-empty set $A(x) = 0$ of equations, a set of linear monotonic constraints $c_{i1}x_1 + c_{i2}x_2 + \ldots + c_{iq}x_q < c_i$ with $c_{ij}, c_i \in N$, written $C(x) < c$, $d_{i'1}x_1 + d_{i'2}x_2 + \ldots + d_{i'q}x_q > d_{i'}$ with $d_{i'j}, d_{i'} \in N$, written $D(x) > d$, and membership constraints $x_j \in I_j = [l_j^1...u_j^1] \cup \ldots \cup [l_j^{k_j}...u_j^{k_j}]$, written $x \in I$. Assume in addition that the solutions belong to a finite domain $[0...d_1] \times \ldots \times [0...d_q] \subseteq N^q$.

The algorithm for solving $A(x) = 0 \wedge C(x) \leq c \wedge D(x) \geq d \wedge x \in I$ in the finite domain $[0...d_1] \times \ldots \times [0...d_q]$ works as follows:

- Initialize $\mathcal{S}$ with $\emptyset$ and start searching a directed acyclic graph rooted at $(l_1, l_2, \ldots, l_q)$,
- **if** the node y is not a solution of $C(x) \leq c$, **then** y has no son,
- **if** y is a solution of $C(x) \leq c \wedge A(x) \neq 0$, **then** compute its sons by increasing each j-th component of y, until it reaches the lower bound of $[y_j + 1...d_j] \cap I_j$, which satisfies the condition $A(y) \cdot A(e_j) < 0 \wedge [y_j + 1...d_j] \cap I_j \neq \emptyset$.
- **if** y is a solution of $C(x) \leq c \wedge A(x) = 0$, **then** compute its sons by increasing each j-th component of y, until it reaches the lower bound of $[y_j + 1...d_j] \cap I_j$, which satisfies the condition $[y_j + 1...d_j] \cap I_j \neq \emptyset$. If y is also a solution of $D(x) \leq d$, then add y to $\mathcal{S}$.

The above algorithm is very similar to the previous one for solving $A(x) = 0 \wedge C(x) \leq c$ in N^q. The only difference comes from the finiteness of the domain which allows to consider constraints of the form $D(x) \geq d$ which are actually lower bounds on the value of the solutions. Indeed, if we apply this algorithm to the system

$$\begin{array}{ccccccc} x_1 & + & x_2 & - & x_3 & = & 0 \\ x_1 & - & x_2 & + & x_3 & = & 0 \\ x_1 & & & & & > & 3 \end{array}$$

in N^q, the graph is not finite.

A stronger version of the algorithm, however, allows to deal with a single such constraint on the variables of the problem (using the fact that the negation of such a single constraint is a constraint of the other form).

Theorem 3.2 *The above algorithm constructs a finite graph which contains all solutions of $A(x) = 0 \wedge C(x) \leq c \wedge D(x) \geq d \wedge x \in I$ in the finite domain $[0...d_1] \times \ldots \times [0...d_q]$.*

The finiteness of the graph is obvious since all its nodes are in the finite domain $\mathcal{D}$. The condition that the jth component of y is increased (to its next value) only if the scalar product $A(y) \cdot A(e_j)$ is negative or $A(y)$ itself is null, is still complete, because it allows to reach all solutions which are greater than y.

As in the standard case, there is also a forest-version and a stack-version of this algorithm.

4 Incrementality

The goal now is to split the search of the set of minimal solutions of $A(x) = 0 \wedge B(x) = 0$ into a search of the set $\mathcal{M} = \{m_1, \ldots, m_n\}$ of minimal solutions of $A(x) = 0$, followed by a search acting on $B(x) = 0$. We are of course not interested in the “British Museum” method, which would consist here in solving the new system

$$\sum_{j=1}^{j=n} B(m_j) z_j = 0$$

built from $\mathcal{M}$ and $B(x) = 0$, before to combine the solutions in $\mathcal{M}$ according to the coefficients which are solutions of the above system. In contrast, our method searches a provably smaller space. The idea is to increment the current vector y labeling a node in the graph by the solutions in $\mathcal{M}$ replacing the vectors e_i of the canonical basis:

• search a direct acyclic graph rooted at $m_1, \ldots, m_n$ breadth-first,
• **if** a node y is equal to or greater than an already computed solution, **then** y has no son,
• **otherwise** generate its jth potential son by increasing each component of y by the corresponding component of m_j, yielding $y + m_j$, when m_j satisfies the condition $B(y) \cdot B(m_j) < 0$.

The graph built by this method is the image by the linear mapping L

$$L : \quad N^n \to N^q$$
$$e_j \mapsto m_j$$

of the "top" of the graph built by the graph-version of the algorithm for the system $\sum_{j=1}^{j=n} B(m_j)y_j = 0$. Indeed, L maps some internal nodes of the first graph to some leaves in the new graph. This is so, because L is not compatible with $\gg$, and hence we detect that a linear combination of m_j's is greater than a global solution sooner. This remark ensures the termination of this algorithm, and the fact that our algorithm is *always better* than the classical one, that is that the number of computed nodes is less than or equal to the number of nodes in the British Museum method. Since any solution of $A(x) = 0 \wedge B(x) = 0$ is also a solution of $A(x) = 0$, it is a linear combination of the m_j's, and the proof of completeness is now the same as in the standard case.

Example 4.1

$$\begin{array}{rcrcrcr} A(x) & = & -\ x_1 & + & 3x_2 & - & x_3 \\ B(x) & = & 2x_1 & + & x_2 & - & x_3 \end{array}$$

The set of minimal solutions of $A(x) = 0$ is equal to

$$\{(3,1,0),(2,1,1),(1,1,2),(0,1,3)\}.$$

The incremental algorithm builds the graph displayed on figure 6 in order to solve $A(x) = 0 \wedge B(x) = 0$.

This graph is included or equal (up to the linear mapping L) to the top of the graph built by the standard algorithm for solving the system

$$B(m_1)y_1 + B(m_2)y_2 + B(m_3)y_3 + B(m_4)y_4 = 0 \quad \equiv \quad 7y_1 + 4y_2 + y_3 - 2y_4 = 0$$

which is displayed on figure 7.

Again, there is a forest-version of the incremental algorithm, but no stack-version since the orderings we deal with are not compatible with the linear mapping L, and it can be seen on the above example that some nodes may be greater than a solution which occurs at its left hand side.

Remark : If some (new) variables occur in the second system, but not in the first one, the above method still applies. Assume we want to solve $\sum_{j=1}^{j=q} a_{ij}x_j = 0,\ 1 \le i \le p \wedge \sum_{j=1}^{j=q+q'} b_{ij}x_j = 0,\ 1 \le i \le p'$, and we have

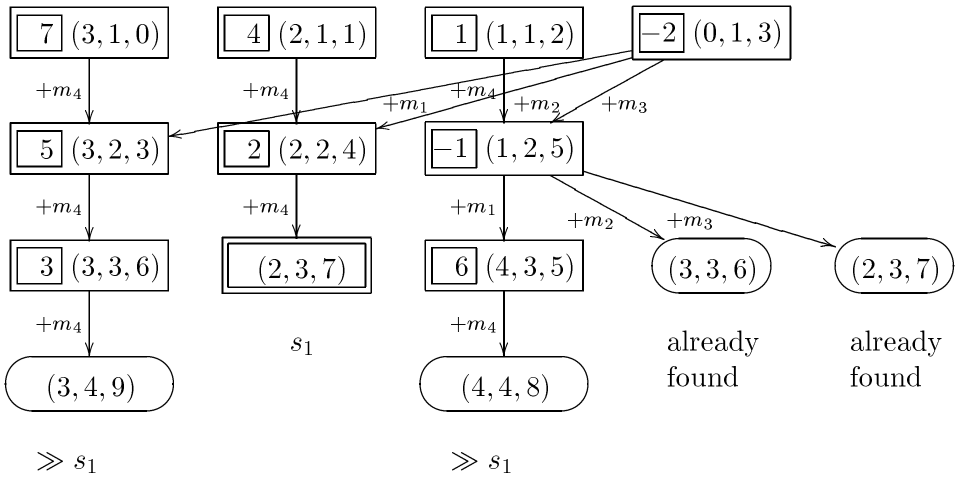


Figure 6: Incremental version of **ABCD**

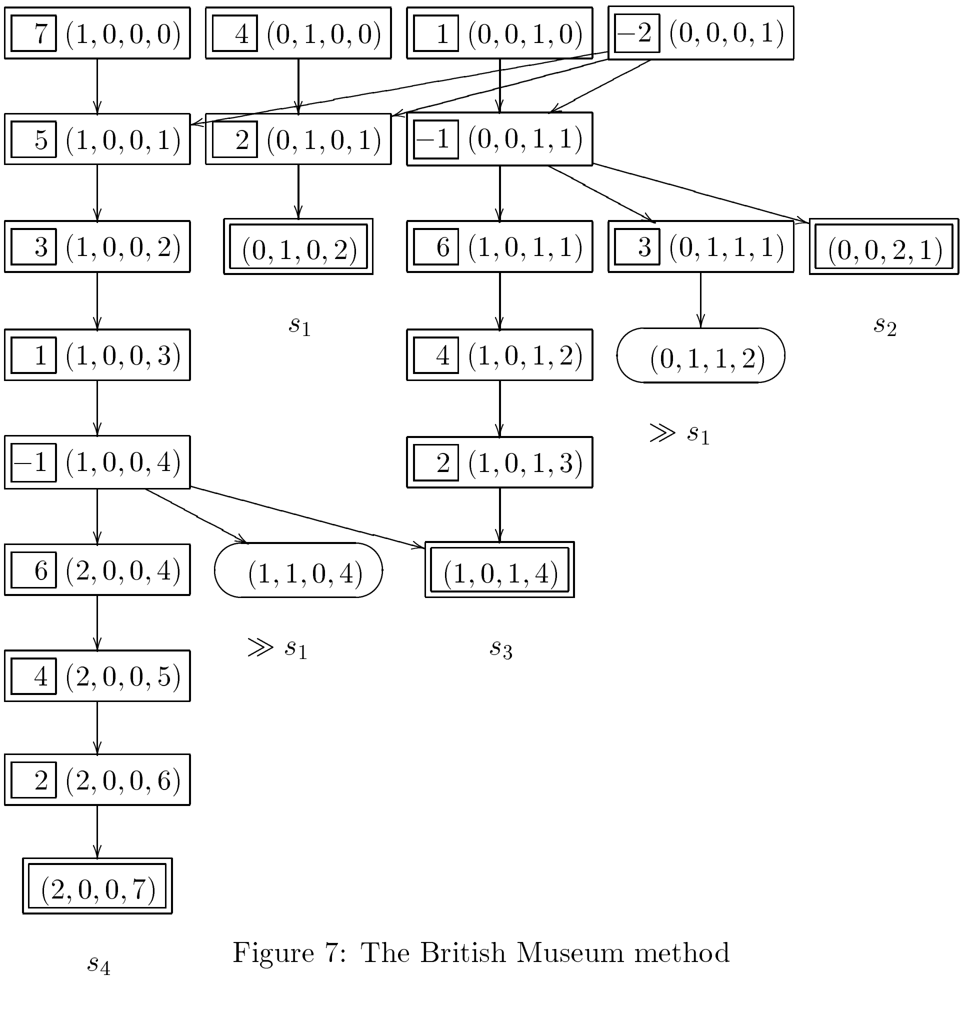


Figure 7: The British Museum method

already computed the set $\mathcal{M} = \{m_1, \ldots, m_n\}$ of minimal solutions of $\sum_{j=1}^{j=q} a_{ij}x_j = 0,\ 1 \leq i \leq p$ in N^q. The set $\mathcal{M}'$ of minimal solutions of $\sum_{j=1}^{j=q} a_{ij}x_j + 0(\sum_{j=q+1}^{j=q+q'} x_j) = 0,\ 1 \leq i \leq p$ in $N^{q+q'}$ is equal to

$$\{(m_1, \underbrace{0, \ldots, 0}_{q'}), \cup(m_2, \underbrace{0, \ldots, 0}_{q'}), \ldots, (m_n, \underbrace{0, \ldots, 0}_{q'})\} \cup e_{q+1}, e_{q+2}, \ldots e_{q+q'}\},$$

where $e_{q+1}, e_{q+2}, \ldots e_{q+q'}$ are the q'-th last vectors in the canonical basis of $N^{q+q'}$. The incremental algorithm simply uses $\mathcal{M}'$ instead of $\mathcal{M}$, during the initialization and as a set of increments, but works as described above.

5 An incremental satisfiability test

ABCD's stack version provides obviously a satisfiability test for a system of linear Diophantine equations, by stopping at the first solution. More interesting is the fact that this test is incremental.

Definition 5.1 *Let $S \equiv S_1 \wedge S_2 \wedge ... \wedge S_k$ be a system of linear Diophantine equations made of k non-empty subsystems S_i, and q the number of unknowns in S. We define recursively a sequence of stacks $St_i(S)$ as follows:*
- *$St_0(S)$ is obtained by stacking the vectors of the canonical basis.*
- *$St_{i+1}(S)$ is the stack obtained by running* **ABCD***'s stack-version on the input system $S_1 \wedge \ldots \wedge S_i \wedge S_{i+1}$ with the $St_i(S)$ as initial stack, and stopping as soon as a solution is found (or the stack becomes empty).*

Theorem 5.2 *Let $S \equiv S_1 \wedge S_2 \wedge ... \wedge S_k$ be a system of linear Diophantine equations. S has a solution if and only if $St_k(S)$ is non empty, in which case, the top of $St_k(S)$ is a solution of S.*

The proof of this theorem is based on the following lemma:

Lemma 5.3 *Let $S \equiv S_1 \wedge S_2 \wedge ... \wedge S_k$ be a system of linear Diophantine equations. For all i in $[1...k]$,* **ABCD***'s stack-version applied to the system $S_1 \wedge \ldots \wedge S_i$ with the stack initialized to $St_{i-1}(S)$ yields all minimal solutions of $S_1 \wedge \ldots \wedge S_i$ and stops.*

Proof: By induction on i. If i is equal to 1, we are back to the non-incremental version of **ABCD**. Assuming the lemma true for i, we now show that it is still true for $i+1$. Let s be a minimal solution of $S_1 \wedge \ldots \wedge S_i \wedge S_{i+1}$. s is also a solution of $S_1 \wedge \ldots \wedge S_i$. There are two cases.

First case: s is a minimal solution of $S_1 \wedge \ldots \wedge S_i$. By induction hypothesis, s can be obtained from a tuple which belongs to St_{i-1}. The stack-algorithm is deterministic, hence St_i is reached when the stack-algorithm is initialized with St_i. Now, s is either on the top of St_i if it is the first enumerated solution, or it can be obtained from a tuple belonging to St_i.

Second case: s is greater than some minimal solution of $S_1 \wedge \ldots \wedge S_i$. Then s is a linear combination of some minimal solution $m_1, \ldots, m_l$ of $S_1 \wedge \ldots \wedge S_i$. The algorithm enumerates the minimal solutions in a deterministic way. We can assume without loss of generality that m_j is found before m_{j+1} for $j \in [1...l-1]$. m_l can be obtained from a tuple t in S_i (see the first case). We now show that s can also be obtained from t. Let i be a frozen component of t. t comes from a tuple a which can be incremented at least on two components i and i', yielding $a + e_i$ and $a + e_{i'}$, and t comes from $a + e_{i'}$ where i is a frozen component:

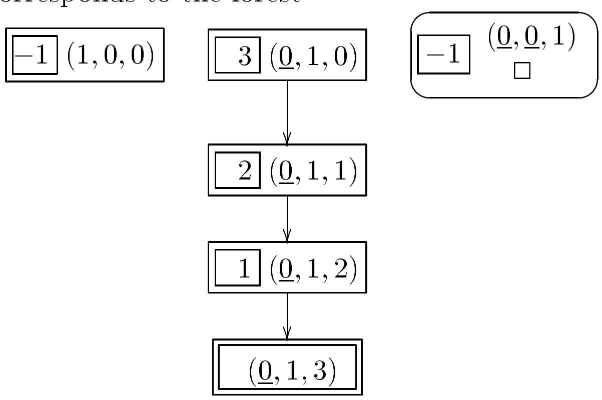
$$\begin{tikzcd} & a \arrow[dl] \arrow[dr] & \\ a + e_i & & a + e_{i'} \arrow[d, dotted, no head] \\ & & t \arrow[d, dotted, no head] \\ & & m_k \end{tikzcd}$$

If the i-th component of s is strictly greater than the i-th component of m_k, then s is greater than $a + e_i$. Hence there is a minimal solution in the sub-graph rooted at $a + e_i$ which is smaller than s. This in contradiction with the fact that m_k is the last minimal encountered solution smaller than s. Hence s and t are equal on the frozen components of t, and s is greater than s. Since the stack-version of **ABCD** is complete, s comes from a tuple (t) which belongs to S_{i+1}.

Example 5.4 Consider the following systems:

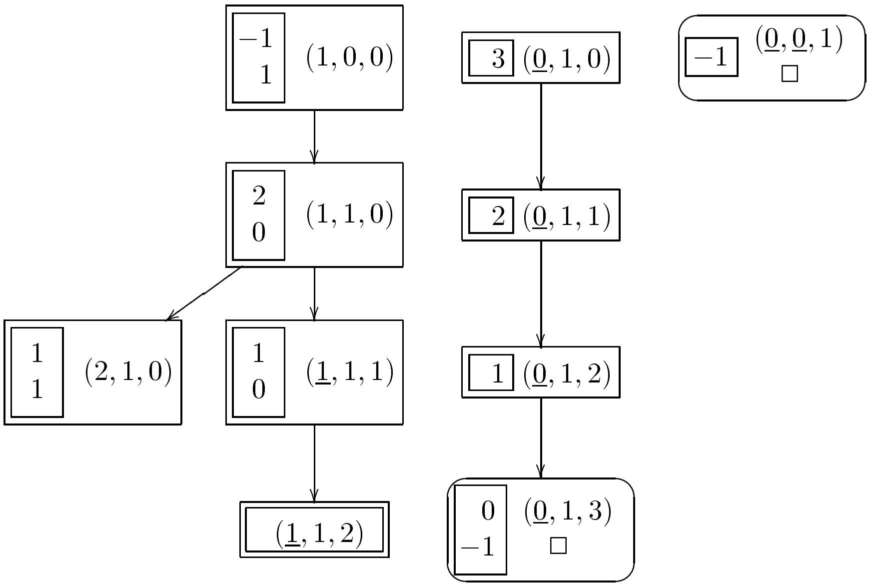
$$\begin{array}{lcrcrcrcl} S_1 & \equiv & - & x_1 & + & 3x_2 & - & x_3 & = & 0 \\ S_2 & \equiv & & x_1 & - & x_2 & & & = & 0 \end{array}$$

The stack-algorithm stops with the stack $\begin{array}{|c|}\hline (\underline{0},1,3) \\ \hline (1,0,0) \\ \hline \end{array}$ when testing the satisfiability of S_1, which corresponds to the forest



s_1

Now, for testing the satisfiability test of $S_1 \wedge S_2$, the algorithm stops with the stack $\begin{array}{|c|}\hline (\underline{1},1,2)\\ \hline (2,1,0)\\ \hline\end{array}$ which corresponds to the forest



s'_1

Remark : Again, some new variables $x_{q+1}, \ldots, x_{q+q'}$ may appear in the last subsystems, and not in the first ones. This can be handled by adding the corresponding component with the frozen value 0 to the already computed nodes as well as some new roots $e_{q+1}, \ldots, e_{q+q'}$ in the forest version, corresponding to some new tuples in the the bottom of the stack in the stack version. The ordering used in order to avoid the redundancies has to fulfils the following condition:
$x + e_{j_1} \prec_x x + e_{j_2}$ if $j_1 > q, j_2 \leq q$. From a theoretical point of view, we have solved the problem of making incremental the obvious satisfiability test derived from our algorithm. We have still to investigate whether such a test is really better than restarting the checking from the beginning with the whole new system.

6 Conclusion

To summarize, **ABCD** and its variants have several advantages which make it suitable for constraint logic programming applications:
• 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 heterogeneous linear equations and monotonic inequalities without overhead,

• it can be made incremental,
• it provides an incremental satisfiability test.

The last three advantages are very important for an integration in a constraint logic programming language. This integration is currently under way in CHIP [6] in cooperation with COSYTEC. We have not yet compared this algorithm with the method based on the combination of simplex with cutting planes. On the other hand **ABCD** alone is not as efficient as the constraint propagation (used *e.g.* in CHIP) on some examples in the case of finite domains. However, **ABCD** is compatible with the constraint propagation, and we shall compare the combination of **ABCD** with the constraint propagation with the other existing methods for the case of finite domains.

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.

This research is part of COLINE, a cooperative project between University Paris-Sud at Orsay and COSYTEC SA. It is supported by the French Ministère de la Recherche et de l'Espace, décision d'aide n^o 92 S 0600.

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

[3] Evelyne Contejean. A partial solution for D-unification based on a reduction to AC1-unification. In *Proc. ICALP 93*, Lund, Sweden, July 1993.

[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. Solving systems of linear diophantine equations: An algebraic approach. In *Proc. 16th Mathematical Foundations of Computer Science, Warsaw*. Springer-Verlag, 1991.

[8] François Fages. Associative-commutative unification. *Journal of Symbolic Computation*, 3(3), June 1987.

[9] Alexander Herold and Jorg H. Siekmann. Unification in abelian semi-groups. *Journal of Automated Reasoning*, 3(3):247–283, 1987.

[10] J. Hopcroft and J.J. Pansiot. On the reachability problem for 5-dimenensional vector addition systems. *Theoretical Computer Science*, 8:135–159, 1978.

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

[12] S. R. Kozaraju. Decidability of reachbility in vector addition systems. In *Proc. 14 th ann. ACM STOC*, pages 267–281, 1982.

[13] 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.

[14] G.S. Makanin. The problem of solvability of equations in a free semigroup. *Akad. Nauk. SSSR*, 233(2), 1977.

[15] J. V. Matijasevic. Enumerable sets are diophantine. *Soviet Mathematics (Dokladi)*, 11(2):354–357, 1970.

[16] Gérard Memmi. *Méthodes d'analyse de reseaux de Petri. Réseaux à files. Applications au temps réél.* Thèse d'Etat, Univ. Pierre et Marie Curie, France, 1983.

[17] 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.

[18] M. Stickel. A unification algorithm for associative-commutative functions. *Journal of the ACM*, 28(3):423–434, 1981.