US 20170315958A1

(19) **United States**

(12) **Patent Application Publication**

Nieuwenhuis

(10) **Pub. No.: US 2017/0315958 A1**

(43) **Pub. Date: Nov. 2, 2017**

(54) # COMPUTER-IMPLEMENTED METHOD FOR SOLVING SETS OF LINEAR ARITHMETIC CONSTRAINTS MODELLING PHYSICAL SYSTEMS

(71) Applicant: **BARCELOGIC SOLUTIONS S.L.**, Barcelona (ES)

(72) Inventor: **Robert L. M. Nieuwenhuis**, Barcelona (ES)

(21) Appl. No.: **15/651,122**

(22) Filed: **Jul. 17, 2017**

Related U.S. Application Data

(63) Continuation-in-part of application No. 14/192,909, filed on Feb. 28, 2014.

Publication Classification

(51) **Int. Cl.**

G06F 17/12 (2006.01)

G06F 17/10 (2006.01)

(52) **U.S. Cl.**

CPC ***G06F 17/12*** (2013.01); ***G06F 17/10*** (2013.01)

(57) **ABSTRACT**

A computer-implemented method for solving sets of linear arithmetic constraints modelling physical systems by programmed execution of mathematical operations in a processor unit, wherein the programmed execution of mathematical operations decide, given a set of constraints S, whether S has any solution, and if so, find one or more of them.

COMPUTER-IMPLEMENTED METHOD FOR SOLVING SETS OF LINEAR ARITHMETIC CONSTRAINTS MODELLING PHYSICAL SYSTEMS

CROSS REFERENCE TO RELATED APPLICATIONS

[0001] This application is a Continuation-in-Part of U.S. patent application Ser. No. 14/192,909, filed Feb. 28, 2014, the contents of such application being incorporated by reference herein.

FIELD OF THE INVENTION

[0002] The invention relates to data processing generally, and more particularly, to data processing under the guidance of a computer implemented method for search-based integer linear programming (ILP), involving the programmed execution of mathematical operations in a processor unit for deciding, given a set of constraints S, whether S has any solution, and if so, finding one or more of them.

Definitions

[0003] Along this description following notions/terms will be used:

[0004] A constraint over a finite set of variables, X $\{x_1 \ldots x_n\}$ is an expression of the form $a_1x_1 + \ldots + a_nx_n \leq a_0$, in which the coefficients $a_0 \ldots a_n$ are integer numbers.

[0005] A solution for a set S of constraints or integer program (IP) over $\{x_1 \ldots x_n\}$ is a function Sol mapping each variable x of $\{x_1 \ldots x_n\}$ to an integer value Sol(x) such that all constraints are satisfied, that is, for each constraint of the form $a_1x_1 + \ldots + a_nx_n \leq a_0$, the integer number $a_1.\mathrm{Sol}(x_1) + \ldots + a_n \cdot \mathrm{Sol}(x_n)$ is smaller than or equal to a_0.

[0006] Optimization: maximizing (or minimizing) an objective function (or a cost function), an expression of the form $a_1x_1 + \ldots + a_nx_n$, that is, finding a solution Sol such that $a_1.\mathrm{Sol}(x_1) + \ldots + a_n \cdot \mathrm{Sol}(x_n)$ is maximized (minimized).

[0007] MIP: Solving Mixed IPs (MIPs): finding solutions where some variables must take integer values and others can be arbitrary rationales.

[0008] A lower bound for a variable x is an expression of the form $a \leq x$, where a is an integer number, and an upper bound for a variable x is an expression of the form $x \leq a$, where a is an integer number and a bound is an expression that is either a lower bound or an upper bound.

[0009] The negation of a lower bound $a \leq x$ is the upper bound $x \leq a-1$ and the negation of an upper bound $x \leq a$ is the lower bound $a+1 \leq x$.

[0010] A lower bound $a_1 \leq x$ and an upper bound $x \leq a_2$ are called conflicting if $a_1 > a_2$.

[0011] A lower bound $a \leq x$ is called new in a given set of bounds B if there is no lower bound $a' \leq x$ in B with $a' \geq a$, and an upper bound $x \leq a$ is called new in a given set of bounds B if there is no upper bound $x \leq a'$ in B with $a' \leq a$, and a variable x is called defined to the value a in a given set of bounds B, if B contains the bounds $a \leq x$, and $x \leq a$.

[0012] A monomial is an expression of the form a x, where a is an integer or a rational number and x is a variable. It is called negative if a is negative and positive otherwise.

[0013] Propagation:

[0014] If C is a linear arithmetic constraint of the form $a_1x_1 + \ldots + a_nx_n \leq a_0$ where:

[0015] the subset of positive monomials of $\{a_1x_1, \ldots, a_nx_n\}$ is $\{ax, c_1y_1, \ldots, c_py_p\}$;

[0016] the subset of negative monomials of $\{a_1x_1 \ldots a_nx_n\}$ is $\{d_1z_1, \ldots, d_qz_q\}$;

[0017] R is a set of bounds $\{l_1 \leq y_1, \ldots, l_p \leq y_p, z_1 \leq u_1, \ldots, z_q \leq u_q\}$;

[0018] u is the largest integer such that $u \leq (a_0 - c_1l_1 - \ldots - c_pl_p - d_1u_1 - \ldots - d_qu_q)/a$, then C and R propagate the upper bound $x \leq u$.

[0019] For example, if C is $2x+3y+3z \leq 13$ and R is $\{1 \leq x, 2 \leq y\}$ then C and R propagate $z \leq 1$, since 1 is the largest integer u such that $u \leq (13-2 \cdot 1-3 \cdot 2)/3=(13-8)/3=5/3$.

[0020] For example, if C is $2x \leq 13$ and R is the empty set, then C and R propagate $x \leq 6$, since 6 is the largest integer u such that $u \leq 13/2$.

[0021] If C is a linear arithmetic constraint of the form $a_1x_1 + \ldots + a_nx_n \leq a_0$ where:

[0022] the subset of positive monomials of $\{a_1x_1, \ldots, a_nx_n\}$ is $\{c_1y_1, \ldots, c_py_p\}$;

[0023] the subset of negative monomials of $\{a_1x_1, \ldots, a_nx_n\}$ is $\{ax, d_1z_1, \ldots, d_qz_q\}$;

[0024] R is a set of bounds $\{l_1 \leq y_1, \ldots, l_p \leq y_p, z_1 \leq u_1, \ldots, z_q \leq U_q\}$;

[0025] l is the smallest integer such that $l \geq (a_0 - c_1l_1 - \ldots - c_pl_p - d_1u_1 - \ldots - d_qu_q)/a$, then C and R propagate the lower bound $l \leq x$.

[0026] For example, if C is $2x+3y-3z \leq 13$ and R is $\{1 \leq x, 2 \leq y\}$ then C and R propagate $-1 \leq z$, since -1 is the smallest integer l such that $l \geq (13-2 \cdot 1-3 \cdot 2)/-3=(13-8)/-3=-5/3$.

[0027] Conflicting Subset or CSS, is a data structure storing a set of bounds.

[0028] Conflicting constraint or CC, is a data structure storing a linear arithmetic constraint.

[0029] Cut

[0030] If C_1 is a linear arithmetic constraint $a_1x_1 + \ldots + a_nx_n \leq a_0$ and C_2 is a linear arithmetic constraint $b_1x_1 + \ldots + b_nx_n \leq b_0$, then a cut between C_1 and C_2 is a linear arithmetic constraint $c_1x_1 + \ldots + c_nx_n \leq c_0$ such that c and d are positive natural numbers and $c_i = c \cdot a_i + d \cdot b_i$ for each i in $0 \ldots n$; and

[0031] If $c_j=0$ for some j in $1 \ldots n$ then this cut is said to eliminate the variable x_j.

[0032] Learning a constraint

[0033] a propagation record is a triple (b, R, C) where b is a bound C is a linear arithmetic constraint and R is a set of bounds such that C and R propagate b, then R being termed the reason set of b and C being termed the reason constraint of b; in a special kind of propagation record called a decision, the components R and C are null,

[0034] a propagation stack is a data structure having capabilities of a standard stack data structure whose elements are propagation records, with standard operations for pushing and popping elements and for inspecting the topmost element and in addition the nonstandard capability of inspecting non-topmost elements;

[0035] a bound b is said to be in a propagation stack B if b is the first element of some propagation record of B; similarly a set of bounds R is said to be in a propagation stack B if R is a subset of the set of all first elements of the propagation records of B,

[0036] a constraint C is said to be learned when it is added to the set of linear arithmetic constraints S.

BACKGROUND OF THE INVENTION

[0037] Efficient ILP is crucial for many applications. For example, to find a feasible or optimal schedule in a limited period of time for a set of industrial tasks, where each task has a given duration and requires certain amounts of different limited resources (machines, trucks, employees). ILP (as well as SAT, see below) is NP-complete: no efficient (polynomial) algorithm for it has been found and the existence of such a polynomial algorithm is considered unlikely.

[0038] The use of computer implemented ILP methods, models or algorithms for automatically solving with the aid of a processor unit, different integer problems expressed in the form of a set S of constraints appears disclosed in the following patents U.S. Pat. No. 7,653,561, U.S. Pat. No. 8,515,280, U.S. Pat. No. 8,402,396 and patent applications US 2011/0153709 and US 2012/0250500 addressing different technical fields.

[0039] Just about any discrete optimization problem is an IP or a MIP: scheduling, routing, planning, configuration, timetabling, etc.

[0040] One concrete physical application is the 'knapsack' problem that is following detailed.

[0041] For instance, a truck will be going from A to B. There are n different types of items $\{1 \ldots n\}$ to be carried, where each type of item i has a_i units available, and each unit of it weights w_i kg and brings a profit of p_i per carried unit.

[0042] The problem is to decide how many units x_i of each item type i to carry, without exceeding the truck's total capacity of K kg, in order to make a total profit of at least P$: the IP will consist of $w_1x_i + \ldots + w_nx_n \le K$ and $p_1x_1 + \ldots + p_n x_n \ge P$, with initial bounds $0 \le x \le a$.

[0043] The corresponding optimization problem is, instead of requiring the total profit $p_1x_1 + \ldots + p_nx_n$ to be at least P $, to maximize it.

[0044] There are numerous extensions of this problem, such as further constraints on, e.g., a maximal total number of units carried of certain subclasses of items, more than one truck, etc.

[0045] Most current ILP methods work by iteratively solving LP relaxations, i.e., first finding rational (possibly non-integer) solutions for the set of constraints. Additional steps are then performed to progressively turn these solutions into an integer one, for example by cutting-plane or Branch-and-cut methods.

[0046] The method described in this patent application performs no LP relaxations. It does a systematic search over the set of possible integer solutions. It borrows ideas from SAT solving, which can be seen as the special case of ILP where the variables $x_1 \ldots x_n$ can only take the values 0 or 1 (as in 0/1 integer programming) and where constraints are of the form $1\cdot x_1 + \ldots + 1\cdot x_m - 1\cdot y_1 - \ldots - 1\cdot y_n \le m-1$, expressed as clauses $\{\overline{x}_1, \ldots, \overline{x}_m, y_1, \ldots, y_n\}$ i.e., sets (disjunctions) of literals, where a literal is a either variable x or a negated variable $\overline{x}$.

[0047] A basic SAT solving method is DPLL [1, 2] which comprises the following steps maintaining a partial assignment A, written here as a stack of literals that grows to the right:

[0048] 1. start with an empty partial assignment A

[0049] 2. propagate while possible: extend A to A l if there is some clause $\{l\} \cup C$ with all its variables assigned in A except the one of l, and $A \cap C = \emptyset$

[0050] 3. if there is some conflict, a clause C with all variables assigned and $A \cap C = \emptyset$, then go to step 6

[0051] 4. if all variables are assigned and there is no conflict, halt with solution A

[0052] 5. decide: take some unassigned variable x and extend A to A x or to A $\overline{x}$; here the literal x or $\overline{x}$ is called a decision

[0053] 6. backtrack: if there is some conflict and A is of the form $A_1\, l\, A_2$, where l is the rightmost decision in A, then replace A by $A_1\, \overline{l}$ (where $\overline{l}$ is not a decision)

[0054] 7. if there is some conflict and A contains no decisions, then halt with output 'no solution'

[0055] 8. go to step 2.

[0056] It is rather obvious that this procedure performs an exhaustive systematic search over all possible assignments. The key issues are its efficient implementation, that is, a) data structures and b) heuristics for guiding the search: which variables to decide on first and how to prune the search space.

[0057] Indeed, modern extensions of the DPLL method include efficient data structures for propagation as disclosed in U.S. Pat. No. 7,418,369 and for clause learning, at each conflict, a new clause C can be added (learned), such that instead of backtrack one can do a backjump step, replacing $A_1\, l\, A_2$ by $A_1\, l'$ where C propagates l' from A_1. A single backjump step can undo several decisions as l needs not be the rightmost decision in A.

[0058] Pioneering work on clause learning was given by Marques-Silva and Sakallah in [3]. Analysis of the most frequently used learning scheme, the 1-UIP one, was done by Moskewicz, et. al. [4]. Propagations by 1-UIP learned clauses prune the search space very effectively. Such SAT-solving techniques are nowadays called conflict-driven clause learning (CDCL).

[0059] There have been several attempts to carry over CDCL from SAT to ILP. Then, clauses become constraints, literals become bounds (constraints with a single variable, that can be written as lower bounds $a \le x$ or upper bounds $x \le a$), and propagation becomes bound propagation.

[0060] An important problem for applying CDCL in ILP is the following rounding problem. Assume having the two constraints $1x+5y \le 5$ and $1x-5y \le 0$ and taking the decision $1 \le x$. Then from the first constraint $y \le 4/5$, can be inferred, which is rounded, causing a bound propagation of the new bound $y \le 0$, which, together with $1 \le x$ causes a conflict with the second constraint. Now a cut inference, eliminating y generates the new learned 1-UIP constraint $2x \le 5$. But unfortunately, unlike what happens in SAT, it is too weak to force a backjump. This problem is due to the rounding that takes place when propagating y.

[0061] In [5] the rounding problem is solved by limiting the kind of decisions that are allowed. This makes it possible, at each conflict caused by propagations with rounding, to compute so-called tightly propagating constraints that justify the same propagations without rounding. Drawbacks for performance are the complexity of computing the tightly propagating constraints, the limited kind of decisions and that the learned constraints are very different from the 1-UIP ones.

[0062] This invention proposes another method to overcome the rounding problem. It permits arbitrary decisions and guide the search analogously to the 1-UIP approach in SAT. Consider again the two constraints C_1: $1x+5y\leq5$ and C_2: $1x-5y\leq0$. After taking the decision $1\leq x$, the constraint C_2 propagates $1\leq y$ and C_1 propagates $y\leq0$ (obtaining a conflicting pair of bounds). Now along with each propagated bound, it is not only remembered which constraint caused its propagation, but also the set of bounds that caused it. For example, the bound $y\leq0$ has the associated reason set $\{1\leq x\}$ and reason constraint C_1. Similarly, along with $1\leq y$ the reason constraint C_2 and the reason set $\{1\leq x\}$ is stored. If a conflicting pair of bounds appears, a conflict analysis is done.

[0063] First, the conflicting pair is stored in the so called CSS (here, $\{1\leq y, y\leq0\}$). Along the process this CSS always contains a set of bounds that is inconsistent together with the constraints. Similarly to the CDCL SAT solvers' conflict analysis (but with bounds instead of literals) in the CSS the most recently propagated bound it is repeatedly replaced by its reason set. Here, after the first step, the CSS becomes $\{1\leq x, y\leq0\}$. After a finite number of such replacements, one always reaches a CSS that justifies a backjump. Here, after the second replacement (replacing $y\leq0$ by $1\leq x$), the CSS becomes $\{1\leq x\}$, inferring that $1\leq x$ alone is also conflicting, so one can backjump to before the first decision and assert the negation of $1\leq x$, that is, $x\leq0$. In our method this conflict analysis process in addition guides a sequence of cut inferences between the reason constraints of the bounds that are being replaced, and the finally resulting constraint can be learned. This backjumping method can always be applied, even if the learned constraint obtained using the cuts is too weak, due to the rounding problem, to justify this backjump.

[0064] The idea of applying conflicting sets is remotely reminiscent to the learning techniques with literals from SAT Modulo Theories [6, 7]. A less related document, only for rational arithmetic, is [8].

LIST OF CITED REFERENCES

[0065] [1] M. Davis and H. Putnam, 'A Computing Procedure for Quantification Theory', Journal of the ACM, 7:201-215, 1960.

[0066] [2] M. Davis, G. Logemann and D. Loveland, 'A Machine Program for Theorem-Proving', Communications of the ACM, vol. 5, No. 7, pp. 394-397, 1962).

[0067] [3] Marques-Silva and Sakallah, 'GRASP: A Search Algorithm for Propositional Satisfiability', IEEE Transactions on Computers, 1063-6757, 220-227, 1996.

[0068] [4] Moskewicz, et. al. 'Chaff: Engineering an Efficient SAT Solver", Jun. 18-22, 2001 DAC 2001 pp. 530-535.

[0069] [5] Jovanovic de Moura, 'Cutting to the Chase—Solving Linear Integer Arithmetic", J. Autom. Reasoning 51(1): 79-108 (2013).

[0070] [6] Nieuwenhuis et. al. 'Solving SAT and SAT Modulo Theories: From an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T)', Journal of the ACM, 53(6), 937-977, 2006.

[0071] [7] Moura and Bjorner, 'Satisfiability Modulo Theories: Introduction and Applications": Commun. ACM 54(9): 69-77 (2011).

[0072] [8] Korovin and Voronkov, 'Solving Systems of Linear Inequalities by Bound Propagation', CADE 2011: 369-383.

[0073] [9] Robert Nieuwenhuis, 'The IntSat Method for Integer Linear Programming', Springer International Publishing, LNCS 8656, pp. 574-589, 2014.

SUMMARY OF THE INVENTION

[0074] The invention proposes a computer-implemented method for solving sets S of linear arithmetic constraints modelling physical systems for deciding whether a given IP has any solution, and in the positive case finding one or more solutions. The invention comprises a number of data structures and algorithms, based on bound propagation and cuts that make a backtracking-based search procedure efficient and useful.

[0075] In a characteristic manner, the computer-implemented method automatically performs the following steps using a processor unit:

- **[0076]** 1a) feeding the set of linear arithmetic constraints S to the processor unit;
- **[0077]** 1b) creating a standard stack data structure B that is initially empty; said data structure containing a set of bounds and supporting the standard stack operations; said stack data structure B is stored in the processor and is being modified by considering the set of linear arithmetic constraints S by the subsequent steps;
- **[0078]** 1c) if there is a linear arithmetic constraint C in S and a set of bounds R in B such that C and R propagate a bound b that is new in B, then pushing b on top of the stack B, and associating to b the set R as its reason set and the linear constraint C as its reason constraint; and iterating this pushing and associating while possible;
- **[0079]** 1d) treating four non-overlapping cases:
  - **[0080]** 1d1) if there is no conflicting pair of bounds in B and if, for all i in {i . . . n} the variable x_i is defined in B to a value a_i, then halt outputting the solution Sol such that $Sol(x_i)=a_i$ for each i in i . . . n;
  - **[0081]** 1d2) if there is no conflicting pair of bounds in B and at least one variable is not defined in B, then a bound d is pushed on top of B such that d is new in B and d is not conflicting with any other bound in B, said bound d being called a decision;
  - **[0082]** 1d3) if there is at least one conflicting pair of bounds b_1 and b_2 in B such that there is no decision in B below b, nor below b_2 then halt outputting "no solution";
  - **[0083]** 1d4) if there is at least one conflicting pair of bounds b_1 and b_2 in B such that there is at least one decision located in B below b_1 or below b_2 then perform a conflict analysis based on the current stack B and as a consequence of which firstly a number of topmost elements of the stack B are popped and after that a new bound with an associated reason set and reason constraint is pushed on top of the stack and a new linear constraint is learned;
- **[0084]** 1e) return to step 1c).

[0085] In accordance with one embodiment, the conflict analysis further uses following two data structures:

[0086] a Conflicting Subset and

[0087] a Conflicting Constraint,

and the proposed method includes the following automatic actions:

[0088] 2a) if the conflicting pair of bounds is $\{b_1, b_2\}$ such that b_2 is located in the stack B above b_1, then initializing the CSS to $\{b_1, b_2\}$ and initializing the CC to the reason constraint of b_2;

[0089] 2b) if b is the bound in the CSS that is located in the stack B closest to the top of B, then

[0090] 2b1) popping bounds from the top of stack B until there are no decisions above b in B;

[0091] 2b2) removing b from the CSS and inserting into the CSS the reason set associated with b; and

[0092] 2b3) performing a cut between the current CC and the reason constraint of b, in such a way that the variable of b is eliminated, thus obtaining a new CC; and if no such a cut exists, then the CC remains unchanged;

[0093] 2c) if after popping k bounds including at least one decision from the stack B there is a set of bounds R in B such that CC and R propagate a bound b that is new in B, then popping k bounds from the stack B and pushing b on top of B with associated reason set R and reason constraint CC, and halting the conflict analysis;

[0094] 2d) if the CSS contains more than one bound that is located in B up or above the topmost decision of B then going to step 2b);

[0095] 2e) if the CSS contains exactly one bound b that is located in B up or above the topmost decision of B, then:

[0096] 2e1) if the CSS contains b as its unique element, then popping bounds from the stack B until B contains no decisions and after that pushing on the stack a new bound being the negation of b with an associated empty reason set and the final CC as its reason constraint; then this CC is learned,

[0097] 2e2) if the CSS contains more than one bound, then if b_1 is the bound of the CSS different from b such that b_1 is located in the stack B closest to the top of B, above exactly k decisions, then popping bounds from the stack B until B contains exactly k decisions and after that pushing on the stack a new bound being the negation of b, having this new bound as its associated reason set the result of removing b from the CSS, and the final CC as its reason constraint, and then learning this CC.

[0098] The step 2c) is optional and can be omitted.

[0099] To be noted that, in particular, the first time step 2b) is performed no such a cut exists since in that case CC and the reason constraint of b are the same linear constraint.

[0100] In one embodiment in step 2d), even if the CSS contains zero or one bound located in B up or above the topmost decision of B, then also going to step 2b).

[0101] In another embodiment after each application of step 2b), bounds of the form $a \leq x$ are eliminated from the CSS whenever a bound $a' \leq x$ with $a' > a$ is in the CSS and bounds of the form $x \leq a$ are eliminated from the CSS whenever a bound $x \leq a'$ with $a' < a$ is in the CSS.

[0102] In an alternative approach in step 2b) instead of the reason set of b, a set of bounds R is computed and inserted in the CSS, with all elements of R being located below b in B and the reason constraint of b and R also propagating b.

[0103] In an alternative approach in the step 1c) the reason set is not associated to b nor stored and in step 2b) a set of bounds R is computed and inserted in the CSS, with all elements of R being located below b in B and the reason constraint of b and R also propagating b.

[0104] In accordance with an embodiment, in step 1c) the linear arithmetic constraint C is not associated to b and in step 1d4) the conflict analysis is performed omitting steps 2b3) and 2c) involving the CC, and no new constraint is learnt.

[0105] In accordance with an embodiment, in step 1c) the iteration is performed non-exhaustively.

[0106] In accordance with an embodiment, the linear arithmetic constraints further include expressions of the form $a_1x_1 + \ldots + a_nx_n \geq a_0$, or $a_1x_1 + \ldots + a_nx_n = a_0$, or $a_1x_1 + \ldots + a_nx > a_0$, or $a_1x_1 + \ldots + a_nx_n < a_0$ or combinations thereof, where the coefficients $a_0 \ldots a_n$ can be arbitrary rational numbers, sets of which are all expressible by sets S of linear constraints of the form $b_1x_1 + \ldots + b_nx_n \leq b_0$, with integer coefficients $b_0 \ldots b_n$ so that the resulting set of constraints S has the same set of solutions.

[0107] Concerning the use for optimization of the method detailed in the previous embodiment, in order to find a solution Sol that minimizes the value of $a_1.\mathrm{Sol}(x_1) + \ldots + a_n.\mathrm{Sol}(x_n)$ for a given expression $a_1x_1 + \ldots + a_nx_n$, in a first iteration applying the method, and in successive iterations, while new solutions are found, applying the method with an additional constraint $a_1x_1 + \ldots + a_nx_n \leq a_0$ where a_0 is $a_1.\mathrm{Sol}(x_1) + \ldots + a_1.\mathrm{Sol}(x_n) - 1$ where Sol is the solution found in the previous iteration.

[0108] Further in order to find a solution Sol that maximizes the value of $a_1.\mathrm{Sol}(x_1) + \ldots + a_n \cdot \mathrm{Sol}(x_n)$ for a given expression $a_1x_1 + \ldots + a_nx_n$, applying the previous embodiment minimizing $-a_1x_1 + \ldots + -a_nx_n$.

[0109] According to another embodiment in step 2b) instead of popping and replacing the topmost bound b from the CSS another bound is popped and replaced by its reason set.

[0110] According to another embodiment and in order to solve Mixed Integer Programs (MIPs), that is, to find a solution where a given subset I of the variables must take integer values and the remaining variables can take arbitrary rational values, it is proposed to use an arbitrary LP solver for finding a solution RSol minimizing the value of an expression $a_1x_1 + \ldots + a_nx_n$, where in RSol all variables are allowed to take rational values, outputting RSol as a solution and halting if RSol(x) is an integer for every variable x of I, and if no such a solution RSol exists, generating an infeasible subset using the LP solver and taking as CSS the subset of bounds of the infeasible subset, and taking as CC any other constraint of the infeasible subset, and continuing the conflict analysis with step 2b).

[0111] Finally, in yet another embodiment the coefficients of the linear constraints are rational or floating point numbers. In this case, in step 1 d4) the conflict analysis is performed using cuts where c and d are positive rational or floating point numbers.

EXAMPLES

Example 1

[0112] This example involves the embodiment without reason constraints, without cuts and without learning new constraints. In this example, and in the following one, when a bound b in the stack B has exactly k decisions at or below it in B then b is said to belong to decision level (dl) k

[0113] Consider the following two constraints:

$$1x+1y+3z\leq 5$$

$$-1x-1y\leq -11$$

[0114] In addition, there are six one-variable constraints stating that all three variables are between −10 and 10. Note that these six constraints propagate the first six bounds with empty reason sets. Below the stack is shown (depicted here growing downwards) after propagating the initial constraints, and taking and propagating three decisions:

bound	reason set
−10 ≤ x	{ }
x ≤ 10	{ }
−10 ≤ y	{ }
Y ≤ 10	{ }
−10 ≤ z	{ }
z ≤ 10	{ }
1 ≤ x	{y ≤ 10}
1 ≤ y	{x ≤ 10}
z ≤ 1	{1 ≤ x, 1 ≤ y}
7 ≤ y	decision
z ≤ −1	{1 ≤ x, 7 ≤ y}
x ≤ 5	decision
−1 ≤ z	decision
x ≤ 1	{7 ≤ y, −1 ≤ z}
10 ≤ y	{x ≤ 1}
x ≤ −2	{10 ≤ y, −1 ≤ z}

[0115] Now there is a conflict with initial CSS {1≤x, x≤−2}.

[0116] In the first conflict analysis step, x≤−2 is removed from the CSS and its reason set {10≤y, −1≤z}, inserted obtaining {1≤x, −1≤z, 10≤y}. Since this CSS contains more than one bound of the highest decision level (dl 3), 10≤y is also replaced by its reason, getting {1≤x, −1≤z, x≤1)}. Since there are still two bounds of dl 3, x≤1 is also replaced getting {1≤x, 7≤y, −1≤z}. After this, the conflict analysis process terminates, since this final CSS contains only one bound of dl 3, which in this case is the last decision itself, −1≤z.

[0117] The negation of −1≤z is z≤−2, which is added to dl 1 (the dl of 7≤y) with reason set {1≤x, 7≤y}. Altogether, a backjump is done to:

bound	reason set
−10 ≤ x	{ }
x ≤ 10	{ }
−10 ≤ y	{ }
y ≤ 10	{ }
−10 ≤ z	{ }
z ≤ 10	{ }
1 ≤ x	{y ≤ 10}
1 ≤ y	{x ≤ 10}
z ≤ 1	{1 ≤ x, 1 ≤ y}
7 ≤ y	decision
z ≤ −1	{1 ≤ x, 7 ≤ y}
z ≤ −2	{1 ≤ x, 7 ≤ y}

[0118] After one more propagation and two further decisions and their propagations, the following stack is obtained:

bound	reason set
−10 ≤ x	{ }
x ≤ 10	{ }
−10 ≤ y	{ }
y ≤ 10	{ }
−10 ≤ z	{ }
z ≤ 10	{ }
1 ≤ x	{y ≤ 10}
1 ≤ y	{x ≤ 10}
z ≤ 1	{1 ≤ x, 1 ≤ y}
7 ≤ y	decision
z ≤ −1	{1 ≤ x, 7 ≤ y}
z ≤ −2	{1 ≤ x, 7 ≤ y}
−2 ≤ z	decision
x ≤ 4	{7 ≤ y, −2 ≤ z}
4 ≤ x	decision
y ≤ 7	{4 ≤ x, −2 ≤ z}
z ≤ −2	{4 ≤ x, 7 ≤ y}

[0119] It gives the solution where x=4, y=7 and z=−2, since all variables are fully defined to these values.

Example 2

[0120] Consider the following three constraints:

$$C_0:+1x-3y-3z\leq 1$$

$$C_1:-2x+3y+2z\leq -2$$

$$C_2:+3x-3y+2z\leq -1$$

and the stack (depicted here growing downwards) with some initial bounds coming from one-variable constraints, and taking and propagating two decisions:

bound	reason set	reason constraint
−2 ≤ x	{ }	
x ≤ 3	{ }	
1 ≤ y	{ }	
y ≤ 4	{ }	
−2 ≤ z	{ }	
z ≤ 2	{ }	
1 ≤ x	{1 ≤ y, −2 ≤ z}	C_1
y ≤ 2	{x ≤ 3, −2 ≤ z}	C_1
z ≤ 0	{x ≤ 3, 1 ≤ y}	C_1
x ≤ 2	decision	
z ≤ −1	{x ≤ 2, 1 ≤ y}	C_1
z ≤ −2	decision	
x ≤ 1	{y ≤ 2, z ≤ −2}	C_0
2 ≤ y	{1 ≤ x, z ≤ −2}	C_0
2 ≤ x	{2 ≤ y, −2 ≤ z}	C_1

[0121] Now there is a conflict with initial CSS {x≤1, 2≤x}. In the first conflict analysis step, 2≤x is removed from the CSS and its reason set {2≤y, −2≤z} inserted, obtaining the CSS {−2≤z, x≤1, 2≤y}, with two bounds of this decision level (dl 2).

[0122] In the second conflict analysis step, 2≤y is replaced by its reason set {1≤x, z≤−2} obtaining the new CSS {−2≤z, 1≤x, z≤−2, x≤1} which does not allow yet to backjump since it still contains two bounds of dl 2. But now a cut is attempted between the initial CC, which is C_1, and the reason constraint of 2≤y, which is C_0, in such a way that y is eliminated. Here this cut exists, with c=d=1, and the new constraint C_3:

[0123] $-1x-1z\leq-1$ is obtained and learned. This new constraint allows one to backjump to dl 1, since there it propagates $2\leq x$. At that point, after three more propagations,

bound	reason set	reason constraint
$-2 \leq x$	{ }	
$x \leq 3$	{ }	
$1 \leq y$	{ }	
$y \leq 4$	{ }	
$-2 \leq z$	{ }	
$z \leq 2$	{ }	
$1 \leq x$	$\{1 \leq y, -2 \leq z\}$	C_1
$y \leq 2$	$\{x \leq 3, -2 \leq z\}$	C_1
$z \leq 0$	$\{x \leq 3, 1 \leq y\}$	C_1
$x \leq 2$	decision	
$z \leq -1$	$\{x \leq 2, 1 \leq y\}$	C_1
$2 \leq x$	$\{z \leq -1\}$	C_3
$-1 \leq z$	$\{x \leq 2\}$	C_3
$2 \leq y$	$\{2 \leq x, z \leq -1\}$	C_0
$2 \leq x$	$\{2 \leq y, -1 \leq z\}$	C_1

another conflict exists with CSS $\{x\leq2, 3\leq x\}$, which after the first step becomes $\{x\leq2, -1\leq z, 2\leq y\}$, all three of this decision level (dl 1). After the second step (replacing $2\leq y$) the CSS becomes $\{x\leq2, z\leq-1, 2\leq x, -1\leq z\}$, all in dl 1. As before, the performed cut between C_1 and C_0 (the initial CC and the reason constraint of $2\leq y$) eliminates the variable y, obtaining $-1x-1z\leq-1$.

[0124] After the third step (replacing $-1\leq z$), the CSS becomes $\{x\leq2, z\leq-1, 2\leq x\}$, all in dl 1. The CC does not change because no cut eliminating z exists with C_3.

[0125] After the 4th step (replacing $2\leq x$), the CSS becomes $\{x\leq2, z\leq-1\}$, both in dl 1. Again the CC does not change because no cut eliminating z exists with C_3.

[0126] After the 5th step (replacing $z\leq-1$), the CSS becomes $\{1\leq y, x\leq2\}$, with only one literal of dl 1. The backjump with this CSS can take us to the dl of $1\leq y$ (dl 0) and add there the negation of $x\leq2$, which is $3\leq x$.

[0127] The result of the cut on CC with C_1 eliminating z gives us $-4x+3y\leq-4$. The backjump with this cut can also take us to the dl of $1\leq y$ (dl 0), propagating $2\leq x$. Since this is weaker than the bound $3\leq x$ obtained from the CSS, here the CSS one has been chosen. After two more propagations, the procedure returns 'no solution' since the conflicting pair of literals $y\leq2$ and $3\leq y$ appear at dl 0:

bound	reason set	reason constraint
$-2 \leq x$	{ }	
$x \leq 3$	{ }	
$1 \leq y$	{ }	
$y \leq 4$	{ }	
$-2 \leq z$	{ }	
$z \leq 2$	{ }	
$1 \leq x$	$\{1 \leq y, -2 \leq z\}$	C_1
$y \leq 2$	$\{x \leq 3, -2 \leq z\}$	C_1
$z \leq 0$	$\{x \leq 3, 1 \leq y\}$	C_1
$3 \leq x$	$\{1 \leq y\}$	C_4
$-1 \leq z$	$\{3 \leq x, y \leq 2\}$	C_0
$3 \leq y$	$\{3 \leq x, -1 \leq z\}$	C_0

Data Structures and Algorithms

[0128] The method proposed in this invention heavily relies on the efficiency of its implementation, for which new data structures and algorithms are given.

[0129] There is an array, the Bounds Array, indexed by variable number, that can return in constant time the current upper and lower bounds for that variable. Property 1: It always stores, for each variable x_i, the positions pl_i and pu_i in the stack of its current (strongest) upper bound and lower bound, respectively, with $pl_i=0$ ($pu_i=0$) if x_i has no current lower (upper) bound.

[0130] The data structure for the stack is another array containing at each position three data fields: a bound, a natural number pos, and an info field containing, among other information, the reason set and the reason constraint. Property 2: The value pos is always the position in the stack of the previous bound of the same type (lower or upper) for this variable, with pos=0 for initial bounds.

[0131] When pushing a new bound on the stack, and when popping a bound from the stack (during backjumping), it is easy to maintain properties 1 and 2 in constant time.

TABLE 1

Bounds array		
	Height in stack of current bound	
	Lower:	upper
x_1	1	2
x_2	0	0
	.	.
	.	.
	.	.
x_7	40	31
	.	.
	.	.
	.	.

TABLE 2

Stack			
1	$0 \leq x_1$	0	info
2	$x_1 \leq 8$	0	info
		.	
		.	
		.	
13	$0 \leq x_7$	0	info
14	$x_7 \leq 9$	0	info
		.	
		.	
		.	
23	$2 \leq x_7$	13	info
		.	
		.	
		.	
31	$x_7 \leq 6$	14	info
		.	
		.	
		.	
40	$5 \leq x_7$	23	info
		.	
		.	
		.	

[0132] Another important data structure allows for efficient bound propagation. For each variable x, there are two occurs lists. The positive occurs list for x contains all pairs (I_C, a) s.t. C is a linear constraint where x occurs with positive coefficient a. The negative occurs list contains the same for occurrences with a negative coefficient a. Here I_C is an index to the constraint header of C in the array of constraint headers. Each constraint header contains an integer F_C called a filter, and a (pointer to) the constraint C itself. The filter F_C is maintained cheaply, in such a way that C can only propagate if $F_C>0$, thus avoiding many useless (cache-) expensive inspections of the actual constraint C. This is done as follows.

[0133] Let C be a constraint of the form $a_1x_1+ \ldots +a_nx_n \leq a_0$. Let $l_i \leq x_i$ and $x_i \leq u_i$ be the current lower and upper bounds (if any) for x_i. Each expression a_ix_i in C can have a minimal value m_i, which is $a_i \cdot l_i$ if $a_i>=0$, and $a_i \cdot u_i$ otherwise.

[0134] Here m_i is undefined if there is no such bound l_i (or u_i). Initially, if some m_i is undefined, then F_C is set to a special value undefined and otherwise to $-a_0+m_1+ \ldots +m_n+\max_i$, $\{\text{abs}(a_i \cdot (u_i-l_i))\}$ where max and abs denote the maximum and absolute value functions, respectively. After that, F_C is said to be precise: the constraint C propagates if and only if, undefined$\neq F_C>0$. Property 3: At all timepoints, F_C=undefined or F_C is an upper approximation of the precise one.

[0135] To preserve property 3, these filters need to be updated when new bounds are pushed onto the stack, and need to be restored when backjumping.

[0136] Let a new lower bound $k \leq x$ be pushed onto the stack. Let the previous lower bound for x (if any) be $k' \leq x$. For each pair (I_C, a) in the positive occurs list of x, using I_C, access is done to the F_C and increase it by abs $(a \cdot (k-k'))$. If there was no previous lower bound, then F_C was undefined and is now set to 1. If F_C becomes positive, the constraint C is visited because it may propagate some new bound. After each time a constraint C is visited, F_C is set to its precise value.

[0137] Let a new upper bound $x \leq k$ be pushed on the stack. Then exactly the same is done, where $x \leq k'$ is the previous upper bound for x (if any), and using the negative occurs list. In order to be able to restore the filters when backjumping, each time an F_C value is increased by an amount d, a pair (F_C, d) is pushed onto a filter backtrack stack, and when it is moved from undefined to 1 a pair $(F_C$, undefined) is pushed.

[0138] For each decision that is taken, i.e., when pushing the i+1th decision on the stack, in a separate data structure a natural number h_i is recorded, h_i being the height of the filter backtrack stack before taking decision i+1. Then, when backjumping to a stack with k decisions, each pair (F_C, d) in the filter backtrack stack above height h_k is popped, and its F_C is decreased by d if d≠undefined, and restored to undefined if d=undefined.

[0139] Following a particular application case in which the proposed method is applied will be explained.

[0140] A steel factory needs to plan its next week, 168 hours in which it has to carry out N tasks (orders).

[0141] Each task i in 1 . . . N has a duration of d_i consecutive (whole) hours and requires, during all its d_i hours of activity, the exclusive use of one or more units of R different resources. For example, a certain task may occupy two mechanics, one operator, three cranes and two trucks. If during a certain hour several tasks are active simultaneously, they cannot share the resources they use.

[0142] For each resource j and each hour h in 1 . . . 168, let the integer $\text{used}_{j,h}$ denote the total number of units of resource j used during hour h and let $\text{peak}_j=\max(\text{used}_{j,1}, \ldots \text{used}_{j,168})$ be the (integer) peak usage of resource j during the week. The problem is to schedule all tasks, i.e., for each task i determine a starting time, while minimizing the total resource cost $C=c_1 \cdot \text{peak\_}_1 \leq, + \ldots +c_R \cdot \text{peak}_R$, where each c_j is a cost associated to resource j.

[0143] In practice the planned period of course needs not be 168 time units, and typically there are many more constraints, usually involving logical relationships, such as precedences between (groups of) tasks, temporary unavailability of resources, earliest or latest starting times for tasks, storage capacities for intermediate products, etc.

[0144] This problem is well known to have a large impact on costs and benefits in industry. Finding good solutions can be extremely hard.

[0145] A standard notation for it uses two sets of N·168 binary variables $\text{starts}_{i,h}$ ("task i starts on hour h") and $\text{isactive}_{i,h}$ ("task i is active on hour h") for all i and h.

[0146] It uses constraints expressing that $\text{starts}_{i,h}$ implies $\text{isactive}_{i,h}$ (i.e., $-\text{starts}_{i,h}+\text{isactive}_{i,h} \geq 0$) and also $\text{starts}_{i,h}$ implies $\text{isactive}_{i,h+1}$, etc., for its whole duration d_i. Also, each task i starts exactly once: $\text{starts}_{i,1}+ \ldots +\text{starts}_{i,168}=1$. In addition, for each resource j and hour h, we have $\text{used}_{j,h}=\text{units\_needed}_{1,j} \cdot \text{isactive}_{1,h}+ \ldots +\text{units\_needed}_{N,j}-\text{isactive}_{N,h}$, where $\text{units\_needed}_{i,j}$ is the number of units of j needed by task i. Finally, for each resource j there are 168 constraints $\text{peak}_j \geq \text{used}_{j,1} \ldots \text{peak}_j \geq \text{used}_{j,168}$.

[0147] A state-of-the-art methodology is the one used in MIP solvers, by solving a collection of LP relaxations of the given constraints, i.e., temporarily "forgetting" that certain variables must take integer values. Rational (possibly non-integer) solutions are sought for by means of (variants of) simplex or interior point methods. Such "forgetful" or "blind" intermediate non-integer solutions may be meaningless for our problem, since they may say, e.g., that a certain binary variable $\text{active}_{i,h}$ is 0.7, or that 3.54 units of resource j are used during some hour h.

[0148] Additional steps are then performed to turn these intermediate solutions into an integer one. For example, branch-and-cut methods maintain a tree of subproblems pending to be explored, with their rational solutions. Given a leaf node with variable x getting a non-integer solution 3.54, one can branch, i.e., split the problem into two subproblems, one with $x \leq 3$ and one with $x \geq 4$, or compute and add a new constraint (by a cut), precluding the non-integer solution 3.54 for x.

[0149] Such MIP solvers are well known to perform extremely poorly on pure SAT problems, where all variables are binary and constraints are (purely logical) clauses, and where conflict-driven clause-learning (CDCL) techniques are vastly superior and hence the method of choice in practice.

[0150] Here it is claimed a similar superiority of this invention, also a SAT-like method, for this industrial scheduling problem, which also has many binary variables and other relatively small-domain integer ones, as well as an essentially logical constraint structure as in pure SAT.

[0151] The proposed method explores the search space as CDCL does in SAT: by taking decisions (in our case each decision, as stated in step 1d2, is a heuristically guided guess of an upper or lower bound for a given variable) and efficiently inferring and adding the implied information by propagating these decisions. And, again as in CDCL, when a conflict appears (i.e., two contradictory bounds), a conflict analysis procedure allows one to backjump to an earlier search state, enrich it by an additional bound and its propagations, and possibly learning a new constraint.

[0152] Together with [5], the proposed method is the first of this nature able to handle integer variables and constraints, but with the advantage over [5] that in the proposed method arbitrary decisions can be taken. Indeed it performs much better than [5], as revealed by the experiments of the publication [9] (which also reports superiority over the main commercial MIP solvers on other ILP problems from the well-known standard MIPLIB).

[0153] As it happens in methods for SAT (cf. section "Background of the invention"), it is rather obvious that the proposed method performs a systematic search over the possible solutions. This involves only trivial mathematics. Again as for SAT, the key aspects of the proposed method are the engineering of novel data structures (propagation record, propagation stack, reason set, reason constraint) and the novel heuristics for guiding the search and for learning new constraints. Such techniques for SAT are disclosed in U.S. Pat. No. 7,418,369.

[0154] Applied to the industrial scheduling problem, the proposed method uses heuristics for taking good decisions (typically, as in SAT, on variables involved in many recent conflicts). For example, it will guess an upper bound on the total resource cost C or on some of the individual variables $peak_h$. Meaningful new bounds will then be propagated (unlike what happens in LP relaxations; present propagation takes into account that variables must be integer). Similarly, meaningful strong new constraints are quickly learned, stating that certain combinations of tasks cannot take place simultaneously, etc.

[0155] These newly learned constraints again strengthen the propagation power and prevent future similar conflicts.

[0156] Being able to take arbitrary decisions is essential (step 1d2 of claim **1**) to the proposed method. In [5] one can only decide a given variable to be equal to its current upper bound or to its current lower bound; in practice, one needs to guess it to belong to, say, the lower (or upper) halve of the interval between its current upper and lower bounds, an idea akin to binary search.

[0157] While preferred embodiments of the invention have been shown and described herein, it will be understood that such embodiments are provided by way of example only. Numerous variations, changes and substitutions will occur to those skilled in the art without departing from the spirit of the invention. Accordingly, it is intended that the appended claims cover all such variations as fall within the spirit and scope of the invention.

1. A computer-implemented method for solving sets of linear arithmetic constraints modelling physical systems, the method comprising using a computer processor unit performing a programmed execution of mathematical operations wherein, being $\{x_1 \ldots x_n\}$ a set of variables, said linear arithmetic constraints are expressions of the form $a_1x_1 + \ldots + a_nx_n \leq a_0$, in which the coefficients $a_0 \ldots a_n$ are integer numbers,

wherein a solution for a set of linear arithmetic constraints S is an expression Sol mapping each variable x of $\{x_1 \ldots x_n\}$ to an integer value Sol(x) such that all constraints are satisfied, that is, for each constraint of the form $a_1x_1 + \ldots + a_nx_n \leq a_0$ in S, the integer number $a_1.\mathrm{Sol}(x_1) + \ldots + a_n \cdot \mathrm{Sol}(x_n)$ is smaller than or equal to a_0;

wherein the following notions/terms are used:

bound (constraint with a single variable x, that can be written as lower bounds $a \leq x$ or upper bounds $x \leq a$) where a is an integer number

the negation of a lower bound $a \leq x$ is the upper bound $x \leq a-1$ and the negation of an upper bound $x \leq a$ is the lower bound $a+1 \leq x$;

a lower bound $a_1 \leq x$ and an upper bound $x \leq a_2$ are called conflicting if $a_1 > a_2$;

a lower bound $a \leq x$ is called new in a given set of bounds B if there is no lower bound $a' \leq x$ in B with $a' \geq a$, and an upper bound $x \leq a$ is called new in a given set of bounds B if there is no upper bound $x \leq a'$ in B with $a' \leq a$, and a variable x is called defined to the value a in a given set of bounds B, if B contains the bounds $a \leq x$, and $x \leq a$; and

a monomial is an expression of the form a x, where a is an integer or a rational number and x is a variable; it called negative if a is negative and positive otherwise;

propagation:

If C is a linear arithmetic constraint of the form $a_1x_1 + \ldots + a_nx_n \leq a_0$ where:

the subset of positive monomials of $\{a_1x_1 + \ldots + a_nx_n\}$ is $\{ax, c_1y_1, \ldots, c_py_p\}$;

the subset of negative monomials of $\{a_1x_1 + \ldots + a_nx_n\}$ is $\{d_1z_1, \ldots, d_qz_q\}$;

R is a set of bounds $\{l_1 \leq y_1, \ldots, l_p \leq y_p, z_1 \leq u_1, \ldots, z_q \leq u_q\}$;

u is the largest integer such that $u \leq (a_0 - c_1l_1 - \ldots - c_pl_p - d_1u_1 - \ldots - d_qu_q)/a$, then C and R propagate the upper bound $x \leq u$;

If C is a linear arithmetic constraint of the form $a_1x_1 + \ldots + a_nx_n \leq a_0$ where:

the subset of positive monomials of $a_1x_1 + \ldots + a_nx_n$ is $\{c_1y_1, \ldots, c_py_p\}$;

the subset of negative monomials of $a_1x_1 + \ldots + a_nx_n$ is $\{ax, d_1z_1, \ldots, d_qz_q\}$;

R is a set of bounds $\{l_1 \leq y_1, \ldots, l_p \leq y_p, z_1 \leq u_1, \ldots, z_q \leq u_q\}$;

l is the smallest integer such that $l \geq (a_0 - c_1l_1 - \ldots - c_pl_p - d_1u_1 - \ldots - d_qu_q)/a$, then C and R propagate the lower bound $l \leq x$,

a propagation record is a triple (b, R, C) where b is a bound C is a linear arithmetic constraint and R is a set of bounds such that C and R propagate b, then R being termed the reason set of b and C being termed the reason constraint of b; in a special kind of propagation record called a decision, the components R and C are null,

a propagation stack is a data structure having capabilities of a standard stack data structure whose elements are propagation records, with standard operations for pushing and popping elements and for inspecting the topmost element and in addition the nonstandard capability of inspecting non-topmost elements;

a bound b is said to be in a propagation stack B if b is the first element of some propagation record of B; similarly a set of bounds R is said to be in a propagation stack B if R is a subset of the set of all first elements of the propagation records of B,

a constraint C is said to be learned when it is added to the set of linear arithmetic constraints S;

wherein said programmed execution of mathematical operations of the method being automatically performed, by the following steps:

1a) receiving by the computer processor unit the set of linear arithmetic constraints S;

1b) creating using the computer processor unit a propagation stack B that is initially empty; being said propagation stack B stored in the computer processor unit and being automatically modified using the computer processor unit by considering the set of linear arithmetic constraints S by implementing the subsequent steps;

1c) if there is a linear arithmetic constraint C in S and a set of bounds R in B such that C and R propagate a bound b that is new in B, then pushing the propagation record (b, R, C) on top of the stack B; and iterating this pushing while possible;

1d) treating four non-overlapping cases using the computer processor unit:

1d1) if there is no conflicting pair of bounds in B and if, for all i in {i . . . n} the variable x_i is defined in B to a value a_i, then halt outputting the solution Sol such that $Sol(x_i)=a_i$ for each i in i . . . n;

1d2) if there is no conflicting pair of bounds in B and at least one variable is not defined in B, then a propagation record (d, –, –) is pushed on top of B such that d is new in B and d is not conflicting with any other bound in B, said bound d and the propagation record (d, –, –) being termed a decision, and then return to step 1c);

1d3) if there is at least one conflicting pair of bounds b_1 and b_2 in B such that there is no decision in B below b_1 nor below b_2 then halt outputting "no solution";

1d4) if there is at least one conflicting pair of bounds b_1 and b_2 in B such that there is at least one decision located in B below b_1 or below b_2 then a conflict analysis is performed based on the current propagation stack B and as a consequence of which firstly a number of topmost elements of the propagation stack B are popped and after that a new bound with an associated reason set and reason constraint is pushed on top of the stack and a new linear constraint C is learned, and then return to step 1c).

2. The method of claim **1** wherein said conflict analysis further uses a data structure called the CSS, Conflicting Subset, a set data structure storing a subset of the bounds of B, and another data structure called the CC, Conflicting Constraint, wherein the following notions are used:

if C_1 is a linear arithmetic constraint $a_1x_1+ \ldots +a_nx_n \leq a_0$, and C_2 is a linear arithmetic constraint $b_1x_1+ \ldots +b_nx_n \leq b_0$, then a cut between C_1 and C_2 is a linear arithmetic constraint $c_1x_1+ \ldots +c_nx_n \leq c_0$ such that c and d are positive natural numbers and $c_i=c \cdot a_i+d \cdot b_i$ for each i in 0 . . . n; and

if $c_j=0$ for some j in 1 . . . n then this cut is said to eliminate the variable x_j, the method comprising the following steps:

2a) if the conflicting pair of bounds is {b_1, b_2} such that b_2 is located in the stack B above b_1, then initializing the CSS to {b_1, b_2} and initializing the CC to the reason constraint of b_2;

2b) if b is the bound in the CSS that is located in the stack B closest to the top of B, then

2b1) popping bounds from the top of stack B until there are no decisions above b in B;

2b2) removing b from the CSS and inserting into the CSS the reason set associated with b;

2b3) performing a cut between the current CC and the reason constraint of b, in such a way that the variable of b is eliminated, thus obtaining a new CC; and if no such a cut exists, then the CC remains unchanged;

2c) if after popping k bounds including at least one decision from the stack B there is a set of bounds R in B such that CC and R propagate a bound b that is new in B, then popping k bounds from the stack B and pushing b on top of B with associated reason set R and reason constraint CC, and halting the conflict analysis;

2d) if the CSS contains more than one bound that is located in B at the height of the topmost decision of B or above then going to step 2b);

2e) if the CSS contains exactly one bound b that is located in B at the height of the topmost decision of B or above:

2e1) if the CSS contains b as its unique element, then popping bounds from the stack B until B contains no decisions and after that pushing on the stack a new bound being the negation of b with an associated empty reason set and the final CC as its reason constraint; then this CC is learned, and

2e2) if the CSS contains more than one bound, then if b_1 is the bound of the CSS different from b such that b_1 is located in the stack B closest to the top of B, above exactly k decisions, then popping bounds from the stack B until B contains exactly k decisions and after that pushing on the stack a new bound being the negation of b, having this new bound as its associated reason set the result of removing b from the CSS, and the final CC as its reason constraint, and then learning this CC.

3. The method of claim **1** wherein said conflict analysis further uses a data structure called the CSS, Conflicting Subset, a set data structure storing a subset of the bounds of B, and another data structure called the CC, Conflicting Constraint, wherein the following notions are used:

if C_1 is a linear arithmetic constraint $a_1x_1+ \ldots +a_nx_n \leq a_0$, and C_2 is a linear arithmetic constraint $b_1x_1+ \ldots +b_nx_n \leq b_0$, then a cut between C_1 and C_2 is a linear arithmetic constraint $c_1x_1+ \ldots +c_nx_n \leq c_0$ such that c and d are positive natural numbers and $c_i=c \cdot a_i+d \cdot b_i$ for each i in 0 . . . n; and

if $c_j=0$ for some j in 1 . . . n then this cut is said to eliminate the variable x_j, the method comprising the following steps:

3a) if the conflicting pair of bounds is {b_1, b_2} such that b_2 is located in the stack B above b_1, then initializing the CSS to {b_1, b_2} and initializing the CC to the reason constraint of b_2;

3b) if b is the bound in the CSS that is located in the stack B closest to the top of B, then

3b1) popping bounds from the top of stack B until there are no decisions above b in B;

3b2) removing b from the CSS and inserting into the CSS the reason set associated with b;

3b3) performing a cut between the current CC and the reason constraint of b, in such a way that the variable of b is eliminated, thus obtaining a new CC; and if no such a cut exists, then the CC remains unchanged;

3c) if the CSS contains more than one bound that is located in B at the height of the topmost decision of B or above then going to step 3b);

3d) if the CSS contains exactly one bound b that is located in B at the height of the topmost decision of B or above:

3d1) if the CSS contains b as its unique element, then popping bounds from the stack B until B contains no decisions and after that pushing on the stack a new bound being the negation of b with an associated empty reason set and the final CC as its reason constraint; then this CC is learned, and

3d2) if the CSS contains more than one bound, then if b_1 is the bound of the CSS different from b such that b_1 is located in the stack B closest to the top of B, above exactly k decisions, then popping bounds from the stack B until B contains exactly k decisions and after that pushing on the stack a new bound being the negation of b, having this new bound as its associated reason set the result of removing b from the CSS, and the final CC as its reason constraint, and then learning this CC.

4. The method of claim **2**, wherein in step 2d), even if the CSS contains zero or one bound located in B above a decision, being the topmost decision of B, then going to step 2b).

5. The method of claim **2**, wherein after each application of step 2b) bounds of the form $a \leq x$ are eliminated from the CSS whenever a bound $a' \leq x$ with $a' > a$ is in the CSS and bounds of the form $x \leq a$ are eliminated from the CSS whenever a bound $x \leq a'$ with $a' < a$ is in the CSS.

6. The method of claim **2**, wherein in step 2b) instead of the reason set of b, a set of bounds R is computed and inserted in the CSS, with all elements of R being located below b in B and the reason constraint of b and R also propagating b.

7. The method of claim **2** wherein in step 1c) the reason set is not associated to b nor stored and in step 2b) a set of bounds R is computed and inserted in the CSS, with all elements of R being located below b in B and the reason constraint of b and R also propagating b.

8. The method of claim **1** wherein in step 1c) the linear constraint C is not associated to b and wherein in step 1d4) the conflict analysis is performed omitting steps 2b3) and 2c) involving the CC, and no new constraint is learnt.

9. The method of claim **1** wherein in step 1c) the iteration is performed non-exhaustively.

10. The method of claim **1** wherein the linear arithmetic constraints further include expressions of the form $a_1x_1 + \ldots + a_nx_n \geq a_0$, or $a_1x_1 + \ldots + a_nx_n = a_0$, or $a_1x_1 + \ldots + a_nx_n > a_0$, or $a_1x_1 + \ldots + a_nx_n < a_0$ or combinations thereof, where the coefficients $a_0 \ldots a_n$ can be arbitrary rational numbers, sets of which are all expressible by sets S of linear constraints of the form $b_1x_1 + \ldots + b_nx_n \leq b_0$, with integer coefficients $b_0 \ldots b_n$ so that the resulting set of constraints S has the same set of solutions

11. The method of claim **10** further comprising in order to find a solution Sol that minimizes the value of $a_1.\mathrm{Sol}(x_1) + \ldots + a_n \cdot \mathrm{Sol}(x_n)$ for a given expression $a_1x_1 + \ldots + a_nx_n$, in a first iteration applying steps 1a) to 1e) of the method, and in successive iterations, while new solutions are found, applying steps 1a) to 1e) of the method with an additional constraint $a_1x_1 + \ldots + a_nx_n \leq a_0$ where a_0 is $a_1.\mathrm{Sol}(x_1) + \ldots + a_1 \cdot \mathrm{Sol}(x_n) - 1$ and Sol is the solution found in the previous iteration.

12. The method of claim **10** further comprising in order to find a solution Sol that maximizes the value of $a_1.\mathrm{Sol}(x_1) + \ldots + a_n \cdot \mathrm{Sol}(x_n)$ for a given expression $a_1x_1 + \ldots + a_nx_n$, in a first iteration applying steps 1a) to 1e) of the method, and in successive iterations, while new solutions are found, applying steps 1a) to 1e) of the method with an additional constraint $-a_1x_1 - \ldots - a_nx_n \leq a_0$ where a_0 is $-a_1.\mathrm{Sol}(x_1) - \ldots - a_n \cdot \mathrm{Sol}(x_n) - 1$ where each time Sol is the solution found in the previous iteration.

13. The method of claim **2**, wherein in step 2b) instead of popping and replacing the topmost bound b from the CSS another bound is popped and replaced by its reason set.

14. The method of claim **2** wherein in order to solve Mixed Integer Programs (MIPs), that is, to find a solution where a given subset I of the variables must take integer values and the remaining variables can take arbitrary rational values, it is proposed to use an arbitrary LP solver for finding a solution RSol minimizing the value of an expression $a_1x_1 + \ldots + a_nx_n$, where in RSol all variables are allowed to take rational values, outputting RSol as a solution and halting if RSol(x) is an integer for every variable x of I, and if no such a solution RSol exists, generating an infeasible subset using the LP solver and starting a conflict analysis.

15. The method of claim **14**, wherein to perform said conflict analysis a data structure called the CSS, Conflicting Subset, a set data structure storing a subset of the bounds of B, and another data structure called the CC, Conflicting Constraint are used, wherein the following notions are used:

if C_1 is a linear arithmetic constraint $a_1x_1 + \ldots + a_nx_n \leq a_0$, and C_2 is a linear arithmetic constraint $b_1x_1 + \ldots + b_nx_n \leq b_0$, then a cut between C_1 and C_2 is a linear arithmetic constraint $c_1x_1 + \ldots + c_nx_n \leq c_0$ such that c and d are positive natural numbers and $ci = c \cdot a_i + d \cdot b_i$ for each i in $0 \ldots n$; and

if $c_j = 0$ for some j in $1 \ldots n$ then this cut is said to eliminate the variable x_j, the method comprising the following steps:

15a) initializing the CSS to the subset of bounds of the infeasible subset, and initializing the CC to any other constraint of the infeasible subset;

15b) if b is the bound in the CSS that is located in the stack B closest to the top of B, then

15b1) popping bounds from the top of stack B until there are no decisions above b in B;

15b2) removing b from the CSS and inserting into the CSS the reason set associated with b;

15b3) performing a cut between the current CC and the reason constraint of b, in such a way that the variable of b is eliminated, thus obtaining a new CC; and if no such a cut exists, then the CC remains unchanged;

15c) if after popping k bounds including at least one decision from the stack B there is a set of bounds R in B such that CC and R propagate a bound b that is new in B, then popping k bounds including at least one decision from the stack B and pushing b on top of B with associated reason set R and reason constraint CC, and halting the conflict analysis;

15d) if the CSS contains more than one bound that is located in B up or above the topmost decision of B then going to step 15b);

15e) if the CSS contains exactly one bound b that is located in B up or above the topmost decision of B:

15e1) if the CSS contains b as its unique element, then popping bounds from the stack B until B contains no decisions and after that pushing on the stack a new bound being the negation of b with an associated empty reason set and the final CC as its reason constraint; then this CC is learned, and

15e2) if the CSS contains more than one bound, then if b_1 is the bound of the CSS different from b such that b_1 is located in the stack B closest to the top of B, above exactly k decisions, then popping bounds from the stack B until B contains exactly k decisions and after that pushing on the stack a new bound being the negation of b, having this new bound as its associated reason set the result of removing b from the CSS, and the final CC as its reason constraint, and then learning this CC.

16. The method of claim **14**, wherein to perform said conflict analysis a data structure called the CSS, Conflicting Subset, a set data structure storing a subset of the bounds of B, and another data structure called the CC, Conflicting Constraint are used, wherein the following notions are used:

if C_1 is a linear arithmetic constraint $a_1x_1+ \ldots +a_nx_n \leq a_0$, and C_2 is a linear arithmetic constraint $b_1x_1+ \ldots +b_nx_n \leq b_0$, then a cut between C_1 and C_2 is a linear arithmetic constraint $c_1x_1+ \ldots +c_nx_n \leq c_0$ such that c and d are positive natural numbers and $ci=c \cdot a_i+d \cdot b_i$; for each i in 0 . . . n; and

if $c_j=0$ for some j in 1 . . . n then this cut is said to eliminate the variable x_j, the method comprising the following steps:

16a) initializing the CSS to the subset of bounds of the infeasible subset, and initializing the CC to any other constraint of the infeasible subset;

16b) if b is the bound in the CSS that is located in the stack B closest to the top of B, then

16b1) popping bounds from the top of stack B until there are no decisions above b in B;

16b2) removing b from the CSS and inserting into the CSS the reason set associated with b;

16b3) performing a cut between the current CC and the reason constraint of b, in such a way that the variable of b is eliminated, thus obtaining a new CC; and if no such a cut exists, then the CC remains unchanged;

16c) if the CSS contains more than one bound that is located in B up or above the topmost decision of B then going to step 16b);

16d) if the CSS contains exactly one bound b that is located in B up or above the topmost decision of B:

16d1) if the CSS contains b as its unique element, then popping bounds from the stack B until B contains no decisions and after that pushing on the stack a new bound being the negation of b with an associated empty reason set and the final CC as its reason constraint; then this CC is learned, and

16d2) if the CSS contains more than one bound, then if b_1 is the bound of the CSS different from b such that b_1 is located in the stack B closest to the top of B, above exactly k decisions, then popping bounds from the stack B until B contains exactly k decisions and after that pushing on the stack a new bound being the negation of b, having this new bound as its associated reason set the result of removing b from the CSS, and the final CC as its reason constraint, and then learning this CC.

17. The method of claim **1** wherein the coefficients of the linear constraints are rational or floating point numbers and wherein in step 1 d4) the conflict analysis is performed using cuts where c and d are positive rational or floating point numbers.

* * * * *