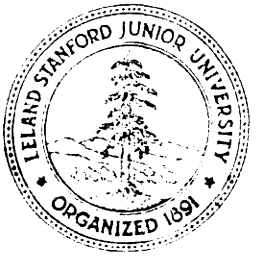
A POLYNOMIAL TIME ALGORITHM FOR SOLVING SYSTEMS OF LINEAR INEQUALITIES WITH TWO VARIABLES PER INEQUALITY

by

Bengt Aspvall and Yossi Shiloach

STAN-CS-79-703
January 1979

COMPUTER SCIENCE DEPARTMENT
School of Humanities and Sciences
STANFORD UNIVERSITY



A Polynomial Time Algorithm for Solving Systems of Linear Inequalities with Two Variables per Inequality

Bengt Aspvall and Yossi Shiloach

Computer Science Department
Stanford University
Stanford, California 94305

January, 1979

Abstract

We present a constructive algorithm for solving systems of linear inequalities (LI) with at most two variables per inequality. The algorithm is polynomial in the size of the input. The LI problem is of importance in complexity theory since it is polynomial time equivalent to linear programming. The subclass of LI treated in this paper is also of practical interest in mechanical verification systems, and we believe that the ideas presented can be extended to the general LI problem.

Keywords: linear inequalities, linear programming, polynomial time, algorithm, complexity, loop residue, hidden inequality.

This work was supported by National Science Foundation Grants MCS-75-22870 and MCS-77-23738, by Office *of* Naval Research Contract NOOO14-78-C-0330, and by a Chaim Weizmann Postdoctoral Fellowship, Reproduction in whole or in part is permitted for any purpose of the United States government.

1 INTRODUCTION

In this paper we give a polynomial time algorithm for solving systems of linear inequalities where each inequality contains at most two variables. We start this chapter by introducing the problem and relating it to other problems and previous results. In the second section we present the general approach to solving the problem and in the last section we define the representation and the complexity measure to be used throughout the paper.

1.1 Linear Inequalities

Given a rational $m \times n$ matrix A and a rational m-vector $\mathbf{c}$ the linear inequalities (LI) problem is to determine whether or not there exists an n-vector x of rational numbers such that

$$Ax \leq \mathbf{c}. \tag{1}$$

If such a vector x exists we say that the system is *satisfiable* and that x is a *feasible* vector, otherwise the system is *unsatisfiable* and no feasible vectors exist. If the system is satisfiable one may, or may not, ask for a feasible vector x. The algorithm presented in this paper does supply a feasible vector if the system is satisfiable.

The LI problem is of theoretical interest in complexity theory. It is well-known that the linear programming (LP) problem-where one wants to maximize a linear function subject to linear inequality constraints-is polynomial time (Turing) equivalent to the LI problem [GJ, I, RD]. Since the complexity of the LP problem is one of the foremost open problems, the complexity of LI is of the same interest.

We will use LI(k) to denote the class of LI problems with at most k variables per inequality. Any instance of the LI problem can be transformed into an equivalent LI(3) instance by introducing additional variables and constraints. (By using a binary encoding scheme the coefficients of the new problem can be restricted to $\{-1, 0, +1\}$ [I].) The transformations can be done with at most a polynomial increase in the number of variables and constraints. Thus if there exists an efficient algorithm for LI(3) there is also one for LP and vice versa, In this paper we will give a polynomial time algorithm for LI(2). Hence if the general LI problem is not polynomial time solvable the result shows that there must be an inherent difference in complexity between LI(2) and LI(3). The LI(2) problem has practical applications in, for example, mechanical verification systems [NO, P].

To denote a typical constraint of an LI(2) problem we will use

$$ax + by \leq c, \tag{2}$$

where x and y are any two variables and a, b and c are rational numbers. Vaughan Pratt [P] has given an $O(n^3)$ algorithm for the case where the inequalities are of the form $x - y \leq c$. Robert Shostak [S] has generalized Pratt's idea to the general LI(2) case, but his algorithm has an exponential worst-case behaviour. By using a modified Fourier-Motzkin elimination method, Greg Nelson [N] has given an $O(mn^{\lceil \log_2 n \rceil + 4} \log n)$ algorithm for LI(2).

1.2 Outline of the Algorithm

It is well-known that the solution space of a system of linear inequalities forms a convex polyhedron. Let $\mathfrak{X}(S)$ be the projection of the solution polyhedron on the x-axis for a given system S. We can also view $\mathfrak{X}(S)$ as the set of values of x for which a solution to the entire system can be constructed. If we can find $\mathfrak{X}(S)$, which is a convex interval on the x-axis, we can assign a value $\tilde{x} \in \mathfrak{X}(S)$ to the variable x. This reduces the number of variables by one and yields a new system of inequalities that is satisfiable if and only if S is satisfiable, Hence if S is satisfiable a solution can be constructed by recursively solving systems of linear inequalities with fewer variables.

Our algorithm is related to a theorem by Shostak. In [S] he shows how to construct an undirected graph from a given system of inequalities such that the system is unsatisfiable if and only if the graph has what he calls an *infeasible simple* loop. Since we use the same graph construction in our algorithm we will describe Shostak's ideas here.

Let S be the system of inequalities of the form (2) and let v_0 be an auxiliary zero *variable* that always occurs with zero coefficient-the only variable that can do that. Without loss of generality we can thus assume that all the inequalities contain two variables. We construct the graph G(S) $= (V, E)$ with $n + 1$ vertices and m edges as follows: (a) For each variable x occurring in S add a vertex named x to G(S). (We will use x to denote both the variable and the vertex when no confusion can occur.) (b) For each inequality $ax + by \leq c$ in S add an undirected edge between x and y to G(S) and label the edge with the inequality (Fig. 1).

Let P be a path of G(S) determined by the vertices $v_1, v_2, \ldots, v_{l+1}$ and the edges $e_1, e_2, \ldots, e_l$. We define the triple sequence of P as

$$\langle a_1, b_1, c_1 \rangle, \langle a_2, b_2, c_2 \rangle, \ldots, \langle a_l, b_l, c_l \rangle$$

where, for $1 \leq i \leq$ I, $a_i v_i + b_i v_{i+1} \leq c_i$ is the inequality associated with e_i. If a_{i+1} and b_i have opposite signs for $1 \leq i < l$ then P is called *admissible* and we

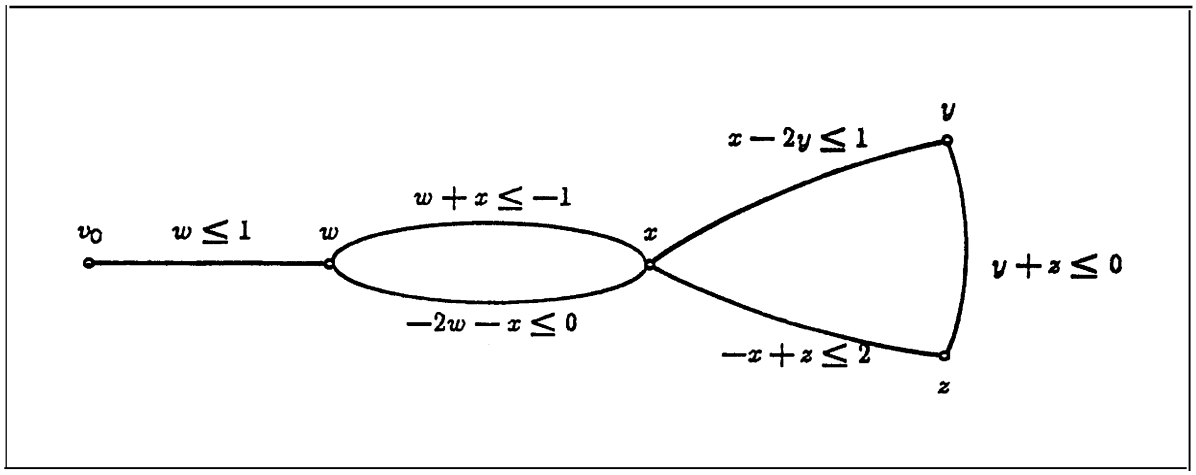


Figure 1. G(S) for
$S = \{ w \leq 1, w + x \leq -1, -2w - x \leq 0, x - 2y \leq 1, y + z \leq 0, -x + z \leq 2 \}$

define *(ap, bp, cp),* the residue of *P,* as

$$\langle a_P, b_P, c_P \rangle = \langle a_1, b_1, c_1 \rangle \odot \langle a_2, b_2, c_2 \rangle \odot \cdots \odot \langle a_l, b_l, c_l \rangle, \tag{3}$$

where $\odot$ is the associative binary operator defined on triples by

$$(a, b, c) \odot (a', b', c') = \langle kaa', -kbb', k(ca' - c'b) \rangle \text{ and } k = \frac{a'}{|a'|}. \tag{4}$$

Intuitively the operator $\odot$ takes two inequalities and derives a new inequality by eliminating a common variable, e.g. $ax + by \leq c$ and $a'y + b'z \leq c'$ imply $-aa'x + bb'z \leq -(ca' - c'b)$ if $a' < 0$ and $b > 0$. Note that the residue imposes a direction on P although the graph is undirected and that the signs of ap and a_1 agree as do the signs of bp and b_l. The significance of path residues is formalized in the following lemma.

Lemma **1** [S]. *If P is an admissible path with initial vertex x,* final vertex y, and residue (ap, *bp, cp),* then any *point* (i.e., assignment of rational values *to* variables) that satisfies the inequalities that label *the edges* of *P* satisfies $apx + bpy \leq cp$.

A path is called a ***loop*** if the initial and final vertices are identical. (A loop is not uniquely specified unless its initial vertex is given.) If all the intermediate vertices of a path are distinct the path is simple. The reverse of an admissible path is always admissible, and the cyclic permutations of a loop are admissible if and only if a_1 and b_l have opposite signs. It follows that no admissible loop with initial vertex v_0 is permutable.

An admissible loop P with initial vertex x is *infeasible* if $a_P + bp = 0$ and $cp < 0$, since by Lemma 1 any solution of S must satisfy the unsatisfiable loop inequality $(a_P + bp)x \leq c_P$. Thus if G(S) has an infeasible loop the system of inequalities S is unsatisfiable. However the converse is not true in general. We say that two systems of linear inequalities S and T are *equivalent* if they have the same solution polyhedron. Next we show how to extend S to an equivalent system S', such that $G(S')$ has an infeasible simple loop if and only if S is unsatisfiable.

For each vertex x of G(S) and for each admissible simple loop P of G(S) with $a_P + b_P \neq 0$ and with initial vertex x, add a new inequality $(a_P + b_P)x \leq c_P$ to S. We will call the new system S' the *Shostak extension* of S (since it was first introduced by Shostak [S]).

Theorem 1 [S]. *Let* S' be *the* Shostak *extension* of S. The *system* of inequalities S is satisfiable if and *only* if $G(S')$ has no infeasible simple loop.

This theorem can easily be used to design an algorithm for LI(2). However, since the number of simple cycles in a graph on n vertices can be exponential in n, the worst-case behaviour of the algorithm can be exponential in the number of variables. We will now outline the method we use in order to avoid examining all the cycles separately.

Assume that G(S) is a graph for S. For each variable x define

$$\begin{aligned} xmin &= \max\{ \frac{c_P}{b_P} \mid P \text{ is an admissible path from } v_0 \text{ to x in G(S) and } bp < 0 \} \\ xmax &= \min\{ \frac{c_P}{b_P} \mid P \text{ is an admissible path from } v_0 \text{ to x in G(S) and } bp > 0 \}, \end{aligned} \tag{5}$$

where we define $\max\{\ \} = -\infty$ and $\min\{\ \} = \infty$. Intuitively $x \geq$ xmin is the most restrictive lower bound on x that we can derive using a chain of inequalities in S, where all but the first have two variables; a similar inequality holds for $xmax$. Let $xmin^{(k)}$ and $xmax^{(k)}$ be defined in the same way as xmin and xmax but with P additionally restricted to length at most k. Thus $xmin^{(\infty)} =$ xmin and $xmax^{(\infty)} = xmax$.

Define a graph G(S) to be *closed* for S if [xmin, xmax] $= \mathfrak{X}(S)$ for all variables x of S. Given a system S of linear inequalities, let T be a system of linear inequalities with the same variables as S. We say that the system T is a closure of S if the following is true: (a) The two systems S and T are equivalent and (b) the graph $G(T)$ is closed for T. Thus if we can find a closure T of S in polynomial time and if there is way to compute [xmin, $xmax$] $= \mathfrak{X}(T) = \mathfrak{X}(S)$ from G(T) in polynomial time, we can construct a solution to S in polynomial time. In section 3.1 we show that the Shostak extension S' is a closure of S and

$[xmin^{(n)}, xmax^{(n)}] = [xmin, xmax]$ so $G(S')$ can in fact be used to reduce S to a smaller system. However, constructing $G(S')$ takes exponential time in the worst case.

We construct in polynomial time a modified extension S^* of S. The extension S^* will be a closure of S and we also have $[xmin^{(n)}, xmax^{(n)}] = [xmin, xmax]$. This enables us to compute $[xmin^{(n)}, xmax^{(n)}] = \mathfrak{B}(S^*) = \mathfrak{B}(S)$ from $G(S^*)$ in polynomial time.

In the construction of the Shostak extension S' from G(S) many redundant inequalities are added to S. These inequalities will not be added in the construction of S^*. Using a binary search technique we will compute from G(S) the non-redundant, i.e. the most restrictive, inequalities x $\geq$ slow and x $\leq xhigh$ for each variable x in polynomial time. In order to do this we keep, for each variable x of S, two intervals $[xlow\downarrow, xlow\uparrow]$ and $[xhigh\downarrow, xhigh\uparrow]$ such that either $xlow \in [xlow\downarrow,$ xlowf] or $xlow = -\infty$, and analogously for $xhigh$. Initially the intervals will be set to [—λ, X], where λ can be computed from the input.

The algorithm will guess values for the variables one at a time. A guessed value of x will be "pushed" through G(S) in a breadth-first manner. This will give new-but not necessarily true—upper and lower bounds on the variables. By analysing the outcome of each guess it is possible to chop the interval for $xlow$ or xhigh by at least half. There will also be a way to decide if a new and more restrictive bound on x can be derived or the intervals can be coalesced into single points. Thus after a finite-and in fact polynomial-number of guesses we will be able to determine the true values of $xlow$ and $xhigh$ for any variable x of S.

Chapter 2 is devoted to the computation of $xlow$ and $xhigh$. After computing the values of $xlow$ and xhigh for each variable x of S from the graph G(S), we construct the extension S^* from S by adding the inequalities x $\geq$ xlow and x $\leq xhigh$ for each variable x. Given $G(S^*)$ it is rather straightforward to compute $xmin^{(n)}$ and $xmax^{(n)}$ using a breadth-first search. This is explained in detail in chapter 3, where we give the LI(2)-Algorithm. In Chapter 4 we analyse the complexity of the algorithm and show that the number of guesses needed is bounded by a polynomial in the input size.

1.3 Representation and Complexity Measure

Our algorithm for LI(2) is polynomial in the size of the input. In order to establish exactly what this means we have to say a few words about the way the input is represented and how the complexity is measured.

We assume that an instance of LI(2) is described as a string of inequalities of the form $ax + by \leq c$. Each rational number is represented as an ordered

pair of integers and each variable by an integer between 1 and n. All integers are written in binary notation, and just enough additional symbols to delimit the input unambigously are allowed. The *input size* is the total length of the string describing a given instance.

Throughout this paper we will use a random access machine (RAM) model. (For a detailed description see [AHU].) The complexity measure will be the worst-case time using a uniform cost criterion, i.e. all elementary arithmetic operations and comparisons take one unit of time. However, we want to establish that the algorithm is polynomial also in the Turing machine sense. The following theorem relates the complexity for the two machine models.

Theorem 2 [AHU]. *The* random access machine *under* logarithmic *cost* and *the* Turing machine are *polynomially* related *models*.

In chapter 4, where we examine the complexity of the algorithm, we will therefore consider the logarithmic cost criterion as well. We will show that the intermediate results do not "blow up" establishing that the LI(2) problem is solvable in polynomial time on a Turing machine.

2 FINDING THE EXTENSION S^*

In this chapter we show how to find S* from G(S) using a binary search technique. We start by classifying different kinds of admissible loops and examining their behaviour under guesses. In the second section we describe how to push a guessed value for a variable through G(S) and in the last section we construct the extension S^* and the graph $G(S^*)$ using the results from the previous sections.

2.1 Behaviour of Loops under Guesses

From a linear inequality $ax + by \leq$ c we can derive an upper bound on y if $b > 0$ or a lower bound if $b < 0$ that depends on x if a $\neq 0$. Let us see how this can be employed in G(S). Let P be an admissible path from x to y (x, y $\neq v_0$) with residue $\langle a_P, bp,$ cp). If we assign a value to x, we can get a constant bound on y from $a_P x + bpy \leq cp$. The residue of P is uniquely defined since the operator $\odot$ is associative. Hence the bound on y is unique with respect to P. For another path P' we might get another bound on y. The trouble is that we do not want to compute the residues for all admissible paths between x and y.

Let us ignore this problem for the moment and turn our attention to admissible loops since they play a fundamental role in the algorithm. By making a guess

for a variable x at one end of an admissible loop we get a bound on the same variable at the other end of the loop. We now show how the guess and the result are related to the residue of the loop.

Let P be an admissible loop with residue (up, $b_P, c_P\rangle$ and initial and final vertex $x \neq v_0$. We call $a_P x + bps \leq c_P$ the hidden inequality of the loop. If $up + bp \neq 0$ *we* can-and often will-write the hidden inequality of P as either $x \leq h$ or $x \geq h$, where $h = c_P/(a_P + bp)$. Without knowing the residue of P *we* can obtain information about its hidden inequality by guessing a value g for x, pushing the guess around P (as described in the next section) and examining the result. There are four classes of admissible loops denoted $=$, $\doteq$, $\mp$, and $\pm$ and distinguished by the signs of up and bp (the upper sign corresponds to the sign of a_P). We will now classify their behaviour under guesses.

Let $h = cp/(ap + bp)$, $r = (cp - a_P g)/b_P$, and $\mu = -a_P/b_P$ so that r can be written as $r = \mu g + (1 - \mu)h$. The hidden inequality for a $\doteq$ loop is of the form $x \leq h$. If we guess the value g for x we get as the result the inequality $x \leq r$. The situation for $=$ loops is similar but with the inequalities in the opposite directions. For these loops $\mu < 0$ so either $g < h < r$, $r < h < g$, *or* $g = h = r$ and, since the guess and the result are on opposite sides of h, we will call them the flipping loops (Figs. 2 and 3).

Figure 2. Flipping $\doteq$ loops

Figure 3. Flipping $=$ loops

The $\mp$ loops are slightly more complicated. We always get a result of the form $x \leq r$ but we have to distinguish three different cases for the hidden inequality: (a) If $ap + bp > 0$ we have $x \leq h$ and $0 < \mu < 1$ so the result lies in the interval between g and h-the converging case (Fig. 4a). (b) If $a_P + b_P < 0$ *we* have $x \geq h$ and $\mu > 1$ so h and r are on opposite sides of g-the *diverging* case (Fig. 4b). (c) If $a_P + b_P = 0$ the hidden inequality is $0 \leq c_P$ and we get $r < g$ if $c_P < 0$, i.e. the hidden inequality is a contradiction, and $r \geq$ g otherwise-the contradiction and tautology cases (Fig. 4c).

Figure 4a. Converging case for $\mp$ loops

Figure 4b. Diverging case for $\mp$ loops

Figure 4c. Contradiction (left) and tautology cases for $\mp$ loops

The $\pm$ loops are similar to the $\mp$ loops in their behaviour-the difference is that the inequalities go in the opposite directions. Thus the result is always $x \geq r$ and again there are three cases for the hidden inequality (Figs. **5a,b,c).** For the the converging and diverging cases of the $\mp$ loops and $\pm$ loops we have ignored the possibility that $g = r$. If this happens we must have $h = g$ as for the $=$ loops and $\mp$ loops. Table 1 summarizes the loop results for reference in subsequent sections. Note that the directions of the hidden inequality and the resulting inequality disagree only for diverging loops.

Figure 5a. Converging case for $\pm$ loops

Figure 5b. Diverging case for $\pm$ loops

Figure 5c. Contradiction (left) and tautology cases for $\pm$ loops

Class of loop	Hidden ineq.	Result	Relations between g, r, and h	Figure
$\neq$	$x \leq h$	$x \leq r$	$g < h < r$, $g = h = r$, or $r < h < g$	2
$=$	$x \geq h$	$x \geq r$	$g < h < r$, $g = h = r$, or $r < h < g$	3
$\mp$, $a_P + b_P > 0$	$x \leq h$	$x \leq r$	$g < r < h$, $g = h = r$, or $h < r < g$	4 a
$\mp$, $a_P + b_P < 0$	$x \geq h$	x 5 7	$r < g < h$, $g = h = r$, or $h < g < r$	4b
$\mp$, $a_P + b_P = 0$	$0 \leq c_P$	$x \leq r$	$r < g$ or $g \leq r$	4c
$\pm$, $a_P + b_P < 0$	$x \geq h$	$x \geq r$	$g < r < h$, $g = h = r$, or $h < r < g$	5 a
$\pm$, $a_P + b_P > 0$	$x \leq h$	$x \geq r$	$r < g < h$, $g = h = r$, or $h < g < r$	5b
$\pm$, $a_P + b_P = 0$	$0 \leq c_P$	$x \geq r$	$g < r$ or $r \leq g$	5c

Table 1.

2.2 Pushing Guesses through G(S)

In the previous section we saw that we could obtain partial information about the hidden inequality of a given admissible loop by guessing a value for a variable, "pushing" it around the loop, and examining the result for the same variable. We now describe how to "push" the guessed value through G(S) in order to obtain the resulting inequality.

The hidden inequalities that we want to find are those that give the most restrictive lower and upper bounds for each variable. All other hidden inequalities are redundant and are therefore not added to S^*. The algorithm will find the non-redundant hidden inequalities after examining only a polynomial number of all the hidden inequalities in G(S).

Clearly we do not want to "push" a guessed value for x separately around each admissible loop with initial vertex x—not even around all the simple ones since there might be exponentially many. We will instead "push" the guessed value in a breadth-first way with at most n stages, This allows us to find a new and more restrictive hidden inequality involving x.

We call an edge e labeled ay + $bz \leq$ c a positive edge for y if a $>$ 0 and *negative* for y if a $<$ 0. Note that the same edge can be, for example, positive for y and negative for z. Given a lower bound on y we can derive a (lower or upper) bound on z using a positive edge for y. We say that the vertex y ***sends*** the lower bound on y over the edge e, the edge *transfers* this bound on y into a bound on z, which is then received by the vertex z. Similarly, a negative edge for y can transfer only an upper bound on y into a (lower or upper) bound on z.

A guessed value g for x will be spread like a rumor through G(S). Whenever a vertex y $\neq x$ receives a new and more restrictive lower (upper) bound the vertex y records it as the current *lower* (upper) bound on y and in the next stage of the algorithm y sends this new bound out over all its positive (negative) edges.

Algorithm 1 (The Grapevine). The input to the algorithm is the graph G(S) and a guessed value g for x. The algorithm finds the most restrictive lower and upper bounds on x that can be obtained from g using admissible loops of at most n edges with x occurring only as the initial and final vertex. The algorithm stores enough information that the loops corresponding to the most restrictive lower and upper bounds can be reconstructed if desired.

Step **1.** [Send guess from x.] Let $i \leftarrow 1$. Transfer the guessed value g over all edges incident to x. For each vertex y $\neq$ x record the most restrictive lower and upper bounds received on y, and record also the edges over which they were transferred together with the current stage number 1. (If the same bound was received over several edges record one of them.)

Step **2.** [Termination?] If $i < n$ set $i \leftarrow i + 1$ and go to Step 3. Otherwise the algorithm terminates and returns the current lower and upper bounds on x as the result.

Step **3.** [Stage i.] For each vertex y $\neq$ x do the following: (a) If the currently most restrictive lower bound on y was recorded in stage $i - 1$ send it *over* all its positive edges. (b) If the currently most restrictive upper bound on y was recorded in stage $i - 1$ send it over all its negative edges.

Step 4. [Record new bounds.] For each vertex y do the following: If a new, and more restrictive, lower (upper) bound on y was received during stage i record it as the current lower (upper) bound, and record also the edge that transferred the new bound together with the current stage number i. (If the same bound was received over several edges record one of them.) Go to Step 2. □

Later we need to trace the loop that gave the most restrictive lower or upper bound on x. This can be done simply by tracing the loop backwards. The algorithm stores for each vertex y the edges over which the lower and upper bounds were received, Since only a lower (upper) bound on y can have been sent out over a positive (negative) edge for y there is no ambiguity between lower and upper bounds when tracing the loop backwards.

We call an admissible loop P of length at most n, with x occurring only as the initial and final vertex, and with hidden inequality $(a_P + b_P)x \leq c_P$ a *lower* loop for x if $b_P < 0$ and an *upper* loop for x if $b_P > 0$. Thus a lower loop is either a $=$ loop or a $\pm$ loop and an upper loop is either a $\mp$ loop or a $\mp$ loop. Transferring a

guess around a lower loop for x gives a lower bound on x as the result. Similarly, by using an upper loop for x we get an upper bound on x. A lower (resp. an upper) loop for x is *optimal* with respect to g if for all other lower (resp. upper) loops P' for x we have $(c_{P'} - a_{P'}g)/b_{P'} \leq (c_P - a_P g)/b_P$ (resp. $(c_{P'} - a_{P'}g)/b_{P'} \geq (c_P - a_P g)/b_P$). Note that a simple admissible loop is either a lower or an upper loop for some vertex. The proof of the following lemma is straightforward and left to the reader.

Lemma 2. Algorithm 1 returns *no* **lower (upper)** bound *on x if no lower (upper) loop* **for** x *exists.* Otherwise *there exists* an *optimal lower (upper) loop P for x* with *respect tog* and *Algorithm 1 returns* $x \geq r$ $(x \leq r)$, **where** $r = (c_P - a_P g)/b_P$.

2.3 Constructing the Extension S* using Binary Search

In this section we show how to construct the extension S^* of S without explicitly examining all simple admissible loops as is done in the construction of S', the Shostak extension of S. In the construction of S' there might be exponentially many inequalities involving the variable x added to S. However all but at most two of these inequalities are redundant. We define

$$\begin{aligned} xlow &= \max\{\, h_P \mid P \text{ is a lower loop for x with hidden inequality } x \geq h_P \,\} \\ xhigh &= \min\{\, hp \mid P \text{ is a upper loop for x with hidden inequality } x \leq hp \,\}. \end{aligned} \tag{6}$$

Using a binary search technique and the results from the two previous sections we can compute the values of slow and *xhigh* for all vertices x without examining all admissible simple loops. To obtain the extension S^* we then add for each variable x the two inequalities $x \geq$ xlow and $x \leq xhigh$ to S.

It is now an easy task to construct $G(S^*)$ as follows: For each variable x add two edges between x and v_0 to G(S) and label the edges $x \geq xlow$ and $x \leq xhigh$ respectively. The graph $G(S^*)$ will be a subgraph of $G(S')$ in the following sense: (a) There is an edge between x and y (x, $y \neq v_0$) in $G(S^*)$ if and only if there is an edge between x and y in G(S') with the same label, (b) If e is an edge between x and v_0 in $G(S^*)$ with label $x \geq c$, then there exists an edge e' between x and v_0 in $G(S')$ with label $x \geq c'$, where $c \geq c'$. (c) If e is an edge between x and v_0 in $G(S^*)$ with label $x \leq c$, then there exists an edge e' between x and v_0 in $G(S')$ with label $x \leq c'$, where $c \leq c'$.

In order to find the values of slow and *xhigh* we keep, for each variable x of S, two intervals $[xlow\downarrow, xlow\uparrow]$ and $[xhigh\downarrow, xhigh\uparrow]$ such that either $xlow \in [xlow\downarrow, xlow\uparrow]$ or $xlow = -\infty$ and analogously for *xhigh*. Initially the intervals will be set to $[-\lambda, h]$, where λ can be computed from the input as explained in section 4.1,

The two intervals $[xlow\downarrow, xlow\uparrow]$ and $[xhigh\downarrow, xhigh\uparrow]$ will be identical for some number of steps of the binary search and we will use Algorithm 2 to chop their joint interval by at least half in each iteration. If the intervals become non-identical we switch to Algorithm 3 and continue the search. The intervals will either be identical or have at most one point in common. We will always take the midpoint of an interval as a guess and use Algorithm 1 to provide the most restrictive lower and upper bounds with respect to this guess.

Algorithm 2 *(Chop Joint Interval).* The input to the algorithm is G(S), a variable x, and an initial bound λ such that either $xlow \in [-\lambda, \lambda]$ or $xlow = -\infty$, and either $xhigh \in [-\lambda, \lambda]$ or $xhigh = \infty$. In each iteration the algorithm chops the joint interval for xlow and *xhigh* by at least half. The algorithm terminates when either an infeasible loop is found or the two intervals [xlow↓, xlowf] and *[xhighl, xhighf]* become non-identical. In the latter case xlowf = *xhighl*, *xlow* $\in [xlow\downarrow, xlowf]$ or xlow $= -\infty$, and *xhigh* $\in$ *[xhighl, xhigh↑]* or *xhigh* $= \infty$.

Step 1. [Initialize.] Let $xlow\downarrow \leftarrow -\lambda$ and $xhighf \leftarrow \lambda$.

Step 2. [Distribute guess.] Let g t $(xlow\downarrow + xhigh\uparrow)/2$. Use Algorithm 1 to find $x \geq r$ and $x \leq r'$—the most restrictive lower and upper bounds on x with respect to g.

Step 3. [Chop or split?] If $x \geq r$ and $r >$ g go to Step 4. Otherwise if $x \leq r'$ and $r' <$ g go to Step 5. Otherwise go to Step 6.

Step 4. [Use lower result.] Trace the loop giving the bound $x \geq r >$ g and compute its hidden inequality. If $x \geq h$ and $h > g$ let $xlow\downarrow \leftarrow h$ else if $x \leq h$ and $h < g$ let $xhigh\uparrow \leftarrow h$ else terminate the algorithm due to an infeasible loop. Go to Step 2.

Step 5. [Use upper result.) Trace the loop giving the bound $x \leq r' <$ g and compute its hidden inequality. If $x \leq h$ and $h < g$ let $xhigh\uparrow \leftarrow h$ else if $x \geq h$ and $h > g$ let $xlow\downarrow \leftarrow h$ else terminate the algorithm due to an infeasible loop. Go to Step 2.

Step 6. [Split interval and terminate.] Let xlowf ← $xhigh\downarrow \leftarrow$ g. Terminate the algorithm and return the two intervals [xlow↓, xlowf] and *[xhighl, xhighf]* as the result. Cl

The following lemma will be very important in the proofs of correctness for Algorithms 2 and 3.

Lemma *3. The following two statements* are equivalent:

(a) *Algorithm 1 returns either* $x \geq r > g$ *or* $x \leq r < g$.

(b) *In G(S) there exists* a *lower loop for x with hidden* inequality $x \geq h > g$, *or an* upper *loop for* x with *hidden* inequality $x \leq h < g$, *or an infeasible loop of length* at most *n* with initial *vertex x.*

Proof. From the behaviour of different classes of loops (Table 1) and Lemma 2 it is easy to see that (a) implies (b). We will therefore only show that (b) implies (a). If there exists a lower loop P for x in G(S) with hidden inequality $x \geq h > g$ we know from Table 1 that we can derive either $x \geq r > g$ or $x \leq r < g$ from P. Since Algorithm 1 finds an optimal lower loop P' either $x \geq r' \geq r > g$ or $x \leq r' \leq r < g$ must be returned. The proof for an upper loop for x is similar. If there exists an infeasible loop P of length at most n and with initial vertex x we know, from the behaviour of the contradiction case for $\mp$ loops and $\pm$ loops, that either $x \geq r > g$ or $x \leq r < g$ can be derived from P. Since P is either a lower or an upper loop for x, Algorithm 1 must return either $x \geq r' \geq r > g$ or $x \leq r' \leq r < g$ by Lemma 2. □

Theorem 3. *In* each *iteration Algorithm* 1 *chops the* joint *interval for xlow and xhigh by at least* half. Algorithm 1 *terminates either* because an infeasible loop has *been found or the two intervals* $[xlow\downarrow, xlow\uparrow]$ and $[xhigh\downarrow, xhigh\uparrow]$ have *become non-identical. In* the latter case $xlow\uparrow = xhigh\downarrow$, $xlow \in [xlow\downarrow, xlowf]$ *or* $xlow = -\infty$, and $xhigh \in [xhigh\downarrow, xhigh\uparrow]$ *or* $xhigh = \infty$.

Proof. It is easy to see that each time the algorithm returns to Step 2 the interval $[xlow\downarrow, xhigh\uparrow]$ has been chop by at least half, Furthermore, the new endpoint corresponds to a bound on x that has been obtained from a new hidden inequality. Since the algorithm has to go to Step 2 in each iteration and since there are only finitely many lower and upper loops of length at most n in G(S) the algorithm must terminate. Clearly if the algorithm terminates in Step 6 the two intervals for slow and *xhigh* have only one point in common.

Let us now show that if the algorithm terminates in Step 4 an infeasible loop has been found, Suppose the algorithm terminates in Step 4 without having found an infeasible loop. The loop that gave the result $x \geq r > g$ must then have either $x \geq h > g$ or $x \leq h < g$ as its hidden inequality according to Lemma 3. But in this case we do not terminate the algorithm so we have a contradiction. The proof for termination in Step 5 is similar.

Finally we show that if the algorithm terminates in Step 6 then either $xlow \in [xlow\downarrow, xlow\uparrow]$ or $xlow = -\infty$. Suppose to the contrary that $slow \notin [xlow\downarrow, xlowf]$ and $slow \neq -\infty$. Then either $-\infty < xlow < xlow\downarrow$ or $xlow > xlow\uparrow = g$, where g is the last guess used in Step 2. By assumption $xlow \geq xlow\downarrow$ or $xlow = -\infty$ after Step 1. The only statements that change $xlow\downarrow$ are $xlow\downarrow \leftarrow h$ in Steps 4 and 5, and at those points we know that $x \geq h$. Thus the first case cannot happen. If $xlow > xlow\uparrow = g$ there must be a lower loop with hidden inequality $x \geq h > g$. By Lemma 3 we see that Algorithm 1 must return either $x \leq r < g$ or $x \geq r > g$. But in this case we do not go to Step 6 from Step 3 so we have a

contradiction. Hence we conclude that xlow $\in [xlow\downarrow$, xlowf] or $xlow = -\infty$. We omit the corresponding proof for $xhigh$ since it is analogous to the one for $xlow$. □

By using Algorithm 2 we can compute an interval $[xlow\downarrow, xlow\uparrow]$ such that $xlow \in [xlow\downarrow$, xlowf] or $xlow = -\infty$, and $xhigh \geq xlow\uparrow$. We now show how to use this result to compute the correct value of xlow with an iterative technique similar to the one used in Algorithm 2. The main difference is the termination criterion. Guessing $xlow\downarrow$, the left endpoint of the interval for $xlow$, allows us to determine if there exists a lower loop with hidden inequality $x \geq h > xlow\downarrow$. If this is not the case we can coalesce the interval $[xlow\downarrow$, xlowf] into the single point $xlow\downarrow$.

Algorithm 3 (Chop Lower Interval). The input to the algorithm is G(S), a variable x, and an interval $[xlow\downarrow$, xlowf] such that xlow $\in [xlow\downarrow$, xlowf] or $xlow = -00$. Furthermore $xlow\uparrow \leq xhigh$ is assumed. In each iteration the algorithm chops the interval $[xlow\downarrow, xlow\uparrow]$ by at least half. The algorithm terminates if either an infeasible loop is found or xlow is determined to be either $xlow = xlow\downarrow$ or xlow $= -\infty$. In the latter case the point $xlow\downarrow$ is returned.

Step 1. [Chop or coalesce interval?] Let g $\leftarrow xlow\downarrow$. Use Algorithm 1 to find $x \geq r$ and $x \leq r'$-the most restrictive lower and upper bounds on x with respect to g. If $x \geq r$ and $r >$ g, or $x \leq r'$ and $r' <$ g go to Step 3.

Step 2. [Coalesce interval and terminate.] Terminate the algorithm and return $xlow\downarrow$ as the result.

Step 3. [New guess.] Let g $\leftarrow (xlow\downarrow + xlow\uparrow)/2$. Use Algorithm 1 to find $x \geq r$ and $x \leq r'$—the most restrictive lower and upper bounds on x with respect to g.

Step 4. [Which end to chop?] If $x \geq r$ and $r >$ g go to Step 5. Otherwise if $x \leq r'$ and $r' <$ g go to Step 6. Otherwise set xlowf $\leftarrow$ g and go to Step 3.

Step 5. [Use lower result.] Trace the loop giving the bound $x \geq r > g$ and compute its hidden inequality. If $x \geq h$ and $h >$ g let $xlow\downarrow \leftarrow h$ and go to Step 1. Otherwise terminate the algorithm due to an infeasible loop.

Step 6. fuse upper result.] Trace the loop giving the bound $x \leq r' <$ g and compute its hidden inequality. If $x \geq h$ and $h >$ g let $xlow\downarrow \leftarrow h$ and go to Step 1. Otherwise terminate the algorithm due to an infeasible loop. □

Lemma 4. *Algorithm 3 terminates; furthermore in* each *iteration the interval* $[xlow\downarrow, xlow\uparrow]$ *is chopped by* at least *half.*

Proof. The test in Step 1 guarantees, according to Lemma 3 and the assumption that $xhigh \geq xlow\uparrow$, that whenever we get to Step 3 there exists either a lower

loop for x with hidden inequality $x \geq h > xlow\downarrow$ or an infeasible loop of length at most n with initial vertex x. If there is an infeasible loop or h is in the right half of $[xlow\downarrow$, xlowf] this will be detected in Step 4, according to Lemma 3, and the algorithm will proceed to Step 5 or 6. Otherwise the right half of $[xlow\downarrow, xlow\uparrow]$ will be chopped and the algorithm will return to Step 3. Thus h must fall in the right half of $[xlow\downarrow$, xlowf] after finitely many executions of Steps 3 and 4 and the algorithm will proceed to Step 5 or 6.

To show that Algorithm 3 terminates it remains to be shown that it cannot return to Step 1 infinitely many times. It is easy to see that each time the algorithm returns to Step 1 the interval $[xlow\downarrow, xlow\uparrow]$ has been chopped by at least half. Furthermore the new left endpoint corresponds to a bound on x that has been obtained from a new hidden inequality, Since there are only finitely many lower loops in G(S) the algorithm must terminate, □

Theorem 4. *If there exist any* infeasible loops *of length* at *most n* with initial *vertex x in G(S) Algorithm 3 finds one and terminates. Otherwise the* algorithm *terminates* and returns $xlow\downarrow$ such that *either* $xlow = slow)$ *or* $xlow = -\infty$.

Proof. From Lemma 4 we know that Algorithm 3 terminates. Let us first show that if the algorithm terminates in Step 5 an infeasible loop has been found. Suppose the algorithm terminates in Step 5 without having found an infeasible loop. The loop that gave the result $x \geq r > g$ must then have either $x \leq h <$ g or $x \geq h > g$ as its hidden inequality according to Lemma 3. By assumption xlowf $\leq xhigh$ at the beginning of the algorithm and xlowf is never increased so $x \leq h <$ g cannot be the case. If $x \geq h >$ g is the case in Step 5 we do not terminate the algorithm. Hence we have a contradiction. Termination in Step 6 is handled similarly.

We now show that if there exists an infeasible loop of length at most n with initial vertex x the algorithm terminates in Step 5 or 6. Suppose to the contrary that the algorithm terminates in Step 2 despite the fact that such a loop exists and let g be the last guess used in Step 1. Lemma 3 tells us that Algorithm 1 must return either $x \geq r >$ g or $x \leq r <$ g in Step 1. But if this is the case we do not go to Step 2. Hence we have a contradiction.

It remains to be shown that if the algorithm terminates in Step 2 either $slow = xlow\downarrow$ or $slow = -\infty$. In order to prove this we need the following claim, which is proved below.

- Whenever the algorithm gets to Step 1 we have either $xlow \in [xlow\downarrow, xlow\uparrow]$ or $slow = -\infty$.

Let us assume that the algorithm terminates in Step 2 but neither $xlow = xlow\downarrow$ nor $slow = -\infty$ is true. From the claim above we conclude that $xlow > xlow\downarrow =$ g, where g is the last guess used in Step 1. Thus there must be a lower loop P for

x with hidden inequality $x \geq h > g$. Algorithm 1 would therefore, according to Lemma 3, return either $x < r < g$ or $x \geq r > g$ in Step 1. But in this case we do not go to Step 2 so we have a contradiction.

Proof of claim: By assumption the claim is true the first time the algorithm reaches Step 1. Suppose the claim is violated for the first time when the algorithm returns to Step 1 the i^{th} time. Then either $-\infty < slow < xlow\downarrow$ or $xlow > xlowf$. The only statements that change $xlow\downarrow$ are $xlow\downarrow \leftarrow h$ in Steps 5 and 6. At those points we know that $x \geq h$ so the first case cannot have happened. Thus $xlow > xlowf$ which implies that there exists a lower loop P for x with hidden inequality $x \geq h > xlowf$. Hence xlowf must have been erroneously changed since the previous execution of Step 1. The only statement that changes $xlow\uparrow$ is $xlowf \leftarrow g$ in Step 4, where g is the last guess used in Step 3. Due to the existence of a lower loop P with hidden inequality $x \geq h > xlow\uparrow = g$ Algorithm 1 must, according to Lemma 3, return either $x \leq r < g$ or $x \geq r > g$ as the result in Step 3. But then we would not change xlowf in Step 4 so we have a contradiction. Cl

After using Algorithm 3 we know that either $xlow = xlow\downarrow$ or $slow = -\infty$, but we do not know which alternative is the true one. However, if λ is sufficiently large, so that initially $xlow = -\infty$ or $slow \in (-\lambda, \lambda)$, then $xlow = -\infty$ if and only if $xlow\downarrow = \lambda$.

We now know how to compute the correct value of slow. The algorithm for computing $xhigh$ is almost identical to Algorithm 3 and is therefore omitted. Given the values of $xlow$ and $xhigh$ for each variable x of S we can construct the extension S^* and the graph $G(S^*)$ as described at the beginning of this section.

3 THE LI(2)-ALGORITHM

The LI(2)-Algorithm is given and proved to be correct in this chapter. We start by showing that the Shostak extension S' is a closure of S. In the first section we also show that $[xmin^{(n)}, xmax^{(n)}] = [xmin, xmax]$ for $G(S')$. In the second section we use these results to show that the extension S^* is also a closure of S and that $[xmin^{(n)}, xmax^{(n)}] =$ [xmin, $xmax$] holds for $G(S^*)$ too. This allows us to present the polynomial algorithm for LI(2).

3.1 The Shostak extension S' is a closure of S

In this section we show that S' is a closure of S and that $[xmin^{(n)}, xmax^{(n)}] = \mathfrak{X}(S') = \mathfrak{X}(S)$ for $G(S')$. If an infeasible simple loop in G(S) is found during the construction of S' we know that S is unsatisfiable by Theorem 1 and we therefore define $[xmin^{(n)}, xmax^{(n)}]$ to be equal to the empty set $\emptyset$. The following lemma is immediate from the proof of Theorem 1.

Lemma 5 [S]. *Let xmin* and *xmax be determined from* $G(S')$, *let* x *be* any variable *in* S, *and let* $\tilde{x}$ *be* any value such that $\tilde{x} \in [xmin, xmax]$. The *system* of inequalities S is satisfiable if *and only* if $S \bigcup \{x \leq \tilde{x}, x \geq \tilde{x}\}$, i.e. S with $x = \tilde{x}$, is satisfiable.

We call an algorithm for LI(2) *constructive* if it supplies a feasible vector whenever the system of inequalities is satisfiable. Lemma 5 does not directly lead to a constructive algorithm for LI(2). We now give three preliminary lemmas that allow us to prove Lemma 9—the constructive version of Lemma 5.

Lemma **6.** *The* Shostak *extension S' is* a *closure of S.*

Proof. In order to establish the lemma we have to prove that S and S' are equivalent systems and that $G(S')$ is closed for S'. From Lemma 1 and the construction of the extension S' it is *easy* to see that S and S' are equivalent.

From Lemma 5 we have $[xmin, xmax] \subseteq \mathfrak{X}(S)$. Thus $G(S')$ is closed for S' if we can show that $[xmin, xmax] \supseteq \mathfrak{X}(S)$. Let $\tilde{x} \in \mathfrak{X}(S)$. If xmin $= -\infty$ then obviously $\tilde{x} \geq xmin$. If xmin $> -\infty$ we know from the way S' is constructed that the inequality x $\geq$ xmin can be derived from S and therefore any solution must satisfy it. Thus $\tilde{x} \geq xmin$. In the same way we can show that $\tilde{x} \leq$ xmax and therefore [xmin, $xmax$] $\supseteq \mathfrak{X}(S)$. Cl

Lemma 7. *If* there is an admissible path P *from* v_0 *to* x in $G(S')$, *there is* an admissible *simple* path Q from v_0 to x *in* $G(S')$ such that the sign *of* b_Q agrees with the sign *of* b_P.

Proof. Given *bp* let Q be a shortest admissible path from v_0 to x in $G(S')$ such that the sign of b_Q agrees with the sign of b_P. We claim that Q is simple. Suppose to the contrary that Q is not simple. By the admissibility of Q, the intermediate vertices of Q are distinct from v_0. Thus Q can be expressed as $Q_1Q_2Q_3$, the concatenation of three admissible paths Q_1, Q_2, and Q_3, where Q_2 is a simple loop. Let (a;, b_i, c_i), $1 \leq i < 3$, be the residues of the three paths and let $\langle a_Q, b_Q, c_Q \rangle$ be the residue of Q. We have two cases to consider, depending on whether Q_2 is permutable.

(a) If Q_2 is permutable then $a_2b_2 < 0$, i.e. a_2 and b_2 are of opposite signs. Since Q is admissible this implies that $b_1a_3 < 0$ so Q' $= Q_1Q_3$ is also admissible. Let $a_lx + b_ly \leq c_l$ be the inequality labelling the last edge of Q, and recall that the signs of b_Q and b_l agree. Since $a_lx + b_ly \leq c_l$ also labels the last edge of Q' the signs of b_Q and $b_{Q'}$ agree, which contradicts the choice of Q as a shortest admissible path.

(b) If Q_2 is not permutable then $a_2b_2 > 0$ and there exists (by definition of S') an edge e labelled $(a_2 + b_2)y \leq c_2$ from v_0 to y in G(S), where y is the initial vertex of Q_2. By the admissibility of Q we have $b_2a_3 < 0$, which implies that $Q' = eQ_3$ is also admissible, Since the two paths Q and Q' both end with the same edge we know that the signs of b_Q and $b_{Q'}$ agree. Thus we again have a contradiction to the choice of Q. □

Lemma 8. *If G(S') has no infeasible simple loops,* $xmin^{(n)} = xmin$ and $xmax^{(n)} = xmax$.

Proof. We show that $xmin^{(n)} =$ xmin; the proof of the other case similar. Trivially $xmin^{(n)} \leq$ xmin so it only remains to be shown that $xmin^{(n)} \geq$ xmin. If there is no admissible path from v_0 to x in $G(S')$ with $b_P < 0$ then $xmin = xmin^{(n)} = -\infty$ so let us assume that such paths exist and let P be one for which $c_P/b_P = xmin$. Since simple paths are of length at most n there exists an admissible path Q of length at most n from v_0 to x in G(S) such that the signs of b_P and b_Q agree (Lemma 7). Let Q be one for which $c_Q/b_Q = xmin^{(n)}$.

Let us add a new edge e between x and v_0 to $G(S')$ and label the new edge $x \leq xmin^{(n)}$. The only admissible loops of length at most n formed by adding this edge are of the form Q'e (or eQ'), where Q' is an admissible path of length at most n-1 from v_0 to x with $b_{Q'} < 0$. Thus from the edge e we have $x \leq xmin^{(n)} = c_Q/b_Q$ and from the path Q' we have $x \geq c_{Q'}/b_{Q'}$, where $c_Q/b_Q \geq c_{Q'}/b_{Q'}$ by the definition of Q and Q'. This implies that the hidden inequality $0 \leq c_Q/b_Q - c_{Q'}/b_{Q'}$ of Q'e is a tautology. Since G(S) did not contain any infeasible simple loops and adding the edge e did not introduce any, the modified graph has no infeasible simple loops. Thus it follows from Theorem 1 and Lemma 1 that $x \leq xmin^{(n)} = c_Q/b_Q$ and $x \geq xmin = c_P/b_P$ must be satisfiable simultaneously. We therefore have $xmin \leq xmin^{(n)}$. □

Lemma 9. If S' is the Shostak extension of S then $[xmin^{(n)}, xmax^{(n)}] = \mathfrak{X}(S)$ *for G(S).*

Proof. From Lemmas 6 and 8 it follows that this lemma is true when G(S) has no infeasible simple loops, so let us assume that $G(S')$ has an infeasible simple loop P. Thus S is unsatisfiable (Theorem 1) and therefore $\mathfrak{X}(S) = \emptyset$. We have two cases to consider depending on whether the initial vertex of P is v_0. If it is, then there exists a vertex x such that $x \geq xmin^{(n)}$, $x < xmax^{(n)}$, and $xmin^{(n)} > xmax^{(n)}$. Thus $[xmin^{(n)}, xmax^{(n)}] = \emptyset$. Otherwise G(S) has an infeasible simple loop and by definition $[xmin^{(n)}, xmax^{(n)}] = \emptyset$. *Cl*

3.2 Constructing a Solution

We will now show how to compute $xmin^{(n)}$ and $xmax^{(n)}$ from the graph $G(S^*)$ and how to use them to construct a feasible solution assuming that one exists. If the system of inequalities S is unsatisfiable then no feasible vector exists and we will detect that during our construction. However before we describe the algorithms we must show that $[xmin^{(n)}, xmax^{(n)}] = \mathfrak{X}(S^*) = \mathfrak{X}(S)$ for $G(S^*)$. If an infeasible simple loop in G(S) is found during the construction of S^* we know that $\mathfrak{X}(S) = \emptyset$ and we define $[xmin^{(n)}, xmax^{(n)}]$ to be equal to the empty set 0.

Theorem 5. *The extension S^* is a closure of S. Furthermore $[xmin^{(n)}, xmax^{(n)}] = [xmin, xmux]$ for $G(S^*)$.*

Proof. From Lemma 1 and the construction of the extension G(S*) it is easy to see that S^* and S are equivalent systems. We now show that G(S) is closed for S^*. If an infeasible loop in G(S) is detected during the construction of S* we have by definition $[xmin, xmax] \subseteq [xmin^{(n)}, xmax^{(n)}] = \mathfrak{X}(S) = \emptyset$. Let us therefore assume that no infeasible loop was found during the construction of S^*. This implies by Theorem 4 that there exists no infeasible simple loop in G(S). From the definition (6) of xlow and *xhigh* we see that the inequalities x $\geq$ xlow and x $\leq$ *xhigh* are as restrictive as any hidden inequality of a simple admissible loop. Thus the inequalities added to S in the construction of S* are at least as restrictive as those added to S in the construction of S'.

Hence if we compute $xmin^{(n)}$ and $xmax^{(n)}$ from $G(S^*)$ instead of from $G(S')$ we get an interval $[xmin^{(n)}, xmax^{(n)}]$, which is a subinterval of $\mathfrak{X}(S') = \mathfrak{X}(S)$ by Lemma 9. Let xmin and xmux be defined in terms of $G(S^*)$. We have by definition [xmin, $xmax$] $\subseteq$ $[xmin^{(n)}, xmax^{(n)}]$. Since x $\geq$ xmin and x $\leq$ $xmax$ can be derived from S they must clearly be satisfied in any feasible solution. It follows that $\mathfrak{X}(S^*) \subseteq$ [xmin, $xmax$] $\subseteq$ $[xmin^{(n)}, xmax^{(n)}] \subseteq \mathfrak{X}(S)$. Since S and S* are equivalent we have $\mathfrak{X}(S) = \mathfrak{X}(S^*)$. Hence $G(S^*)$ is closed for S^* and $[xmin^{(n)}, xmax^{(n)}]$ = [xmin, xmux]. □

In the algorithms presented in chapter 2 the auxiliary zero variable v_0 has never had any significance since it always occurs with zero coefficient. Thus all inequalities in S with only one variable have been ignored so far. Now is the time for v_0 to play its role.

The algorithm to compute $xmin^{(n)}$ and $xmax^{(n)}$ will be quite similar to Algorithm 1. It works on G(S*) insted of G(S) and starts by sending the guess $v_0 = 0$ (any other value will do) from v_0 and recording the most restrictive lower and upper bounds received at vertices adjacent to v_0. What this intuitively means is the following: If x is adjacent to v_0 the lower (upper) bound on x received is the most restrictive lower (upper) bound obtained from the original inequalities

with only one variable, and the inequality x $\geq$ slow (x $\leq$ *xhigh)* added in the construction of $G(S^*)$. The algorithm then proceeds to transfer new and more restrictive bounds on variables other than v_0 in a breadth-first way with n stages.

Algorithm 4 *(The Projector).* The input to the algorithm is the graph G(S*). The algorithm finds $xmin^{(n)}$ and $xmax^{(n)}$ for each variable x $\neq v_0$.

Step 1. [Send guess from v_0.] Let $i \leftarrow 1$. Transfer the value $g = 0$ over all edges incident to v_0. For each vertex x $\neq v_0$ record the most restrictive lower and upper bounds received on x.

Step 2. [Termination?] If $i < n$ set $i \leftarrow i + 1$ and go to Step 3. Otherwise the algorithm terminates and returns for each variable x $\neq v_0$ the current lower and upper bound on x as the result.

Step 3. [Stage i.] For each vertex x $\neq v_0$ do the following: (a) If the currently most restrictive lower bound on x was recorded in stage $i - 1$ send it over all its positive edges. (b) If the currently most restrictive upper bound on x was recorded in stage $i - 1$ send it over all its negative edges. (c) Record new, and more restrictive, bounds on x. Go to Step 2. □

Lemma 10. *Algorithm 4* computes $xmin^{(n)}$ and $xmax^{(n)}$ *for each* variable x $\neq v_0$.

The proof of Lemma 10 is essentially the same as the proof of Lemma 2 and is omitted. Having found the values of $xmin^{(n)}$ and $xmax^{(n)}$ we can use Theorem 5 and the definition of a closure of S to construct a feasible solution if one exists. The theorem and the definition of $\mathfrak{X}(S)$ tell us that if x is any variable of S and $\tilde{x}$ is any value such that $\tilde{x} \in [xmin^{(n)}, xmax^{(n)}] = \mathfrak{X}(S)$ then S is satisfiable if and only $S \bigcup \{x \leq \tilde{x}, x \geq \tilde{x}\}$ is satisfiable. Since adding the two inequalities x $\leq \tilde{x}$ and x $\geq \tilde{x}$ forces x to be equal to $\tilde{x}$ in any solution we have the following constructive algorithm to decide whether S is satisfiable.

Algorithm 5 *(LI(2)-Algorithm).* The input to the algorithm is the system of inequalities S. The algorithm determines whether S is satisfiable. If S is satisfiable the algorithm supplies a feasible vector.

Step 1. [Construct G(S).] Construct the graph G(S) for S as described in section 1.2. Compute λ from S as described in section 4.1 and mark all variables *unassigned.*

Step 2. [Construct S^*.] Use Algorithms 1, 2, and 3 to compute $xlow$ and $xhigh$ for each variable x $\neq v_0$. Add the corresponding inequalities x $\geq xlow$ and x $\leq xhigh$ to S.

Step 3. [Construct $G(S^*)$.] For each variable x $\neq v_0$ add two edges between x and v_0 to G(S) and label the edges x $\geq xlow$ and x $\leq xhigh$ respectively.

Step 4. [Compute $\mathfrak{X}(S)$.] Use Algorithm 4 to compute $\mathfrak{X}(S) = [xmin^{(n)}, xmax^{(n)}]$ for each variable x $\neq v_0$.

Step 5. [Terminate?] If all variables are marked *assigned* terminate the algorithm and return the constructed feasible vector. Otherwise let x be any variable marked unassigned.

Step **6.** [Assign value to x.] If the interval $\mathfrak{X}(S) = [xmin^{(n)}, xmax^{(n)}]$ is empty terminate the algorithm and return *unsatisfiable.* Otherwise mark x assigned and let x $\leftarrow \tilde{x}$, where $\tilde{x} \in [xmin^{(n)}, xmax^{(n)}]$.

Step 7. [Reduce the system.] Add two edges between x and v_0 to $G(S^*)$ and label the edges x $\geq \tilde{x}$ and x $\leq \tilde{x}$ respectively. Go to Step 4. ❑

4 COMPLEXITY

In this chapter we analyse the complexity of the **LI(2)-Algorithm.** We first show how to compute λ, which is needed as an initial bound in the algorithm and also enters into the analysis. In the same section we show that the algorithm runs in $O(mn^2 I|)$ time on a random access machine, where $|I|$ is the input size. We then turn to the Turing machine model in the second section and show that the algorithm is also polynomial time on a Turing machine.

4.1 Random Access Machine Model

In this section we show that the LI(2)-Algorithm runs in polynomial time on a random access machine. We start by showing how to compute λ, which is needed in the algorithm and also enters into the complexity analysis,

Let I denote an instance of LI(2), and let $|I|$ be the length of the string encoding I. Let κ be the product of the absolute value of the non-zero integers that represent the rational coefficients in the input. Clearly $\log_2 \kappa < |I|$.

We will now determine λ such that either slow $\in$ (--X/2, h/2) or $xlow = -\infty$ (the reason for dividing by two will be explained below). If $xlow \in$ (--X/2, $\lambda/2$) then by (6) there exists a lower loop P for x with hidden inequality x $\geq h = c_P/(a_P + b_P)$. Since a lower loop is of length at most n, it follows from (3) and the definition of κ that up $= 0$ or $\kappa^{-1} < |a_P| \leq \kappa$, $b_P = 0$ or $\kappa^{-1} \leq |b_P| \leq \kappa$, and $c_P = 0$ or $\kappa^{-1} \leq |c_P| \leq n\kappa$. Since up + $b_P \neq 0$ *we* have $\kappa^{-2} \leq |a_P + b_P| \leq 2\kappa$ and $cp = 0$ or $\kappa^{-2}/2 \leq |c_P/(a_P + b_P)| \leq n\kappa^3$. Let $\lambda = 3n\kappa^3$ and we have either slow $\in$ (--X/2, $\lambda/2$) or $xlow = -\infty$. Obviously, either $xhigh \in$ (--h/2, $\lambda/2$) or $xhigh = \infty$ for the same choice of λ.

From the previous chapters we know that the LI(2)-Algorithm terminates, but we do not know how many iterations are performed in Algorithms 2 and 3. We now bound the total number of iterations in the two algorithms. Let $x \geq h = c_p/(a_p + bp)$ and $x \geq h' = c_{p\prime}/(a_{p\prime} + b_{p\prime})$ be the hidden inequalities of two lower loops for x. By using our previous bounds on *ap, bp* and *cp* we find that h and h' must be equal if $|h - h'| < (\kappa^{-2}/2)^2$. Clearly this argument does not depend on the fact that we have two lower loops—it holds as well for two upper loops, or one upper and one lower loop. Let $\epsilon = \kappa^{-4}/4$. If h and h' correspond to lower and/or upper loops for some variable then $h \neq h'$ implies $|h - h'| \geq \epsilon$.

We know that in each iteration of Algorithm 2 the joint interval for *xlow* and *xhigh* is chopped by at least half, and that the new endpoint is obtained from a hidden inequality of a lower or an upper loop. Thus when the interval is of size less than ϵ the algorithm must terminate. Initially the interval is of size 2λ so the maximum number of iterations in Algorithm 2 is $\lceil \log_2(2\lambda/\epsilon) \rceil =$ O(log n + logn) $= O(|I|)$.

Let us now turn to Algorithm 3. From Lemma 4 we know that each time Algorithm 3 returns to Step 3 the interval $[xlow\downarrow, xlow\uparrow]$ has been chopped by at least half and that there exists a hidden inequality $x \geq h$ with $h > xlow\downarrow$. We also know that $xlow\downarrow$ has been obtained from a hidden inequality of a lower loop for x unless $xlow\downarrow = -\lambda$. By the choice of λ we can only have $xlow\downarrow = -\lambda$ the first time the algorithm gets to Step 3. This follows since initially the interval [$xlow\downarrow$, xlowf] is of size at most λ. Therefore $g = (xlow\downarrow + xlow\uparrow)/2 \leq -\lambda/2$ in Step 3 when $xlow\downarrow = -\lambda$. By definition h > -X/2. Thus h lies in the right half of $[xlow\downarrow, xlow\uparrow]$ and $xlow\downarrow \leftarrow h$ will be executed in Step 5 or 6 during the first iteration.

We conclude that Algorithm 3 returns to Step 3 at most $\lceil \log_2(\lambda/\epsilon)) \rceil =$ O(log κ + log *n)* $= O(|I|)$ times and the same bound holds when computing *xhigh.* Thus the total number of iterations to compute xlow and *xhigh* is at most $O(|I|)$. It is straightforward to see that Algorithm 1 takes $O(mn)$ time to perform the n stages of the breadth-first pushing of the guess. For both Algorithms 2 and 3 the amount of work in each iteration is dominated by the call of Algorithm 1. Hence the time to compute slow and *xhigh is* at most $O(mn|I|)$.

Let us now bound the amount of time necessary to reduce the number of variables by one using Algorithm 5. The time to construct the graph G(S) from S is $O(m$ + n). Since there are n different variables the total time to compute *xlow* and *xhigh* for each variable x, i.e. to find S', is $O(mn^2|I|)$. We can then construct $G(S^*)$ from G(S) in O(n) time. To compute $xmin^{(n)}$ and $xmin^{(n)}$ for each variable x requires one call of Algorithm 4, which is of complexity $O(mn)$ (the same as Algorithm 1). The remaining steps of Algorithm 5 take $O(m+n)$ time, so the total

time to reduce the number of variables by one is at most $O(mn^2|I|)$. Since the construction of the extension S^* only has to be done once and the total contribution to the running time from the other steps is $O(mn^2)$ we have the following theorem.

Theorem **6.** *The time complexity of the* **LI(2)-Algorithm** *is* $O(mn^2|I|)$ *on a random access machine* with *uniform cost* criterion.

4.2 Turing Machine Model

Since the Turing machine is polynomially related to the RAM model under logarithmic cost criterion we will start by examining the complexity of the algorithm on a RAM under logarithmic cost. In order to simplify the discussion we assume that all the coefficients in the input are integers, otherwise we multiply through by their greatest common denominator γ ($\log_2 \gamma < |I|$). We will also make one assumption of the way we choose $\tilde{x} \in [xmin^{(n)}, xmax^{(n)}]$ in Step 7 of the LI(2)-Algorithm. Let $\tilde{x}$ be any value in $[xmin^{(n)}, xmax^{(n)}]$ if the interval is finite. If the interval $[xmin^{(n)}, xmax^{(n)}]$ is infinite but has one finite endpoint let $\tilde{x}$ be equal to this endpoint, otherwise let $\tilde{x} = 0$.

The number of memory cells required by the LI(2)-Algorithm on a RAM is at most $O(m + n^2)$, where the non-linear term comes from Step 4 of Algorithm 1. We will now bound the size of the numbers that can occur. It is enough to consider results from operations involving multiplication. From (3) it follows that a_P, b_P, and c_P are integers if the coefficients in the input are integers. Since we only consider paths of length at most n in the algorithm we know that up, b_P, and c_P are bounded in magnitude by $n\kappa$. Thus bounds on variables obtained from lower or upper loops can be represented as pairs of integers whose magnitudes are bounded by $n\kappa$. The only other intermediate results obtained using multiplications are the bounds resulting from guesses in Algorithm 1 and in Algorithm 4. A guessed value can always be represented by a pair of integers whose magnitudes are bounded by λ. It is easily seen that the results obtained can be represented as pairs of integers whose magnitudes are bounded by $n\kappa\lambda$. We conclude that no operation takes more than $O(\log m + \log n + \log \kappa + \log \lambda + \log \gamma) = O(|I|)$ time under the logarithmic cost criterion. Thus the LI(2)-Algorithm runs in time $O(mn^2|I|^2)$ on a RAM under the logarithmic cost criterion and by Theorem 2 we have the following theorem.

Theorem 7. The time *complexity* of *the* **LI(2)-Algorithm** *is polynomial in the* size *of the input on* a *Turing machine.*

5 CONCLUSIONS

Solving the LI(2) problem, we have pushed the frontier of the problems in 9 a little bit further. As we have pointed out, extending the method presented to systems of linear inequalities with three variables per inequality will yield an algorithm for LP and therefore this extension is apt to be quite hard to find. Another possible extension would be to allow a fixed number of inequalities with more than two variables.

We have presented a new technique for tackling the LI(2) problem. Our main concern has been that the algorithm runs in polynomial time, so we have not mentioned several short-cuts that can reduce the practical running time. One of them is to use inequalities of the forms $x \geq c$ and $x \leq c'$, which are given in the system S as initial bounds for $x_{low}\downarrow$ and $x_{high}\uparrow$ instead of the theoretically derived λ. Many other modifications are possible and can make the algorithm more practical.

We hope that the techniques that have been introduced in this paper will prove useful in attacking the general LI problem and hence the LP problem. First steps in this direction have already been made by Shostak in [S].

REFERENCES

[AHU] Aho, A. V., J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms,* Addison-Wesley, Reading, Massachusetts (1974).

[GJ] Garey, M. R., and D. S. Johnson, *Computers* and *Intractability,* W. H. Freeman and Company, San Francisco, California (1979).

[I] Itai, A., "Two-commodity flow," *J.* ACM **25:4 (1978),** 595611.

[N] Nelson, C. G., "An $n^{\log n}$ algorithm for the two-variable-per-constraint linear programming satisfiability problem," Technical Report AIM-319, Computer Science Dept., Stanford University (1978).

[NO] Nelson, C. G., and D. C. Oppen, "Simplification by Cooperating Decision Procedures," *Comm.* ACM, to appear.

[P] Pratt, V. R., "Two easy theories whose combination is hard," unpublished manuscript (1977).

[RD] Reiss, S. P., and D. P. Dobkin, "The complexity of linear programming," Technical Report No. 69, Dept. of Computer Science, Yale University (1976).

[S] Shostak, R., "Deciding linear inequalities by computing loop residues," *Proc.* Fourth Workshop on Automatic Deduction, Austin, Texas **(1979),** to appear.