A Fast Algorithm for Solving Systems of Linear Equations with Two Variables per Equation*

Bengt Aspvall†
Computer Science Department
Stanford University
Stanford, California 94305

and

Yossi Shiloach
IBM Scientific Center
Technion City
Haifa, Israel

Submitted by Robert J. Plemmons

ABSTRACT

We present a fast algorithm for solving $m \times n$ systems of linear equations $A\mathbf{x} = \mathbf{c}$ with at most two variables per equation. The algorithm makes use of a linear-time algorithm for constructing a spanning forest of an undirected graph, and it requires $5m + 2n - 2$ arithmetic operations in the worst case.

1. INTRODUCTION

In this paper we present a fast algorithm for solving systems of m linear equations in n unknowns with at most two unknowns per equation. The problem of solving such systems will be denoted the LE(2) problem. Given an $m \times n$ matrix A with at most two nonzeros per row, an m-vector $\mathbf{c}$, and an n-vector $\mathbf{x}$ of unknowns, we want to determine whether

$$A\mathbf{x} = \mathbf{c} \tag{1}$$

has a unique solution, an infinite number of solutions, or no solution at all. If the system has an infinite number of solutions, we show how to partition the

*This work was supported by National Science Foundation Grants MCS-75-22870 and MCS-77-23738, and by a Chaim Weizmann Postdoctoral Fellowship.

†Author's current address: Department of Computer Science, Cornell University, Ithaca, New York 14853.

52 Vanderbilt Ave., New York, NY 10017 0024-3795/80/060117+8$01.75

problem into independent subproblems such that each one has at most one free parameter.

The approach is graph theoretical and makes use of a linear-time depth-first search algorithm for undirected graphs (see e.g. [1]). The relationship with partitioned sparse Gaussian elimination is explained. The algorithm requires $5m+2n-2$ arithmetic operations (additions, multiplications, and divisions) in the worst case.

2. PRELIMINARY RESULTS

We will assume that each equation of an LE(2) instance is of the form

$$ax+by=c, \qquad a\neq 0 \text{ and } b\neq 0, \tag{2}$$

where x and y are any two distinct variables, and a, b, and c are constants. That is, the matrix A has exactly two nonzero elements per row; no other assumptions about the structure of A are made. (The matrix A should be represented so that the m equations are readily available in this form.) Later on we will consider the case in which equations of the form $ax=c$ are present as well.

Let S be a given system of equations of the form (2). We construct an undirected graph $G(S)=(V, E)$ with m edges and n vertices as follows: (a) for each variable x occurring in S, add a vertex named x to $G(S)$; (b) for each equation $ax+by=c$ in S, add an edge between x and y to $G(S)$, and label the edge with the equation.

Actually, since $G(S)$ may contain multiple edges, it is in general a multigraph, but in this paper we will simply call $G(S)$ *the graph associated with* S. The components (the maximal connected components) of $G(S)$ partition the system S into disjoint subproblems that can be solved independently of each other; we will thus assume that $G(S)$ is connected.

THEOREM 1. *Let $G(S)$ be the graph associated with S (i.e., with $A\mathbf{x}=\mathbf{c}$). If $G(S)$ is a tree, then* $\operatorname{rank}(A)=n-1$, *where n is the number of vertices in $G(S)$.*

Proof. The theorem is trivially true for trees $G(S)$ with $n=1$ vertices, so let us assume that it holds for $n=1,2,\ldots,k$. Let $G(S)$ be a tree with $n=k+1$ vertices, and let $G'(S)$ be the tree obtained from $G(S)$ by deleting a vertex y incident with only one edge together with its incident edge (such a vertex y exists in any tree). Thus, $G'(S)$ is the graph $G(S')$ for some system S', $A'\mathbf{x}'=\mathbf{c}'$, and by the inductive hypothesis, $\operatorname{rank}(A')=k-1$. Since y does not occur in S, and there is an equation of the form $ax+by=c$ in S, we have $\operatorname{rank}(A)>\operatorname{rank}(A')=k-1$. Thus, $\operatorname{rank}(A)=k$, which completes the inductive proof. ■

The theorem tells us that if $G(S)$ is a tree, then the system S has a one-dimensional solution space, and the solution space can thus be expressed using one free parameter. We end this section by describing the behavior of the algorithm for the case in which $G(S)$ is a tree; this will essentially be the first of the algorithm's two phases.

Select an arbitrary vertex x as the root of the tree $G(S)$, and let p be the free parameter associated with $G(S)$. All the variables of S will be expressed as linear combinations of p, and we arbitrarily choose to express the variable x as $x=1\times p+0$. Traverse the tree $G(S)$ in depth-first order, and for each new vertex to be explored, do the following: If y is the immediate ancestor of z, $ay+bz=c$ the equation labeling the edge between y and z, and $y=a'p+c'$ the expression for y previously computed, then express the variable z as $z=a''p+c''$, where $a''=-aa'/b$ and $c''=(c-ac')/b$. These $2n$ coefficients completely characterize the solution space and can thus be returned as the solution of S.

We should point out that it is not necessary to perform a depth-first search of the graph—any search algorithm for constructing a spanning tree will do. However, because of its simplicity, we will use the depth-first search throughout this paper.

3. THE ALGORITHM

We now turn to the case in which $G(S)$ is a connected graph, but not necessarily a tree. Since any connected graph has a spanning tree, we have the following corollary to Theorem 1.

COROLLARY 1. *Let $G(S)$ be the graph associated with S (i.e., with $A\mathbf{x}=\mathbf{c}$). If $G(S)$ is connected, then* $\operatorname{rank}(A)\geqslant n-1$, *where n is the number of vertices in $G(S)$.*

Given the graph $G(S)=(V, E)$ for S, the algorithm starts by constructing a spanning tree $T(S)=(V, E')$, $E'\subseteq E$, for $G(S)$ using a depth-first search and with an arbitrarily selected vertex x as the root. The variables will be expressed as linear functions of p, the free parameter associated with $T(S)$, exactly as described in Sec. 2 for the "tree case." When a back-edge (an edge connecting the currently explored vertex with one that already has been explored) is encountered during the construction of $T(S)$, the action of the algorithm depends on the equation labeling the edge.

Let z be the currently explored vertex, and assume that the corresponding variable z has been expressed as $z=a'p+c'$. Furthermore, let y be the previously explored vertex incident with the back-edge, and assume that the variable y has been expressed as $y=a''p+c''$. The equation labeling the back-edge is of the form $ay+bz=c$, and we can therefore express the

variable z also as $z=a'''p+c'''$, where $a'''=-aa''/b$ and $c'''=(c-ac'')/b$. By subtracting the two expressions for z, we obtain the equation $0=(c'-c''')+(a'-a''')p$. Hence, if $a'\neq a'''$, then p must equal $-(c'-c''')/(a'-a''')$ if a solution to S exists. If $a'=a'''$ and $c'=c'''$, the equation labeling the back-edge is a linear combination of previously examined equations. Finally, if $a'=a'''$ and $c'\neq c'''$, we can derive the invalid relation $0\neq 0$, which implies that the system S has no solution.

The different effects of the back-edges lead us to characterize them as either *decisive*, *redundant*, or *contradictory* edges. The redundant edges can simply be deleted from $G(S)$ as they are encountered; if a contradictory edge is found, the algorithm terminates, indicating that no solution exists. The interesting result is a decisive edge: it gives the value that the parameter p must have if a solution to S exists. Thus, $G(S)$ contains a decisive edge if and only if $\text{rank}(A)=n$. The first phase of the algorithm ends either when a decisive or contradictory edge is found or when the depth-first search of the entire graph is completed. In the latter case, all the back-edges were redundant, and S has a one-dimensional solution space characterized by the spanning tree $T(S)$ as described in Sec. 2.

We now turn to the second phase of the algorithm, assuming that a decisive edge has been found during the first phase. Thus S has either a unique solution or no solution at all—the unexplored part of $G(S)$ will tell. From the decisive edge, we know what value the parameter p must have if a solution exists. We can therefore assume that a unique solution exists and start computing it; if S does not have a solution, there must be an edge whose equation will be violated. Recall the behavior of the algorithm when $G(S)$ is a tree. During the depth-first search, a particular solution could have been computed if p had been assigned a certain value beforehand. We use a similar idea in the second phase of the algorithm.

The second phase of our algorithm consists of a depth-first search, as did the first one. A spanning tree $T(S)$ for $G(S)$ is constructed with the previously selected vertex x as its root. Initially, the variable x corresponding to the root is assigned the value of p, since $x=1\times p+0$. Whenever a new vertex is explored, the value of the corresponding variable is computed. For each back-edge encountered, we check to see if the labeling equation is satisfied by the computed values of its variables. If so, the back-edge is a redundant edge and the algorithm continues. Otherwise, the back-edge is a contradictory edge, and the algorithm terminates.

ALGORITHM 1 (The LE(2) solver).

Step 1 [Initialize]. Construct $G(S)$ from S as described in Sec. 2.

Step 2 [Split into subproblems S_i]. For each component $G_i(S_i)$ of $G(S)$, do the following: Let p_i be the free parameter associated with $G_i(S_i)$. Select a root vertex and perform Steps 3 and 4.

Step 3 [Phase 1]. Perform a depth-first search on $G_i(S_i)$: (a) for each explored vertex x, express its corresponding variable as $x=ap_i+c$; (b) remove all redundant edges; (c) if a contradictory edge is found during the search, terminate the algorithm, responding "S has no solution." If a decisive edge is found during the search, compute the value of p_i and go to Step 4. Otherwise, respond "Subsystem S_i has the one-dimensional solution space $\mathbf{x}=\mathbf{a}\times p_i+\mathbf{c}$" and continue with the next component.

Step 4 [Phase 2]. Given the value of p_i, perform a depth-first search on $G_i(S_i)$: (a) for each explored vertex x, compute the value x^* of the corresponding variable; (b) remove all redundant edges; (c) if a contradictory edge is found during the search, terminate the algorithm, responding "S has no solution." Otherwise, respond "Subsystem S_i has the unique solution $\mathbf{x}=\mathbf{x}^*$" and continue with the next component.

So far we have excluded one-variable equations from our consideration. We call a variable *fixed* if the variable occurs in an equation of the form $ax=c$, where $a\neq 0$. If a subsystem S_i contains a fixed variable x, then perform a depth-first search starting at the corresponding vertex x, and do the following: Whenever a new vertex is explored, the value of the corresponding variable is computed from the value of its immediate ancestor and the equation labeling the edge between them. Equations labeling back-edges are checked for consistency, as are computed values for all fixed variables in S_i. The depth-first search of $G_i(S_i)$ terminates either when a contradiction is found or when the search of the component is completed and the unique solution of S_i computed.

4. A GAUSSIAN-ELIMINATION VIEW

In this section, we show that Algorithm 1 can be interpreted as a partitioned sparse Gaussian elimination. Let S be a system of equations $A\mathbf{x}=\mathbf{c}$, where A is an $m\times n$ matrix with exactly two nonzero elements per row; assume that the associated graph $G(S)$ is connected. We claim that the rows and columns of A can be permuted so that

$$P_1AP_2=\begin{pmatrix} A_1 & \mathbf{b}_1 \\ A_2 & \mathbf{b}_2 \end{pmatrix},$$

where A_1 is a lower triangular $(n-1)\times(n-1)$ matrix with nonzero diagonal elements and $\mathbf{b}_1=(\beta,0,0,\ldots,0)^T$, $\beta\neq 0$, is an $(n-1)$-vector. [Since A_1 is triangular and has non-zero diagonal elements, it is nonsingular; thus $\operatorname{rank}(A)\geqslant n-1$.]

Proof of claim: Let $T(S)$ be a spanning tree of $G(S)$. Select any vertex of degree one in $T(S)$. Assign the number n to the selected vertex and the number 1 to its adjacent vertex. Assign numbers, from 2 to $n-1$, to the remaining vertices by repeatedly selecting a vertex adjacent to some already numbered vertex. Order the equations labeling the edges of $T(S)$ appropriately, and put the equations labeling the back-edges of $G(S)$ at the "bottom" (i.e., as the last $m-n+1$ equations).

Without loss of generality, we now assume that $A\mathbf{x}=\mathbf{c}$ can be written in the partitioned form

$$\begin{pmatrix} A_1 & \mathbf{b}_1 \\ A_2 & \mathbf{b}_2 \end{pmatrix}\begin{pmatrix} \tilde{\mathbf{x}} \\ \xi \end{pmatrix}=\begin{pmatrix} \mathbf{c}_1 \\ \mathbf{c}_2 \end{pmatrix},$$

where $\tilde{\mathbf{x}}$ and $\mathbf{c}_1$ are $(n-1)$-vectors, ξ is a scalar, and $\mathbf{c}_2$ is an $(m-n+1)$-vector. Let

$$A=\begin{pmatrix} A_1 & 0 \\ A_2 & \mathbf{b}_2-A_2A_1^{-1}\mathbf{b}_1 \end{pmatrix}\begin{pmatrix} I & A_1^{-1}\mathbf{b}_1 \\ 0 & 1 \end{pmatrix}=LU$$

be an LU-decomposition of A. Here U is an $n\times n$ unit upper triangular (and hence nonsingular) matrix.

Clearly, $A\mathbf{x}=\mathbf{c}$ has a solution if and only if $L\mathbf{y}=\mathbf{c}$ has a solution. We show now how to determine $\mathbf{y}=(\tilde{\mathbf{y}}^T,\eta)^T$, where η is a scalar. Since A_1 is nonsingular, $\tilde{\mathbf{y}}$ is uniquely determined by $A_1\tilde{\mathbf{y}}=\mathbf{c}_1$. Let $\tilde{\mathbf{z}}$ be the $(n-1)$-vector determined (uniquely) by $A_1\tilde{\mathbf{z}}=\mathbf{b}_1$. Given $\tilde{\mathbf{y}}$ and $\tilde{\mathbf{z}}$, we then solve the $(m-n+1)\times 1$ system

$$(\mathbf{b}_2-A_2\tilde{\mathbf{z}})\eta=(\mathbf{c}_2-A_2\tilde{\mathbf{y}}),$$

which has a unique solution, an infinite number of solutions, or no solution at all.

Phase 1 of Algorithm 1 corresponds to the computation of $\mathbf{y}$ (and $\tilde{\mathbf{z}}$), with η playing the same role as the parameter p. Given any solution $\mathbf{y}$, a corresponding solution $\mathbf{x}$ can be obtained by solving $U\mathbf{x}=\mathbf{y}$, i.e., $\xi=\eta$ and $\tilde{\mathbf{x}}=\tilde{\mathbf{y}}-\eta\tilde{\mathbf{z}}$ (with $\tilde{\mathbf{z}}$ defined as above); this corresponds to phase 2 of Algorithm 1.

5. COMPLEXITY

Our complexity measure will be the number of arithmetic operations required, as is customary in computational linear algebra. Since this number depends on the different types of back-edges that are found, we will assume

that S has a unique solution and that $G(S)$ is connected. Thus the graph must have a decisive edge or a fixed variable. The worst case occurs when the graph has a decisive edge, and this edge is the last back-edge to be examined. In order to express the variables in terms of the free parameter p, which is done both when traversing tree-edges and back-edges, the first phase requires $m-1$ additions, $2(m-1)$ multiplications, and $2m$ divisions. (We do not count any additions and multiplications when substituting $x=1\times p+0$ into the first examined equation.) To compute the value of p from the decisive edge requires two more additions and one division. The algorithm also uses $2(m-n)+1$ tests for equal operands when examining the back-edges. Given the value of p, the second phase requires $n-1$ additions and $n-1$ multiplications to compute the values of the variables; this yields a total of $m+n$ additions, $2m+n-3$ multiplications, $2m+1$ divisions, and $2(m-n)+1$ comparisons.

By applying the algorithm separately to each component of $G(S)$, we conclude that the algorithm requires $5m+2n-2$ arithmetic operations in the worst case; if we count the number of comparisons as well, the number of operations is $7m-1$ in the worst case. If the system S has an infinite number of solutions, or no solution at all, the algorithm requires fewer arithmetic operations. A linear-time depth-first search algorithm for constructing a spanning forest of an undirected graph processes the components one at a time and can be adapted, with only minor modifications, to solve the LE(2) problem.

6. CONCLUSIONS

The graph-theoretical approach to the problem of solving sparse systems of linear equations was first used by Parter in 1961 [4]. Since then the subject has been treated extensively in the literature (e.g., [3, 5, 6]). In many algorithms, the edges of a graph are used to represent the nonzero structure of the matrix A, rather than the explicit equations. Some properties of the graph are then explored in order to minimize the amount of computation when Gaussian elimination is used.

We have shown that for systems of linear equations with at most two variables per equation, the edges of a graph are well suited to represent the explicit equations. Similar situations occur in, for example, linear programming and logic when the number of variables per constraint, or respectively per clause, is at most two [2]. For the LE(2) problem, the key observation is that a given system can be partitioned into independent subsystems with at most one free parameter.

It would be interesting to determine whether an approach similar to the presented one can be used for systems with at most three variables per

equation. Can this problem be partitioned into subproblems with relatively few free parameters using some combinatorial structure? Are k-trees interesting generalizations in this context? Any dense $n \times n$ system of linear equations can, by the introduction of new variables, be transformed into a sparse $n^2 \times n^2$ system with at most three variables per equation. It will therefore probably be difficult to find a method using only $O(n^\epsilon)$ free parameters for some small positive ϵ, but an $O(\sqrt{n})$ bound on the number of free parameters would be helpful in developing an efficient algorithm for sparse systems without any apparent structure.

As we mentioned earlier, it is not necessary to perform a depth-first search of the graph. Any algorithm for constructing a spanning tree can be used. This has the advantage that we can try to direct the search so as to minimize the effect of rounding errors if Algorithm 1 is used with finite-precision arithmetic.

When a new edge is traversed, we compute $a'' = -aa'/b$ and $c'' = (c - ac')/b$, where a, b, and c are coefficients given in the input and a' and c' have been computed previously. One strategy for reducing the growth of rounding errors would be to always traverse the edge for which the quotient $|a/b|$ is least. This would require, however, a priority-queue structure. A simpler heuristic would be to first examine equations for which $|a/b|$ is less than some given threshold before examining the remaining ones.

We have also a choice in selecting the root vertex, but we know of no simple heuristic to select the "best" root without trying several alternatives; further investigation is needed.

We would like to thank Donald J. Rose for suggesting the Gaussian-elimination interpretation of Algorithm 1.

REFERENCES

1 A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Mass., 1974.

2 B. Aspvall, M. F. Plass, and R. E. Tarjan, A linear-time algorithm for testing the truth of certain quantified Boolean formulas, *Information Processing Lett.* 8(3):121–123 (1979).

3 I. S. Duff, A survey of sparse matrix research, *Proc. IEEE* 65(4):500–535 (1977).

4 S. Parter, The use of linear graphs in Gaussian elimination, *SIAM Rev.* 8(2):119–130 (1961).

5 D. J. Rose, A graph-theoretic study of the numerical solution of sparse positive definite systems of linear equations, in *Graph Theory and Computing*, Academic, New York, 1972, pp. 183–217.

6 R. E. Tarjan, Graph theory and Gaussian elimination, in *Sparse Matrix Computations* (J. R. Bunch and D. J. Rose, Eds.), Academic, New York, 1976, pp. 3–22.

Received 27 November 1979; revised 2 May 1980