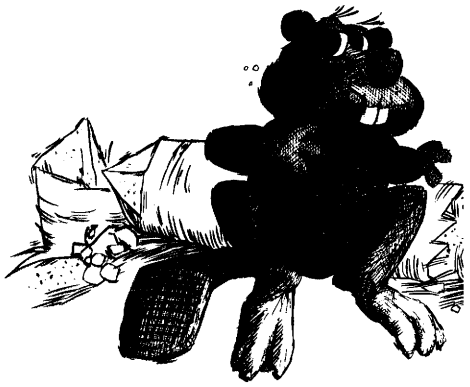
UNIVERSITY OF WATERLOO
UNIVERSITY OF WATERLOO
UNIVERSITY OF WATERLOO
COMPUTER SCIENCE DEPARTMENT
COMPUTER SCIENCE DEPARTMENT
COMPUTER SCIENCE DEPARTMENT



A Survey of Systolic Systems for Solving the Algebraic Path Problem

Faron Moller

CS-85-22

August, 1985

A Survey of Systolic Systems for Solving the Algebraic Path Problem

Faron Moller

Department of Computer Science
University of Waterloo

M.Math Essay

ABSTRACT

This essay describes various methods for solving instances of the Algebraic Path Problem on systolic arrays of processors. Systolic arrays form the basis of the type of parallel processing developed by Kung and Leiserson ([KUN78]). The Algebraic Path Problem is a general problem which under different interpretations yields the problems of computing the inverse of a matrix of elements taken from a field, determining the shortest paths between vertices in a weighted directed graph, and computing the reflexive and transitive closure of a binary relation, among many other important problems. This paper starts out by giving an overview of systolic architecture and algorithms, and a description of the Algebraic Path Problem. Several systolic solutions to (instances of) the Algebraic Path Problem are then described, followed by a discussion comparing the described methods.

July 22, 1985

Table of Contents

A Survey of Systolic Systems for Solving the Algebraic Path Problem

Faron Moller

Department of Computer Science
University of Waterloo

M.Math Essay

1. Introduction

Often in abstract algebra, you can develop theories and propose some abstract problem in a very general setting, solve the problem, and have it be the solution to several important concrete instances of the general problem. That is, in the setting of abstract algebra, you can manipulate a set with some operations defined on it, and solve some problem defined in very general terms within the algebraic structure; under different interpretations of the structure, often seemingly unrelated problems are solved with the same solution. This is precisely the case with the Algebraic Path Problem. This problem is a problem defined over a general semiring, but under different interpretations of the semiring, the problem can be interpreted to represent (among other things) the problems of matrix inversion over a field, the shortest distances in a weighted graph, the reflexive and transitive closure of a binary relation, the maximum capacity paths in a network, the Kleene star closure of a language, and the analysis of global data flow. It will be seen that the general solution to the Algebraic Path Problem can be expressed as a generalized version of the Gauss-Jordan Elimination Algorithm common in numerical analysis.

The purpose of this paper is twofold. Firstly, it attempts to outline the development of the Algebraic Path Problem and the work which has been done to solve it. Secondly and more importantly, the paper describes attempts to solve the problem with systolic algorithms. To achieve these two goals, the paper starts out by defining systolic networks and algorithms, followed by a description of the Algebraic Path Problem. Several systolic solutions to instances of the Algebraic Path Problem are then described, followed by a comparison of these methods.

A systolic algorithm is an algorithm which is performed on a specialized parallel network of processors. All of the solutions to instances of the Algebraic Path Problem described in this paper are related in that they are all systolic algorithms which use a version of the Gauss-Jordan Elimination Algorithm to solve the problem in question. Other parallelizations of algorithms for solving instances of the Algebraic Path Problem have been developed (eg, [LEV72], [KAN78], [SAM78]), but none of these will be discussed in this paper.

2. Systolic Networks and Algorithms

Systolic algorithms were introduced by Kung and Leiserson ([KUN78]), and since that time there has been much effort spent on designing efficient algorithms to be executed on various systolic systems. Systolic algorithms have been found to be appropriate for such problems as various matrix computations (eg., see [KUN78], [KUN79], [KUN80], [MEA80], [KUN82], [KRA83], and [ULL84]), dynamic programming ([GUI79]), and priority queues (eg, [KUN79], [LEI79], [KUN80]). The purpose of this section of the paper is to give a brief description of what is meant by a systolic algorithm, and to give a few examples which will be of importance in the following sections of the paper.

2.1. Definition of Systolic Algorithms

In general, a parallel algorithm is a collection of independent task modules which are executed simultaneously and which communicate with each other during the execution of the algorithm. A *systolic system* is a regular network of processors running in parallel which circulate data in a simple regular fashion. The word *"systole"* is a physiological term referring to the rhythmically recurrent contractions of the heart and arteries which pulse blood through the body. Each processor in a systolic system pumps multiple streams of data through itself, so that the regular beating of these parallel processors maintains a constant flow of data throughout the entire network. As the data pass through a processor, they are involved in some constant-time computation which may update their values in the process. A *systolic algorithm* is an algorithm which runs on a systolic system.

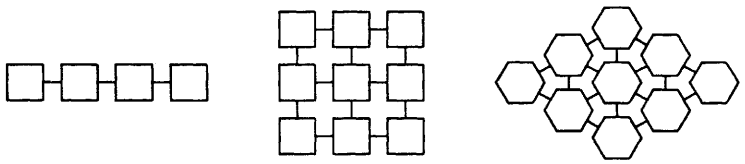
2.2. Communication Geometries

The geometry of the communication paths in a systolic system is simple and regular, to allow for a low-cost and high-performance chip implementation, and to allow for easy development and verification of algorithms to be executed on the system. The processors are all laid out in a patterned fashion, and the only connections which exist between processors in the network simply link neighbouring processors.

The most common arrangements of processors in a systolic system are the three mesh-connected systolic arrays described as follows:

Linear

The one-dimensional linear array (Fig. 2-1*(a)*) is the simplest geometry for connecting processors. However, it has proven to be a very useful network all the same. It has been used for systolic algorithms for such problems as odd-even transposition sort, finite impulse response (FIR) filtering, recurrence evaluation ([KUN80]) and priority queues ([LEI79]).



(a) Linearly-connected *(b) Orthogonally-connected* *(c) Hexagonally-connected*

Fig. 2-1: Mesh-connected systolic arrays.

Orthogonal

The two-dimensional orthogonal array (Fig. 2-1*(b)*) is the most well-known geometry for connecting processors, being one of the very first communication geometries studied. It was the geometry of the cellular automata studied by von Neumann in the early 1950's ([KUN80]). It has been used for systolic algorithms for such problems as relaxation methods for solving partial differential equations ([KUN80]), dynamic programming and transitive closure ([GUI79]), and various other graph problems ([LEV72]).

Hexagonal

The two-dimensional hexagonal array (Fig. 2-1*(c)*) seems to be the most appropriate geometry for performing matrix operations. It is derived by adding diagonal connections in only one direction to the orthogonal array, and in fact is equivalent

in computational power to the orthogonal array, to within a constant factor ([CUL84]). However, the matrix algorithms run more smoothly on the hexagonal array, as they do not have to simulate the diagonal motion. It has been used for systolic algorithms for such problems as matrix multiplication, LU decomposition, and discrete Fourier transforms ([KUN80]).

Other systolic networks have been studied and found to be useful for certain applications. For example, systolic trees have been used for searching algorithms and priority queues ([LEI79]), parallel function evaluation and recurrence evaluation ([KUN80]), and shuffle-exchange networks have been used for the fast Fourier transform and bitonic sort ([KUN80]). However, in this paper we will only be interested in the three mesh-connected systolic arrays described above.

2.3. Examples of Systolic Algorithms

The following examples give the flavour of how a systolic algorithm operates. The particular examples given are both important in the following development of methods for solving the Algebraic Path Problem, as they both utilize processors which perform the so-called *inner product step* computation: $c \leftarrow c + a \times b$. The first example, which performs a matrix-vector multiplication, uses a linear array of inner product step processors (Fig. 2-2*(a)*), and the second example, which performs a matrix-matrix multiplication, uses a hexagonal array of inner product step processors (Fig. 2-2*(b)*). They both appeared in [KUN78] and are two of the first systolic algorithms developed.

2.3.1. Matrix-Vector Multiplication

To compute the matrix-vector product $y = Ax$, where A is an $n \times n$ matrix $A = (a_{ij})$, and x and y are n-tuples, the following recurrence can be used:

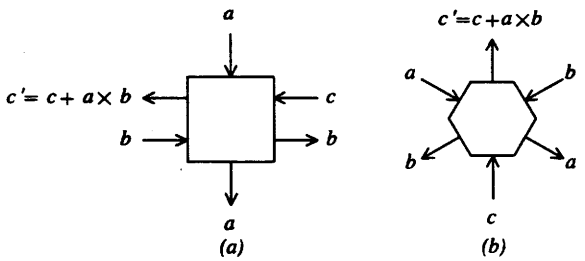


Fig. 2-2: Inner product step processors.

$$y_i^{(0)} = 0,$$

$$y_i^{(k)} = y_i^{(k-1)} + a_{ik}x_k,$$

$$y_i = y_i^{(n)}.$$

Thus the only computations that are done using this recurrence are inner product calculations.

If A is a band matrix with band width ω, then the matrix-vector multiplication can be performed using the above recurrence on a linear systolic array of ω inner product step processors (for a full matrix-vector multiplication problem, $2n-1$ processors would be needed using this method). The example band matrix-vector multiplication problem in Fig. 2-3 has a band width of $\omega=4$, and can be performed on the linear array depicted in Fig. 2-4. The algorithm operates by pumping the data synchronously through the array as outlined in Fig. 2-4, with the four bands of the A matrix being fed in from the top, the x vector being pumped in from the left end, and the y vector being fed through from the right end. Initially, the y_i's are set to 0 (in general, the problem actually being solved is $y \leftarrow Ax+y$), and in passing through the array, each y_i accumulates all of its terms $a_{i,i-2}x_{i-2}$, $a_{i,i-1}x_{i-1}$, $a_{i,i}x_i$, and $a_{i,i+1}x_{i+1}$.

It is clear that after ω steps in the computation, the y_i's start emerging from the left end of the array, and that $2(n-1)$ steps later the last component y_n of y leaves the array. Thus the array takes $2(n-1)+\omega$ steps to do the band matrix-vector

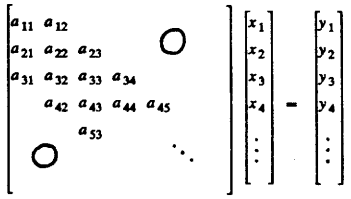


Fig. 2-3: Band matrix-vector multiplication.

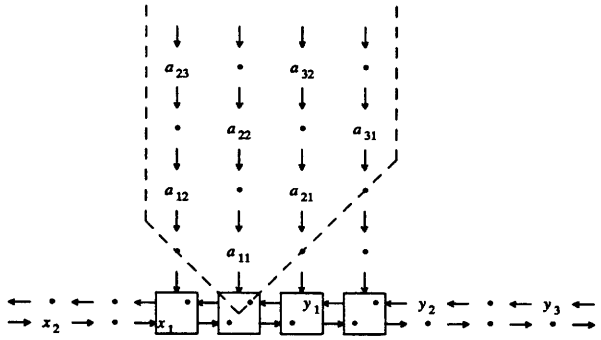


Fig. 2-4: Systolic system for computing the band matrix-vector multiplication in Fig. 2-3.

multiplication with band width ω. However, from the way in which the data are staggered, only half of the processors are actually active at any given step in the computation. Hence the performance of the array can be improved either by coalescing adjacent processors, thereby using an array of $\frac{\omega}{2}$ processors, or by computing two equal-sized matrix-vector products at the same time. To do this, each of the values a_{ij}', x_i', and y_i' of the second system would be fed into the array immediately after the corresponding values a_{ij}, x_i, and y_i of the first system. In this way, all processors would be active at each step of the computation. Also, immediately after the last components of the first two systems have entered the array, the components of yet another

two systems could enter the array, to compute two more matrix-vector products. Continuing in this way, the array would finish processing two matrix-vector products every $2n$ steps.

2.3.2. Matrix-Matrix Multiplication

To compute the matrix-matrix multiplication $C=AB$, where $A=(a_{ij})$, $B=(b_{ij})$, and $C=(c_{ij})$ are three $n\times n$ matrices, the following recurrence can be used:

$$c_{ij}^{(0)}=0,$$

$$c_{ij}^{(k)}=c_{ij}^{(k-1)}+a_{ik}b_{kj},$$

$$c_{ij}=c_{ij}^{(n)}.$$

Again the only calculations performed are inner product calculations.

If A and B are band matrices of band widths ω_1 and ω_2 repectively, then the matrix-matrix multiplication can be performed using the above recurrence on a hexagonal systolic array of $\omega_1\omega_2$ inner product processors (for a full matrix-matrix problem, $(2n-1)^2$ processors would be needed using this method). The example band matrix-matrix multiplication problem in Fig. 2-5 has matrices of band widths $\omega_1=\omega_2=4$, and can be performed on the hexagonal array depicted in Fig. 2-6.

$$\begin{bmatrix} a_{11} & a_{21} & & & \\ a_{21} & a_{22} & a_{23} & & \\ a_{31} & a_{32} & a_{33} & a_{34} & \\ & a_{42} & & \ddots & \\ & & \ddots & & \end{bmatrix} \begin{bmatrix} b_{11} & b_{21} & b_{13} & & \\ b_{21} & b_{22} & b_{23} & b_{24} & \\ & b_{32} & b_{33} & b_{34} & b_{35} \\ & & b_{43} & & \\ & & & \ddots & \end{bmatrix} - \begin{bmatrix} c_{11} & c_{21} & c_{13} & c_{14} & \\ c_{21} & c_{22} & c_{23} & c_{24} & \\ c_{31} & c_{32} & c_{33} & c_{34} & \\ c_{41} & c_{42} & & & \\ & & & & \ddots \end{bmatrix}$$

Fig. 2-5: Band matrix multiplication.

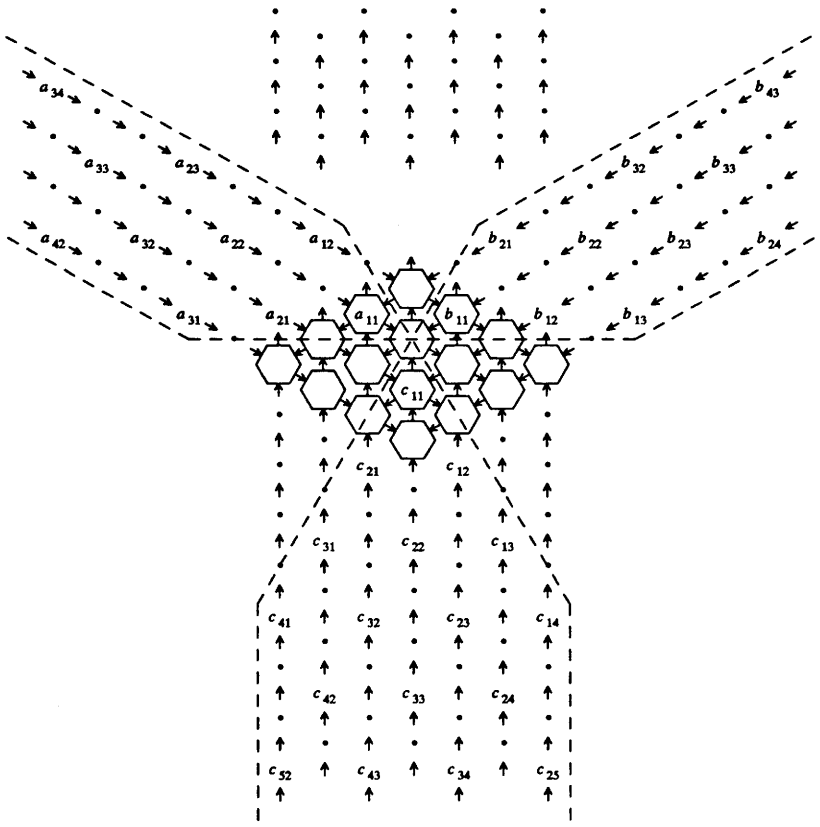


Fig. 2-6: Systolic system for computing the band matrix multiplication in Fig. 2-5.

Again, the algorithm operates by pumping the data synchronously through the array as outlined in Fig. 2-6, with the four bands of the A matrix being pumped in from the top left, the four bands of the B matrix being fed in from the the top right, and the seven bands of the solution matrix C being fed through the array from below. Initially, the c_{ij}'s are set to 0 (again in general, the problem actually being solved is $C \leftarrow AB + C$), and in passing through the array, each c_{ij} accumulates all of its terms.

The first element of the solution matrix C which enters the array is c_{11} and the last element of C to exit the array is c_{nn}. After $min(\omega_1,\omega_2)$ steps in the computation, the c_{11} element leaves the array of processors, and $3(n-1)$ steps later, the c_{nn} element leaves the array. Thus the array takes $3(n-1)+\min(\omega_1,\omega_2)$ steps to do the band matrix-matrix multiplication with band widths ω_1 and ω_2. However, once again from the way that the data are staggered, only one third of the processors are actually active at any given step in the computation. To improve processor utilization, three equal-sized matrix-matrix products can be computed at the same time by feeding in each of the values a_{ij}', b_{ij}', and c_{ij}' of the second system into the array immediately after the corresponding values a_{ij}, b_{ij}, and c_{ij} of the first system, and each of the values a_{ij}'', b_{ij}'', and c_{ij}'' of the third system into the array immediately after the corresponding values of the second system. In this way, all of the processors would be active at each step of the computation. Also the components of three new systems can be fed into the array immediately after the last components of the first three systems have entered the array. Continuing in this way, the array would finish processing three matrix-matrix products every $3n$ steps.

3. The Algebraic Path Problem

The Roy-Warshall Algorithm for computing the transitive closure of a Boolean matrix ([ROY59], [WAR62]), Floyd's Algorithm for computing the shortest distances between nodes in a directed weighted graph ([FLO62]), Kleene's proof that every regular language can be defined by a regular expression ([KLE56]), and the Gauss-Jordan Algorithm for inverting real matrices are all specific instances of a single general algorithm. These above problems are only a few special instances of what is called the Algebraic Path Problem.

3.1. Historical Development

The first person to start describing the relationship amongst the above problems was Carré [CAR71]. He linked the problem of solving systems of linear equations with path problems in graph theory by defining the problem in a general setting using a semiring as his underlying algebraic structure for the problem. He formulated the problems in a unified manner, and arrived at a common solution description for them which he could present as either a sum of measures of paths or as a solution to a system of linear equations. [AHO74] and [BAC75] first established the relationship between the Algebraic Path Problem and problems in regular algebra and language theory. They proposed axiom systems within which the problem could be easily mapped onto these two realms. However, they each assumed that addition in their algebraic structure was idempotent (*ie.*, $a+a = a$), so they could not arrive at a completely general solution to the problem which in particular could engulf the problem of solving systems of linear equations over the real numbers, as this structure does not have an idempotent addition. The first person to derive a suitably weak algebraic structure within which he could describe the Algebraic Path Problem in its fullest generality was Lehmann ([LEH77]).

In the remainder of this section of the paper, three approaches to the Algebraic Path Problem shall be explored. The first is the axiomatic approach taken by Lehmann ([LEH77]) which, as was noted above, generalizes all previous attempts. The second approach is due to Tarjan ([TAR75], [TAR81], and [TAR81B]), and gives a

description which parallels the development of regular expressions and focuses on the relationship of the Algebraic Path Problem to Language Theory. The final approach described is due to Rote ([ROT83]) and focuses on the problems of determining shortest distances in a weighted graph and performing matrix inversion (ie, solving systems of linear equations). A general solution will then be given which is taken from [ROT83] and appears in a slightly different form in [LEH77].

3.2. Lehmann's Approach

Lehmann considers algebras of the form $<S,+,\times,^*,0,1>$ where S is a set, + and $\times$ are binary operations called *addition* and *multiplication* respectively, * is a unary operation called *closure*, and $0,1 \in S$ are constants. A closed semiring is defined to be an algebra of the above type which satisfies the following axioms:

(a)	$a+(b+c) = (a+b)+c$	addition is associative,
(b)	$a+b = b+a$	addition is commutative,
(c)	$a+0 = a$	0 is a unit for addition,
(d)	$a\times(b\times c) = (a\times b)\times c$	multiplication is associative,
(e)	$a\times 1 = 1\times a = a$	1 is a unit for multiplication,
(f)	$a\times(b+c) = (a\times b)+(a\times c)$ $(a+b)\times c = (a\times c)+(b\times c)$	multiplication distributes over addition,
(g)	$a^* = 1+(a\times a^*) = 1+(a^*\times a)$.	

This treatment differs from earlier similar treatments (eg, [AHO74], [BAC75]) in that it does not assume the idempotency of addition, 0 is not assumed to be a multiplicative annihilator (ie., $a\times 0 = 0\times a = 0$ is not assumed), countable sums are not assumed to exist, and the closure is defined differently, although equivalently.

A *partial closed semiring* is defined to be an algebra of the above type where closure is only a partial function, and in which axioms (a) through (f) still always hold and axiom (g) holds whenever the closure is defined. It is then shown that any partial closed semiring S can be extended to a closed semiring $S\cup\{u\}$ where $u \notin S$ is defined to satisfy the following properties: $u+a = a+u = u$, $a\times u = u\times a = u$, $u^* = u$, and

$a^* = u$ if a^* was not previously defined. The new closed semiring is called the *completion* of S.

The operations of addition, multiplication, and closure are then extended to $n \times n$ matrices of elements from a closed semiring. If $A = [a_{ij}]_{i,j \in [1:n]}$ and $B = [b_{ij}]_{i,j \in [1:n]}$ are two such matrices, then

$$A + B = [a_{ij} + b_{ij}]_{i,j \in [1:n]}, \quad \text{and}$$

$$A \times B = [\sum_{k \in [1:n]} a_{ik} b_{kj}]_{i,j \in [1:n]}.$$

The closure of a matrix is defined inductively on the size n of the $n \times n$ matrix. If A is a 1×1 matrix, say $A = [a]$, then $A^* = [a]^* = [a^*]$. If A is an $n \times n$ matrix where $n > 1$, say

$$A = \begin{bmatrix} B & C \\ D & E \end{bmatrix}$$

where for some $0 < k < n$, B is a $k \times k$ matrix, C is a $k \times (n-k)$ matrix, D is an $(n-k) \times k$ matrix, and E is an $(n-k) \times (n-k)$ matrix, then

$$A^* = \begin{bmatrix} B^* + B^* C \Delta^* D B^* & B^* C \Delta^* \\ \Delta^* D B^* & \Delta^* \end{bmatrix}$$

where $\Delta = E + D B^* C$.

Lehmann shows that the above is a valid inductive definition, in particular that the definition of A^* is independent of the decomposition of the matrix A, and that with the two matrices of constants

$$0_n = [c_{ij}]_{i,j \in [1:n]} \quad \textit{with} \quad c_{ij} = 0 \text{ for } i,j \in [1:n], \quad \text{and}$$

$$I_n = [\delta_{ij}]_{i,j \in [1:n]} \quad \textit{with} \quad \delta_{ij} = \begin{cases} 1, & \text{if } i = j, \\ 0, & \textit{otherwise}, \end{cases}$$

the set of $n \times n$ matrices over a closed semiring only fails to be a closed semiring itself when the underlying semiring is the completion of a proper partial closed semiring, whence axiom (e) fails to hold (ie., it is not necessarily the case that

$A \times I_n = I_n \times A = A$). In particular, the closure operation as defined on matrices over closed semirings satisfies the iterative equation of axiom (g):

$$A^* = I_n + A \times A^*$$

The problem of finding the closure of a matrix over a closed semiring is the general Algebraic Path Problem.

The following are a few of the examples given by Lehmann which show the many useful instances of the Algebraic Path Problem:

(i) $<\{0,1\},\cup,\cap,T,0,1>$ where $T(0) = T(1) = 1$.

The closure A^* of a matrix A over this boolean semiring yields the reflexive and transitive closure of the relation represented by A.

(ii) $<\mathbf{R}\cup\{+\infty\},\min,+,Z,+\infty,0>$ where $Z(a) = 0$.

The closure A^* of a matrix A over this semiring yields the minimum-cost matrix for the directed weighted graph represented by A.

(iii) $<\mathbf{R}\cup\{+\infty\},\max,\min,\infty,0,+\infty>$ where $\infty(a) = +\infty$.

The closure A^* of a matrix A over this semiring yields the maximum capacity matrix for the network represented by A.

(iv) $<\mathbf{R}\cup\{u\},+,\times,s,0,1>$ where $s(a) = \frac{1}{1-a}$ for $a \neq 1,u$, and $s(1) = s(u) = u$.

The closure A^* of a matrix A over this semiring satisfies $A^* = (I-A)^{-1}$ if A^* does not contain the element u. Hence, the closure yields matrix inversion.

From this point, Lehmann proceeds to describe two algorithms for computing the closure of a matrix over an arbitrary semiring. The first algorithm is a generalization of the Gauss-Jordan Elimination algorithm, and is given in section 3.5; the second algorithm is a generalization of Gauss's algorithm, which is more advantageous over the Gauss-Jordan algorithm when 0 can be assumed to be a multiplicative annihilator, and the input matrix contains a large number of 0's. Lehmann also gives a description of a restricted semiring for which a generalization of Dijkstra's algorithm ([DIJ59]) can be employed to compute the closure of a matrix, which is a much more efficient algorithm for certain instances of the Algebraic Path Problem.

3.3. Tarjan's Approach

In [TAR81], Tarjan deals with the problem of solving path problems on directed graphs by firstly generating a collection of regular expressions representing sets of paths in the graph. He then describes mappings which take the problem from the realm of regular expressions into the domains for solving problems of shortest paths, systems of linear equations, and global flow analysis. In this sense, he shows that the problem of generating path expressions is the most general instance of the Algebraic Path Problem.

Let Σ be a finite alphabet with $\Sigma \cap \{\lambda, \varnothing, (,)\} = \varnothing$. A *regular expression* over Σ is an expression (ie., a string of symbols) constructed from the following rules:

(i) λ and $\varnothing$ are regular expressions.

(ii) For each $a \in \Sigma$, a is a regular expression.

(iii) If R_1 and R_2 are regular expressions, then so are $(R_1 \cup R_2)$, $(R_1 R_2)$, and $(R_1)^*$.

Every regular expression R defines a set $\sigma(R)$ of strings over Σ as follows:

(i) $\sigma(\lambda) = \{\lambda\}$; $\sigma(\varnothing) = \varnothing$.

(ii) $\sigma(a) = \{a\}$ for each $a \in \Sigma$.

(iii) $\sigma(R_1 \cup R_2) = \sigma(R_1) \cup \sigma(R_2)$; $\sigma(R_1 R_2) = \sigma(R_1)\sigma(R_2)$; $\sigma(R^*) = (\sigma(R))^*$.

A regular expression R is *simple* if $R = \emptyset$ or R does not contain $\emptyset$ as a subexpression.

If $G = (V, E)$ is a directed graph where V is the set of vertices of G, and E is the set of edges of G, then any path in G can be viewed as a string over E. A *path expression* P of *type* (u, v) is a simple regular expression over E such that every string in $\sigma(P)$ is a path from vertex u to vertex v.

Tarjan's approach to the Algebraic Path Problem is to define a mapping from regular expressions into each of the specific problem domains in which he is interested. By doing this, he shows that he can solve the concrete problems expressible as instances of the Algebraic Path Problem simply by generating path expressions. As an example, his approach to shortest path problems is as follows: given a weighted directed graph $G = (V, E)$ with weight (cost) function $c: E \rightarrow \mathbf{R}$, a shortest path from vertex u to vertex v is a path $p = \langle e_1, e_2, \cdots, e_k \rangle$ from u to v such that $\sum_{i=1}^{k} c(e_i)$ is minimized over all paths from u to v. He defines two mappings, *cost* and *shortest path*, as follows:

(i) $cost(\lambda) = 0$;
shortest path$(\lambda) = \lambda$.

$cost(\emptyset) = +\infty$;
shortest path$(\emptyset)$ = *none*.

(ii) $cost(e) = c(e)$ for $e \in E$;
shortest path$(e) = e$ for $e \in E$.

(iii) $cost(P_1 \cup P_2) = min(cost(P_1), cost(P_2))$;
shortest path$(P_1 \cup P_2) = [cost(P_1) \leq cost(P_2) \rightarrow$ *shortest path*(P_1),
$cost(P_1) > cost(P_2) \rightarrow$ *shortest path*$(P_2)]$.

$cost(P_1 \cdot P_2) = cost(P_1) + cost(P_2)$;
shortest path$(P_1 \cdot P_2)$ = *shortest path*(P_1).*shortest path*(P_2).

$$cost(P^*) = [cost(P) < 0 \rightarrow -\infty,\ cost(P) \geq 0 \rightarrow 0];$$
$$shortest\ path(P^*) = [cost(P) < 0 \rightarrow none,\ cost(P) \geq 0 \rightarrow \lambda].$$

If $P(u,v)$ is a path expression representing all paths from u to v, then

- if $cost(P(u,v)) = +\infty$, then there is no path from u to v;
- if $cost(P(u,v)) = -\infty$, then there are paths of arbitrarily small cost from u to v (*ie.*, there exists a cycle of negative cost in some path from u to v);
- if $cost(P(u,v)) \in \mathbf{R}$, then $shortest\ path(P(u,v))$ is a shortest path from u to v, and has cost $cost(P(u,v))$.

3.4. Rote's Approach

In his approach to the Algebraic Path Problem, Rote ([ROT83]) combines the axiomatic approach taken by Lehmann with the language theoretical approach of Tarjan. Like Lehmann, he describes the problem in terms of a general semiring structure, but he does not assume the existence of the closure operation. He then defines the Algebraic Path Problem in terms of a weighted graph with weights taken from the semiring.

Rote defines a semiring as a structure $<\mathbf{H},+,\times>$ with zero 0 and unity 1 which satisfies Lehmann's axioms (a) through (f) but not the closure axiom, but assumes that 0 is a multiplicative annihilator. Thus he makes $<H,+>$ a commutative semigroup with identity element 0, $<H,\times>$ a semigroup with identity element 1, multiplication ($\times$) distribute over addition ($+$), and zero absorptive with respect to multiplication. Given a weighted graph $G=(V,E)$ with weight function $\omega: E \rightarrow \mathbf{H}$, he defines M_{ij} to be the set of all (distinct) paths from vertex i to vertex j in G, and extends the weight function ω to be defined on paths in the graph G as follows: given a path $p = <\epsilon_{i_1,i_2}, \epsilon_{i_2,i_3}, \cdots, \epsilon_{i_{k-1},i_k}>$, where $\epsilon_{i,j}$ is an edge from vertex i to vertex j, the weight of the path is defined as

$$\omega(p) = \omega(\epsilon_{i_1,i_2}) \times \omega(\epsilon_{i_2,i_3}) \times \cdots \times \omega(\epsilon_{i_{k-1},i_k}).$$

Then the Algebraic Path Problem is the problem of finding the matrix $D=(d_{ij})$, where the entries of the matrix are defined as follows:

$$d_{ij} = \sum_{p \in M_{ij}} \omega(p).$$

Note that you can assume that the graph is complete, as you can adjoin missing arcs with associated weight 0. The weight of any path containing an arc of weight 0 will itself be 0, due to the assumption of the 0 annihilation axiom, so it will not contribute to the sum defining the elements of the matrix $D=(d_{ij})$.

As an alternate formulation, you can let $A=(a_{ij})$, where

$$a_{ij} = \begin{cases} \omega(\epsilon_{i,j}), & \text{if } \epsilon_{i,j} \in E; \\ 0, & \textit{otherwise}. \end{cases}$$

Then

$$A^2 = A \times A = (a_{ij}^{(2)}), \textit{ where } a_{ij}^{(2)} = \sum_{1 \le k \le n} a_{ik} \times a_{kj}.$$

$$A^3 = (a_{ij}^{(3)}), \textit{ where } a_{ij}^{(3)} = \sum_{1 \le k_1, k_2 \le n} a_{ik_1} \times a_{k_1 k_2} \times a_{k_2 j}.$$

And in general,

$$A^m = (a_{ij}^{(m)}), \textit{ where } a_{ij}^{(m)} = \sum_{\substack{p \in M_{ij}, \\ p \textit{ contains} \\ \textit{exactly } m \textit{ arcs}}} \omega(p).$$

Thus, if $D=(d_{ij})$ is the matrix of elements given by the Algebraic Path Problem, then

$$D = A^* = \sum_{m \ge 0} A^m = I + A + (A \times A) + (A \times A \times A) + \cdots.$$

Using these two equivalent formulations of the Algebraic Path Problem, Rote is able to show clearly how the general problem represents the problems of shortest paths, reflexive and transitive closure, and matrix inversion. These problems are realized as

follows:

(i) Consider the semiring

$$<\mathbf{R}\cup\{\pm\infty\},\min,+> \text{ with zero } \infty \text{ and unity } 0.$$

This is the set of extended reals under the operations of minimization and addition. From the first formulation of the Algebraic Path Problem, the solution $D=(d_{ij})$ is given by

$$d_{ij} = \sum_{p\in M_{ij}} \omega(p) = \min_{p\in M_{ij}} (\omega(p)),$$

where for a path $p=<\epsilon_1,\epsilon_2, \cdots \epsilon_k>$,

$$\omega(p)=\omega(\epsilon_1)+\omega(\epsilon_2)+ \cdots +\omega(\epsilon_k).$$

Hence the Algebraic Path Problem corresponds to the problem of computing the minimum distance (cost) matrix of a weighted directed graph.

(ii) Consider the semiring

$$<\{0,1\},\cup,\cap> \text{ with zero } 0 \text{ and unity } 1.$$

This is the Boolean semiring under the operations of max ($\cup$) and min ($\cap$). Again from the original formulation, the solution $D=(d_{ij})$ is given by

$$d_{ij} = \sum_{p\in M_{ij}} \omega(p) = \bigcup_{p\in M_{ij}} (\omega(p)),$$

where for a path $p=<\epsilon_1,\epsilon_2, \cdots \epsilon_k>$,

$$\omega(p) = \omega(\epsilon_1)\cap\omega(\epsilon_2)\cap \cdots \cap\omega(\epsilon_k)$$

$$= \begin{cases} 1, & \textit{if there exists the path } p \textit{ from } i \textit{ to } j, \\ 0, & \textit{otherwise}. \end{cases}$$

Hence in this case, the Algebraic Path Problem corresponds to the problem of computing the reflexive and transitive closure of the binary relation represented by the given directed graph.

(iii) Consider the semiring

$$<\mathbf{R},+,\times> \text{ with zero } 0 \text{ and unity } 1.$$

This is simply the set of real numbers **R** under the usual operations of addition and multiplication. This time, from the second formulation of the problem, the solution $D=(d_{ij})$ is given by

$$D = I+A+A^2+A^3+\cdots = (I-A)^{-1}, \quad \text{if } D \textit{ exists}.$$

Hence here the Algebraic Path Problem corresponds to the problem of computing the inverse of a real-valued matrix.

Rote points out that the most general instance of the Algebraic Path Problem is the problem of constructing the sets M_{ij} of paths in the graph from each vertex i to each vertex j, as was noted by Tarjan. This instance is realized by the first formulation of the problem using the following semiring, where $M = \bigcup_{1\le i,j\le n} M_{ij}$:

$$<\mathrm{M},\cup,\cdot> \text{ with zero } \varnothing \text{ and unity } \{\lambda\}.$$

This is the semiring consisting of the set of all paths between vertices in the directed graph, with its addition (+) and multiplication (×) operations defined to be the operations of (disjoint) union (∪) and path concatenation (·). Hence given $P_1, P_2 \in \mathrm{M}$,

(i) $P_1+P_2 = P_1 \cup P_2$; that is, the union of the sets P_1 and P_2 of paths; and

(ii) $P_1 \times P_2 = P_1 \cdot P_2$; that is, the set of all paths which are concatenations $p_1 \cdot p_2$ of paths $p_1 \in P_1$ and $p_2 \in P_2$.

The solution $D=(d_{ij})$ is given by

$$d_{ij} = \sum_{p\in M_{ij}} \omega(p) = \bigcup_{p\in M_{ij}} \omega(p) = \bigcup_{p\in M_{ij}} p = M_{ij},$$

since, for a path $p=<\epsilon_1,\epsilon_2,\cdots,\epsilon_k>$,

$$\omega(p) = \omega(\epsilon_1)\cdot\omega(\epsilon_2)\cdot\cdots\cdot\omega(\epsilon_k) = \epsilon_1\cdot\epsilon_2\cdot\cdots\cdot\epsilon_k = p.$$

3.5. General Solution

The general solution to the Algebraic Path Problem, that is, to the problem of constructing the sets M_{ij} of all (distinct) paths from each vertex i to each vertex j of a directed graph, can be computed using a generalized version of the Gauss-Jordan Elimination Algorithm, such as is given in Fig. 3-1. The algorithm operates in the following manner. First, let:

$M_{ij}^{(k)} =$ the set of all paths from vertex i to vertex j which contain only intermediate vertices x with $1 \leq x \leq k$, for each $k=0,1,2,\cdots,n$. By convention, let $\lambda \in M_{ii}^{(i)}$ for all i, but $\lambda \notin M_{ii}^{(i-1)}$.

Then

$$M_{ij}^{(0)} = \begin{cases} \{\epsilon_{ij}\}, & \text{if } \epsilon_{ij} \in E, \\ \varnothing, & \textit{otherwise}; \end{cases}$$

and

$$M_{ij} = M_{ij}^{(n)}.$$

The values of $M_{ij}^{(k)}$ for $k=0,1,2,\cdots,n$ can be viewed as successive approximations to the values of $M_{ij} = M_{ij}^{(n)}$. The algorithm then simply computes the sets $M_{ij}^{(k)}$ using the following recursive formula:

(i) $M_{ij}^{(k)} = M_{ij}^{(k-1)} + M_{ik}^{(k)} \times M_{kj}^{(k-1)}$, for $1 \leq k \leq n$, $k \neq i$, $k \neq j$.

A path in $M_{ij}^{(k)}$ either avoids vertex k, or it uniquely decomposes into two segments, the first going from vertex i to vertex k, and the second going from vertex k to vertex j with no occurrence of vertex k intermediate to the latter segment.

(ii) $M_{ij}^{(i)} = M_{ii}^{(i)} \times M_{ij}^{(i-1)}$, for $i \neq j$.

A path in $M_{ij}^{(i)}$ must pass through vertex i (recall that $\lambda \in M_{ii}^{(i)}$), and then continue on to vertex j without passing through vertex i again.

PROGRAM: Gauss-Jordan Elimination Algorithm

INPUT: $M_{ij}^{(0)} = \begin{cases} \{\epsilon_{i,j}\}, & \text{if } \epsilon_{i,j} \in E, \\ \varnothing, & \textit{otherwise}; \end{cases}$ where E is the edgeset of a directed graph.

OUTPUT: $M_{ij} = M_{ij}^{(n)}$; the set of all distinct paths from each vertex i to each vertex j.

PHASE 1:

```
for i:=1 to n do
for j:=1 to n do
    begin
        for k:=1 to min(i,j)-1 do
            M_ij^(k) := M_ij^(k-1) + M_ik^(k) × M_kj^(k-1);
        if i=j then M_ii^(i) := (M_ii^(i-1))*;
        if i>j then M_ij^(j) := M_ij^(j-1) × M_jj^(j)
    end;
```

PHASE 2:

```
for i:=1 to n do
for j:=1 to n do
    begin
        if i<j then M_ij^(i) := M_ii^(i) × M_ij^(i-1);
        for k:=min(i,j)+1 to max(i,j)-1 do
            M_ij^(k) := M_ij^(k-1) + M_ik^(k) × M_kj^(k-1);
        if i<j then M_ij^(j) := M_ij^(j-1) × M_jj^(j)
    end;
```

PHASE 3:

```
for i:=1 to n do
for j:=1 to n do
    begin
        if i>j then M_ij^(i) := M_ii^(i) × M_ij^(i-1);
        for k:=max(i,j)+1 to n do
            M_ij^(k) := M_ij^(k-1) + M_ik^(k) × M_kj^(k-1)
    end.
```

Fig. 3-1: Generalized Gauss-Jordan Algorithm for solving the Algebraic Path Problem.

(iii) $M_{ij}^{(j)} = M_{ij}^{(j-1)} \times M_{jj}^{(j)}$, for $j \neq i$.

A path in $M_{ij}^{(j)}$ must reach vertex j for a first time, and then follow some loop at vertex j (recall that $\lambda \in M_{jj}^{(j)}$).

(iv) $M_{ii}^{(i)} = (M_{ii}^{(i-1)})^*$.

A path in $M_{ii}^{(i)}$ uniquely decomposes into a unique number of partial paths from $M_{ii}^{(i-1)}$ (recall that $\lambda \notin M_{ii}^{(i-1)}$).

The algorithm in Fig. 3-1 computes the values of the $M_{ij}^{(k)}$'s using the above recurrences in three phases. The first phase calculates the values of $M_{ij}^{(j)}$ for $i \geq j$ and $M_{ij}^{(i-1)}$ for $i < j$; the second phase calculates the values of $M_{ij}^{(j)}$ for $i \leq j$ and $M_{ij}^{(i-1)}$ for $i > j$; and the third phase calculates the values of $M_{ij}^{(n)} = M_{ij}$. Notice that the order of the processing is arbitrary, as long as all (i', j') with $i' \leq i$, $j' \leq j$ are processed before (i, j) within any of the three phases. This fact makes the algorithm particularly suited for parallel processing.

There are only four types of (update) assignment statements in the algorithm, corresponding to the four parts of the recursive formula given above. These are as follows:

(i)	$c := c + a \times b$;	(inner product step)
(ii)	$c := c \times b$;	(right multiplication)
(iii)	$c := b \times c$;	(left multiplication)
(iv)	$c := c^*$.	(closure)

To solve the concrete problems enveloped by the Algebraic Path Problem, simply substitute the operations in the algorithm with the corresponding operations for the respective semirings for these problems, and replace the input initialization of $M_{ij}^{(0)}$ by

$$M_{ij}^{(0)} = \begin{cases} \omega(\epsilon_{i,j}), & \text{if } \epsilon_{i,j} \in E, \\ 0, & \textit{otherwise}. \end{cases}$$

The last assignment type mentioned above used by the algorithm is the closure operation, and is where the different approaches to the Algebraic Path Problem differ the greatest. According to Lehmann, the operation c^* would be performed by solving the iterative equation $x = 1 + c \times x$; Rote on the other hand defines the operation as $c^* = 1 + c + (c \times c) + (c \times c \times c) + \cdots$. In fact, these two definitions concur, and it is easy to see that the infinite sum, if it exists, is a solution to Lehmann's iterative equation. The problem with the infinite sum is that there is no guarantee that it will exist in general, and no general method for computing it. On the other hand, the iterative equation also has the problem that there is no general method of computing its solution, or even determining if a solution exists. [MAH84] deals with the problem of countable sums in semirings and the solution to the iterative equation.

In the three problem instances which Rote investigated, the closure operation is in fact easy to calculate. For the semiring $<\mathbf{R} \cup \{\pm\infty\}, \min, +>$ used to compute the minimum distance matrix of a directed graph, the closure operation on an element is given by

$$\begin{aligned} c^* &= 1 + c + (c \times c) + (c \times c \times c) + \cdots \\ &= \min(0, c, 2c, 3c, \cdots) = \begin{cases} 0, & \text{if } c \geq 0, \\ -\infty, & \text{if } c < 0. \end{cases} \end{aligned}$$

For the semiring $<\{0,1\}, \cup, \cap>$ used to compute the reflexive and transitive closure of a binary relation, the closure operation on an element is given by

$$\begin{aligned} c^* &= 1 + c + (c \times c) + (c \times c \times c) + \cdots \\ &= 1 \cup c \cup (c \cap c) \cup (c \cap c \cap c) \cup \cdots = 1. \end{aligned}$$

For the semiring $<\mathbf{R}, +, \times>$ used to calculate the inverse of a real-valued matrix, the closure operation on an element is given by the solution to the iterative equation $x = 1 + cx$, which gives

$$c^* = \frac{1}{1-c} \quad (\textit{for } c \neq 1).$$

There are problems with using the general solution given above on any specific instance of the Algebraic Path Problem. For instance, it was already noted above that there are superior methods for some special instances of the problem (*ie.*, Dijkstra's method in [DIJ59] for certain shortest path problems). Also, in the case of matrix inversion, the inverse of a matrix may exist which will not be found using the above algorithm. This is because the algorithm is a Gauss-Jordan Elimination algorithm which does not employ pivoting, so the algorithm may fail. In such a case, either the solution will have an undefined element in it (in the case of Lehmann's approach), or the computation may attempt to divide by zero (in Rote's Approach).

[ZIM81] treats the Algebraic Path Problem in detail, and shows how it can be utilized to solve the problem instances described above in this section,as well as several others which are defined in terms of different semirings, such as problems of path reliability and path capacity in a network, cut-set enumeration, schedule algebra, and k-shortest paths.

4. Kung Solution to LU Decomposition and Triangular Systems

The first solution described for some instance of the Algebraic Path Problem solves the problem of matrix inversion. The algorithm given in Fig. 3-1 has a natural interpretation in the realm of the problem of matrix inversion. The first phase of the algorithm computes the LU decomposition of the input matrix A, where L is a unit lower triangular matrix and U is an upper triangular matrix (Fig. 4-1*(a)*), with the exception that the diagonal elements of U are already inverted after phase 1 to be $c_{ii}^{(i)}=(c_{ii}^{(i-1)})^{-1}$ instead of $c_{ii}^{(i-1)}$. The second phase of the algorithm inverts the two triangular matrices L and U (Fig. 4-1*(b)*). Finally, the third phase of the algorithm multiplies the two matrices U^{-1} and L^{-1} to get the solution $A^{-1}=U^{-1}L^{-1}$ to the inversion problem (Fig. 4-1*(c)*).

In Section 2.3.2, a systolic system was described for computing the product of two matrices. In this section, two systolic systems will be described, one for solving the problem of factoring a matrix A into its LU decomposition, and the second for inverting triangular matrices. These three systems together will solve the problem of matrix inversion using a variant of the algorithm of Fig. 3-1, and can in fact be generalized to solve the general instance of the Algebraic Path Problem.

4.1. LU Decomposition

The following algorithm first appeared in [KUN78] and then later in [MEA80], and solves the problem of factoring a matrix A into lower and upper triangular matrices L and U on a hexagonally-connected systolic network of processors. Given an $n \times n$ matrix $A=(a_{ij})$, the triangular matrices $L=(l_{ij})$ and $U=(u_{ij})$ are computed using the following recurrence:

$$\begin{bmatrix} c_{11}^{(0)} & c_{12}^{(0)} & \cdots & c_{1n}^{(0)} \\ c_{21}^{(0)} & c_{22}^{(0)} & \cdots & c_{2n}^{(0)} \\ c_{31}^{(0)} & c_{32}^{(0)} & \cdots & c_{3n}^{(0)} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1}^{(0)} & c_{n2}^{(0)} & \cdots & c_{nn}^{(0)} \end{bmatrix} = \begin{bmatrix} 1 & & & \\ -c_{21}^{(1)} & 1 & & \bigcirc \\ -c_{31}^{(1)} & -c_{32}^{(2)} & & \\ \vdots & \vdots & \ddots & \\ -c_{n1}^{(1)} & -c_{n2}^{(2)} & \cdots & 1 \end{bmatrix} \begin{bmatrix} c_{11}^{(0)} & c_{12}^{(0)} & \cdots & c_{1n}^{(0)} \\ & c_{22}^{(1)} & \cdots & c_{2n}^{(1)} \\ & & & c_{3n}^{(2)} \\ & \bigcirc & \ddots & \vdots \\ & & & c_{nn}^{(n-1)} \end{bmatrix}$$

(a) $A = LU$ from phase 1

$$L^{-1} = \begin{bmatrix} 1 & & & \\ c_{21}^{(1)} & 1 & & \bigcirc \\ c_{31}^{(2)} & c_{32}^{(2)} & & \\ \vdots & \vdots & \ddots & \\ c_{n1}^{(n-1)} & c_{n2}^{(n-1)} & \cdots & c_{nn}^{(n-1)} \end{bmatrix} \quad \text{and} \quad U^{-1} = \begin{bmatrix} c_{11}^{(1)} & c_{12}^{(2)} & \cdots & c_{1n}^{(n)} \\ & c_{22}^{(2)} & \cdots & c_{2n}^{(n)} \\ & & & c_{3n}^{(n)} \\ & \bigcirc & \ddots & \vdots \\ & & & c_{nn}^{(n)} \end{bmatrix}$$

(b) L^{-1} and U^{-1} from phase 2

$$U^{-1}L^{-1} = \begin{bmatrix} c_{11}^{(n)} & c_{12}^{(n)} & \cdots & c_{1n}^{(n)} \\ c_{21}^{(n)} & c_{22}^{(n)} & \cdots & c_{2n}^{(n)} \\ c_{31}^{(n)} & c_{32}^{(n)} & \cdots & c_{3n}^{(n)} \\ \vdots & \vdots & \ddots & \vdots \\ c_{n1}^{(n)} & c_{n2}^{(n)} & \cdots & c_{nn}^{(n)} \end{bmatrix} = A^{-1}$$

(c) $A^{-1} = U^{-1}L^{-1}$ from phase 3

Fig. 4-1: The three phases of the Gauss-Jordan Elimination Algorithm of Fig. 3-1 for Matrix Inversion.

$$c_{ij}^{(0)} = a_{ij};$$

$$c_{ij}^{(k)} = c_{ij}^{(k-1)} + c_{ik}^{(k)} c_{kj}^{(k-1)} \qquad \text{for } 1 \leq k < min(i,j);$$

$$c_{ii}^{(i)} = \frac{1}{c_{ii}^{(i-1)}};$$

$$c_{ij}^{(j)} = c_{ij}^{(j-1)} c_{jj}^{(j)} \qquad \text{for } i > j.$$

This is precisely the computation performed in phase 1 of the algorithm described in

Fig. 3-1, as applied to matrix inversion. The matrices $L=(l_{ij})$ and $U=(u_{ij})$ are derived from the above recurrence by the following (see Fig. 4-1(*a*)):

$$l_{ij} = \begin{cases} 0, & \text{if } i<j, \\ 1, & \text{if } i=j, \\ -c_{ij}^{(j)}, & \text{if } i>j; \end{cases}$$

$$u_{ij} = \begin{cases} 0, & \text{if } i>j, \\ c_{ij}^{(i-1)}, & \text{if } i\leq j. \end{cases}$$

The systolic network described in Fig. 4-2 and Fig. 4-3 computes the LU decomposition of a 4×4 matrix. The original matrix $A=(a_{ij})=(c_{ij}^{(0)})$ is fed into the system from below, and the resulting matrix output at the top of the processor array is given by the following:

$$\begin{bmatrix} c_{11}^{(0)} & c_{12}^{(0)} & c_{13}^{(0)} & \cdots & c_{1n}^{(0)} \\ c_{21}^{(1)} & c_{22}^{(1)} & c_{23}^{(1)} & \cdots & c_{2n}^{(1)} \\ c_{31}^{(1)} & c_{32}^{(2)} & c_{33}^{(2)} & \cdots & c_{3n}^{(2)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_{n1}^{(1)} & c_{n2}^{(2)} & c_{n3}^{(3)} & \cdots & c_{nn}^{(n-1)} \end{bmatrix}$$

The triangular matrices $L=(l_{ij})$ and $U=(u_{ij})$ are interpreted using the above definitions of l_{ij} and u_{ij} (see Fig. 4-1(*a*)).

By the way the data enters the array in a staggered fashion, the last element c_{nn} to exit the array enters exactly $3(n-1)$ steps after the first element c_{11} enters the array, and it exits the array exactly n steps later. Hence this algorithm computes the LU decomposition of an $n\times n$ matrix A in $4n-3$ steps.

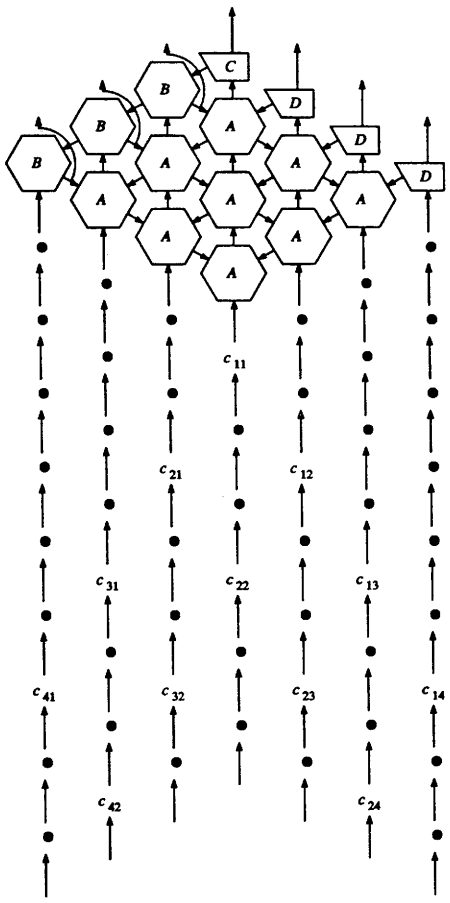


Fig. 4-2: Systolic System for computing the LU decomposition of a 4×4 matrix.

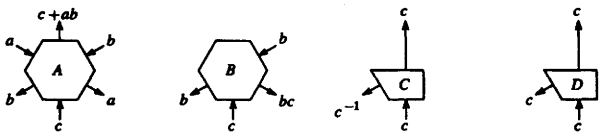


Fig. 4-3: Processor types for the systolic system of Fig. 4-2.

4.2. Triangular Systems of Equations

The problem of inverting a matrix can be reduced to the problem of solving several systems of linear equations. Given an $n \times n$ matrix A, to find the i^{th} column of the inverse matrix A^{-1}, you simply need solve the system of equations $Ax = \epsilon_i$, where $\epsilon_i = (0,\ldots,0,1,0,\ldots,0)^T$ is the n-tuple consisting of a 1 in the i^{th} position and zeroes everywhere else.

In the previous section, a systolic system was described for factoring a matrix A into its LU decomposition, $A = LU$, where L is unit lower triangular and U is upper triangular. In this section, we shall describe a systolic algorithm for solving triangular systems of linear equations. Using this algorithm, we can compute the inverses L^{-1} and U^{-1} of the triangular factors of the matrix A, and then use the matrix multiplication algorithm from section 2.3.2 to compute $A^{-1} = U^{-1}L^{-1}$.

The triangular system solver described here appeared with the LU decomposition algorithm described above in [KUN78] and in [MEA80], and solves the linear system of equations $Ax = b$, where A is an $n \times n$ lower triangular matrix, and $b = (b_1, b_2, \cdots, b_n)^T$ is an n-vector. (Clearly any upper triangular system of equations can be reformulated as a lower triangular system.) The solution $x = (x_1, x_2, \cdots, x_n)^T$ to this system can be computed using the following recurrence:

$$y_i^{(0)} = 0;$$

$$y_i^{(k)} = y_i^{(k-1)} + a_{ik}x_k \qquad \text{for } 0<k<i;$$

$$x_i = \frac{b_i - y_i^{(i-1)}}{a_{ii}}.$$

Fig. 4-4 depicts an example of a 4×4 lower triangular system of linear equations. The linearly-connected systolic array shown in Fig. 4-5 solves this system in the following fashion: The coefficients given in the matrix A are pumped into the systolic array from above, while the elements of the vector b are pumped into the left-end processor from below as shown.

$$\begin{bmatrix} a_{11} & & & \\ a_{21} & a_{22} & & \\ a_{31} & a_{32} & a_{33} & \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ b_4 \end{bmatrix}$$

Fig 4-4: Example lower triangular system of equations.

There are two types of processors in the array, the first type (the square processors) being inner product step processors, and the second type (the circular processor at the left end) being the processor which computes the final values of the solution elements in the vector x. The function of these two processors are depicted in Fig. 4-5. Each y_i flows through the array accumulating its inner product terms $a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{i,i-1}x_{i-1}$ and then at the special end processor, the value of x_i is defined by $x_i \leftarrow \frac{b_i - y_i}{a_{ii}}$.

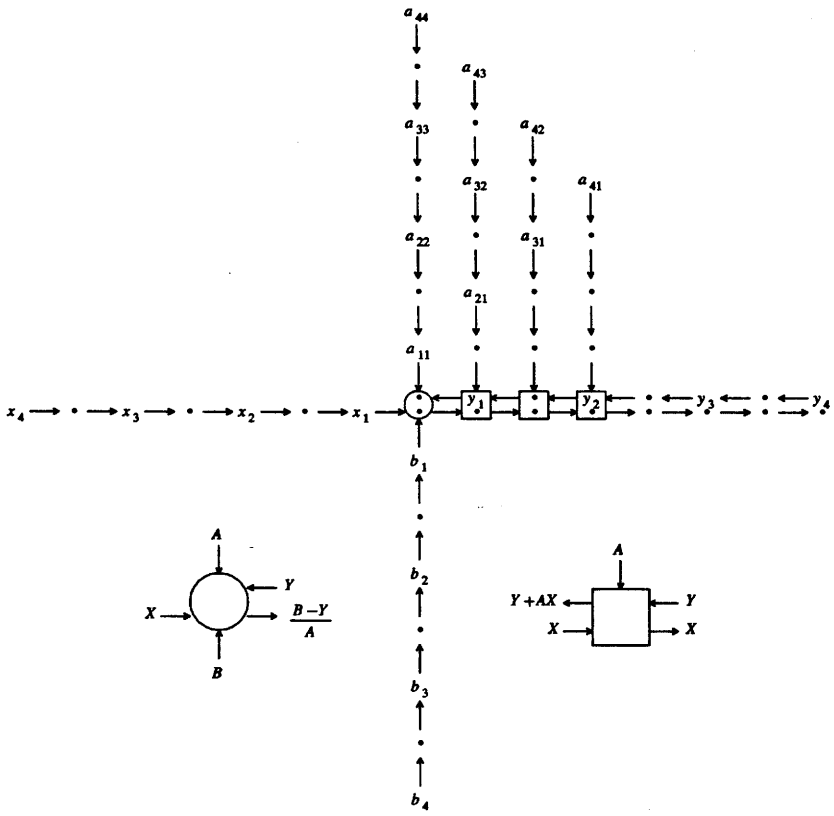


Fig. 4-5: Systolic System for solving a (4×4 lower) triangular system of linear equations.

4.3. Application to Triangular Matrix Inversion

The above systolic algorithm for solving triangular systems can be extended to an algorithm to solve several systems of linear equations $Ax=b_j$, where A is still a lower triangular matrix, by letting the different column vectors b_j in turn flow through a version of the linear array of processors which the coefficients of the A matrix flow through. By doing this, you are invariably solving the problem $AX=B$, where A is an

$n \times n$ lower triangular matrix, B is an $n \times k$ matrix, and the solution X is an $n \times k$ matrix. Hence you are solving $X = A^{-1}B$. Now simply by letting $B = I$, the $n \times n$ identity matrix, you are solving the problem $X = A^{-1}$.

Fig. 4-6 depicts the modified version of the systolic array to solve this extended problem.

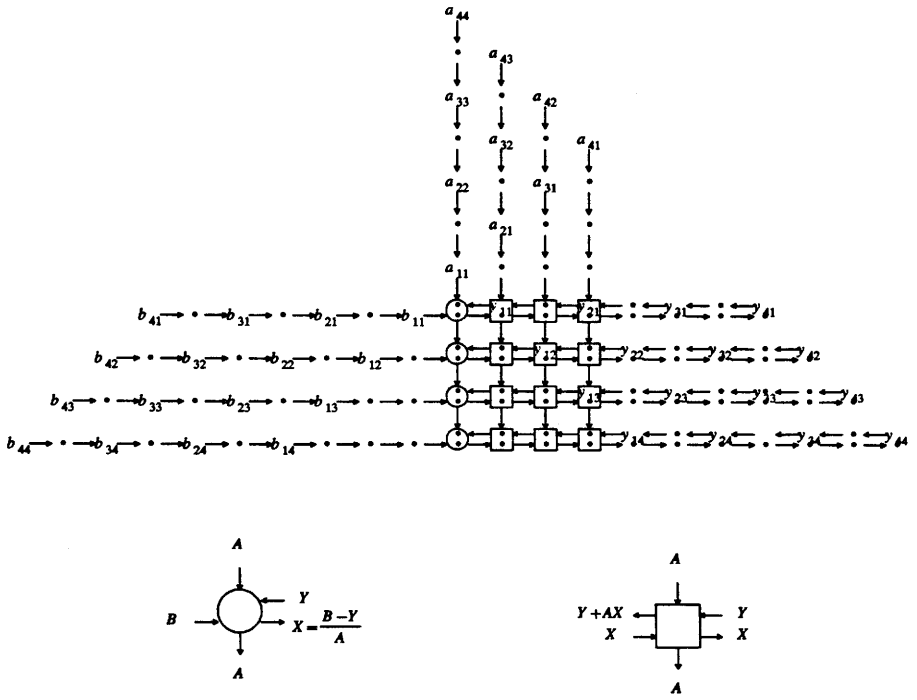


Fig. 4-6: Modified Systolic System for solving a set of (4×4 lower) triangular systems of linear equations.

It simply consists of several versions of the array depicted in Fig. 4-5 stacked on top of each other, with the column vectors $b_j=(b_{ij})$ (which in our case are the unit vectors ϵ_j) entering from the left, the coefficients of the A matrix entering from above, and the solution vectors coming out from the right side of the array. The actions of the two types of processors are the same as the actions performed by the corresponding processors in the array given in Fig. 4-5.

This modified systolic array can find the inverse of an $n\times n$ triangular matrix in $4n-3$ steps. Thus to compute the inverse of a general $n\times n$ matrix A using the LU decomposition algorithm and the triangular matrix inversion algorithm described in this section, along with the matrix multiplication algorithm described in section 2.3.2, you would need $3n-2$ steps to compute $A=LU$, and then $8n-6$ steps to compute L^{-1} and U^{-1}, and then $4n$ steps to compute $A^{-1}=U^{-1}L^{-1}$ (as L and U are band matrices with band widths $\omega_1=\omega_2=n$). Therefore the total cost of computing the inverse of an $n\times n$ matrix this way is $15n-8 = O(n)$.

5. Guibas, Kung, and Thompson Solution to Transitive Closure

The previous section described several early systolic algorithms which could be used together to solve a particular instance of the Algebraic Path Problem, the problem of matrix inversion. The algorithms themselves did not directly address the problem of matrix inversion. However, the solution we arrived at was asymptotically favourable. Using serial processing, the general solution to the Algebraic Path Problem can be computed using $O(n^3)$ operations (*cf*. Fig. 3-1). Using $O(n^2)$ processors in parallel, one can only hope to be able to solve the Algebraic Path Problem in linear time. The method described in the previous section indeed required only $O(n)$ time using $O(n^2)$ processors.

In this section, the first systolic algorithm designed specifically to solve some instance of the Algebraic Path Problem will be described. This algorithm, along with the other algorithms to be described later in this paper, uses $O(n^2)$ processors, and also meets our desired complexity result in solving its particular problem in linear time. The algorithm in this section was developed by Guibas, Kung and Thompson in [GUI79], and later appeared in greater detail in [ULL84]. It solves the problem of computing the transitive and reflexive closure of a binary relation, and can in fact be generalized somewhat to solve other instances of the Algebraic Path Problem such as computing the shortest distance matrix of a nonnegative-valued weighted graph, but it cannot for example solve the more general problem of matrix inversion.

Given a Boolean matrix $A=(a_{ij})$ which represents the adjacency matrix of a directed graph, the transitive and reflexive closure of A is the matrix $A^*=(a_{ij}^*)$, where a_{ij}^* is 1 if and only if there is a path of length zero or more from vertex i to vertex j in the graph. The problem of computing the transitive and reflexive closure of a directed graph was seen to be one instance of the Algebraic Path Problem (*cf*. Section 3).The array of processors depicted in Fig. 5-1 represents a systolic solution to the problem of computing the transitive and reflexive closure of an $n \times n$ Boolean matrix A. There are n^2 processors orthogonally connected in the network, with the processors along the right edge and bottom edge also connected to the processors at the left edge and top edge, respectively, so that the actual topology of the network is that of a torus. For each

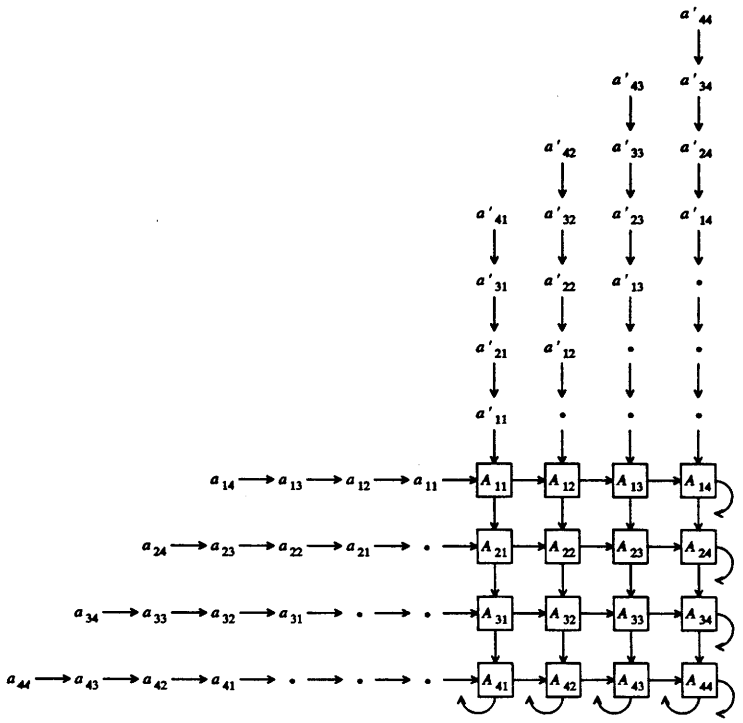


Fig. 5-1: Systolic array for computing the transitive closure of a 4×4 matrix $A=(a_{ij})$.

$0 \leq i, j \leq n$, a Boolean value A_{ij} is associated with the processor in the i^{th} row and j^{th} column of the array network. The value of A_{ij} is initially set to zero, and ultimately will contain the value of a_{ij}^*, so that the solution to the transitive and reflexive closure problem will be given by the values of the A_{ij}'s resident in the processors. Two copies of the input matrix A are passed through the processor array as shown, one from the left side of the array and the other from the top side of the array. When a value passes completely through the array and exits from either the right edge or bottom edge of the array, it is immediately fed back into the array from either the left edge or the top edge, respectively. After each element of the two versions of the input matrix have passed through the array three times, it no longer gets fed back into the array, but

rather just exits from the computation. When all elements have finished their three passes through the array, the resident values A_{ij} will represent the solution to the transitive and reflexive closure of the original array.

The computation performed at each processor is defined as follows: At any step of the computation, the two elements which meet at the processor associated with some A_{ij} will be a_{ik} and a_{kj}' for some value of k. At this point, the value of A_{ij} is updated by the assignment $A_{ij} \leftarrow A_{ij} \cup (a_{ik} \cap a_{kj}')$. This is motivated by the fact that if there is a path from vertex i to vertex k and a path from vertex k to vertex j, then there is a path from vertex i to vertex j. The a_{ij} and a_{ij}' values are generally passed straight through each processor unchanged. However, the values of the a_{ij}'s and the a_{ij}''s are also updated periodically throughout the computation. When a_{ij} passes through the processor associated with A_{ij}, the assignment $a_{ij} \leftarrow A_{ij}$ is performed. Similarly, when a_{ij}' passes through the processor associated with A_{ij}, the assignment $a_{ij}' \leftarrow A_{ij}$ is performed.

[ULL84] explains the algorithm in detail giving a full proof of the correctness of the algorithm. Basically, it shows that after one pass of the algorithm, the resident processor values A_{ij} contain the values $A_{ij}=a_{ij}^{(k)}$, for $k=\min(i,j)$, where $a_{ij}^{(k)}$ denotes the existence of a path from vertex i to vertex j using no vertex numbered higher than k interior to the path. After the second pass, the A_{ij}'s contain the values $A_{ij}=a_{ij}^{(k)}$, for $k=\max(i,j)$. Finally, after the third pass, the A_{ij}'s contain the values $A_{ij}=a_{ij}^{(n)}=a_{ij}^{*}$. These three passes roughly parallel the three phases of the Gauss-Jordan Elimination Algorithm given in Fig. 3-1, but the computaions are carried out somewhat differently. This transitive closure algorithm basically works on the following recurrence:

$$a_{ij}^{(0)} = a_{ij};$$

$$a_{ij}^{(k+1)} = a_{ij}^{(k)} \cup (a_{ik}^{(k)} \cap a_{kj}^{(k)});$$

$$a_{ij}^{*} = a_{ij}^{(n)}.$$

Here, $a_{ij}^{(k)}$ represents the existence of a path from vertex i to vertex j which does not have any interior vertices numbered higher than k. Comparing the above recurrence to the recurrence developed in Section 3.5 to derive the Gauss-Jordan Elimination

Algorithm for the general Algebraic Path Problem of path enumeration, it is easy to see that this recurrence could not be utilized for the problem of path enumeration; the sets M_{ij}, defined in Section 3.5 as the sets of all distinct paths from each vertex i to each vertex j of a directed graph, could not be computed using the above recurrence, as the sets constructed would not contain just distinct paths, but would also have within them multiple copies of the same paths. For this reason, this transitive closure algorithm cannot be generalized to solve the most general instance of the Algebraic Path Problem, nor can it be generalized even to solve the problem of matrix inversion.

The above transitive closure algorithm can in fact however be modified to solve the problem of computing shortest distances between vertices in a weighted directed graph, where the arcs of the graph have nonnegative weights, which as was seen in Section 3 is another instance of the Algebraic Path Problem. The input matrix $A=(a_{ij})$ is given by the weights of the arcs between vertices i and j. If $i=j$, then $a_{ij}=0$. If there is no arc from vertex i to vertex j, then $a_{ij}=+\infty$. The resident processor values A_{ij} are all initialized to $+\infty$. These values are updated using the assignment $A_{ij} \leftarrow min(A_{ij}, a_{ik}+a_{kj}')$, motivated by the fact that a path from vertex i to vertex k appended to a path from vertex k to vertex j may be shorter than the shortest path from vertex i to vertex j so far discovered. The values of a_{ij} and a_{ij}' are updated in exactly the same fashion as in the transitive closure algorithm.

By observation of the systolic array depicted in Fig. 5-1, the last elements of the input matrix to be entered into the array, the a_{nn} and a_{nn}' elements, enter the array after exactly $2(n-1)$ steps, and they exit the array after the third and final pass through the array exactly $3n$ steps later. Thus the algorithm takes $5n-2$ steps to compute the solution to the transitive closure problem being solved, and have the solution resident in the processor array. The solution then needs to be unloaded from the processor array, which takes another n steps to complete. Hence this algorithm takes in total $6n-2$ steps to compute the transitive closure of a binary relation (or the shortest distance matrix of a directed graph with nonnegative weights on its arcs).

6. Kramer and van Leeuwen Solution to Matrix Inversion

The next systolic algorithm developed for solving an instance of the Algebraic Path Problem was the systolic matrix inversion algorithm of Kramer and van Leeuwen ([KRA83]). This algorithm uses an $n \times n$ array of processors connected in an orthogonal pattern to compute the inverse of an $n \times n$ matrix using a parallelization of the Gauss-Jordan Elimination algorithm, and again succeeds in accomplishing the task in linear time. It improves on the method of inverting matrices in Section 4 though as it is a uniform solution, and takes only $6n-4$ steps to complete,as opposed to the previous method which takes $15n-8$ steps to complete.

Basically, the algorithm simply follows the usual Gauss-Jordan Elimination algorithm for inverting matrices as outlined in [PEA67] as follows: Given an $n \times n$ matrix $A=(a_{ij})$, to compute A^{-1}, first extend A to an $n \times 2n$ matrix $(A\ I)$, and perform the following algorithm.

```
for i ←1 to n do
    begin
        multiply row i by a_ii^-1;
        for j ←1 to n do
            if j ≠ i then subtract a_ji times row i from row j
    end.
```

This will transform the matrix $(A\ I)$ into the matrix $(I\ B)$ via elementary matrix transformations, so that the matrix B will in fact be the inverse matrix A^{-1}. [KRA83] presents the simple example given in Fig. 6-1; the original matrix A is juxtaposed to the left of the identity matrix of the same size, and after the algorithm outlined above is performed, the inverse matrix A^{-1} can be read off of the right block of the matrix.

[KRA83] notes that the way to look at the above algorithm is as a two-phase process, the first phase working in a horizontal direction, and the second phase working in a vertical direction. For the first phase, you pass the value of a_{11}^{-1} along the first row, multiplying each element in the row by it, while at the same time passing the values of a_{j1} along their respective rows to broadcast their values to the elements of their rows. For the second phase, you pass the values of the top row elements a_{1j} down their

$$\left(\begin{array}{ccc|ccc} 2 & 0 & 2 & 1 & 0 & 0 \\ 1 & -1 & 1 & 0 & 1 & 0 \\ 0 & 1 & \frac{1}{2} & 0 & 0 & 1 \end{array}\right) \rightarrow \left(\begin{array}{ccc|ccc} 1 & 0 & 1 & \frac{1}{2} & 0 & 0 \\ 0 & -1 & 0 & -\frac{1}{2} & 1 & 0 \\ 0 & 1 & \frac{1}{2} & 0 & 0 & 1 \end{array}\right) \rightarrow$$

$$\left(\begin{array}{ccc|ccc} 1 & 0 & 1 & \frac{1}{2} & 0 & 0 \\ 0 & 1 & 0 & \frac{1}{2} & -1 & 0 \\ 0 & 0 & \frac{1}{2} & -\frac{1}{2} & 1 & 1 \end{array}\right) \rightarrow \left(\begin{array}{ccc|ccc} 1 & 0 & 0 & \frac{3}{2} & -2 & -2 \\ 0 & 1 & 0 & \frac{1}{2} & -1 & 0 \\ 0 & 0 & 1 & -1 & 2 & 2 \end{array}\right)$$

Fig. 6-1: Example of the Gauss-Jordan Elimination Algorithm for matrix inversion.

respective columns to perform the row subtraction step of the algorithm, using the values of a_{j1} broadcast along row j in the first phase. This works for the first row in the algorithm (*ie.*, for $i=1$), but for the rest of the rows, the action is identical, using the following trick: when you pass the a_{j1} along the rows in the first phase, you can at the same time exchange their values with the values which they pass, to move the second column to the first column, the third column to the second column, and so on through to the $(2n)^{th}$ column to the $(2n-1)^{st}$ column, and then leave the values of a_{j1} in the $(2n)^{th}$ column. Then in the second phase, you can similarly exchange rows as you pass the values of a_{1j} down the columns, to move the second row to the first row, the third row to the second row, and so on through to the $(2n)^{th}$ row to the $(2n-1)^{st}$ row, and then leave the values of a_{1j} in the $(2n)^{th}$ row. This makes the new value of a_{11} to be the value which would be a_{22} in the original matrix, so rather than performing the function of the above algorithm for $i=2$, you simply perform it once again for $i=1$. Repeatedly employing this exchanging trick, you can simply perform the function of the algorithm n times with the value $i=1$, rather than once for each $i=1,2,3,\cdots,n$. This provides the regularity needed to perform the algorithm within the framework of a systolic network. Fig. 6-2 shows the action performed on the matrix $(A\ I)$ using this strategy.

$$\left(\begin{array}{ccc|ccc} 2 & 0 & 2 & 1 & 0 & 0 \\ 1 & -1 & 1 & 0 & 1 & 0 \\ 0 & 1 & \frac{1}{2} & 0 & 0 & 1 \end{array}\right) \rightarrow \left(\begin{array}{ccc|ccc} -1 & 0 & -\frac{1}{2} & 1 & 0 & 0 \\ 1 & \frac{1}{2} & 0 & 0 & 1 & 0 \\ 0 & 1 & \frac{1}{2} & 0 & 0 & 1 \end{array}\right) \rightarrow$$

$$\left(\begin{array}{ccc|ccc} \frac{1}{2} & -\frac{1}{2} & 1 & 1 & 0 & 0 \\ 1 & \frac{1}{2} & 0 & 0 & 1 & 0 \\ 0 & \frac{1}{2} & -1 & 0 & 0 & 1 \end{array}\right) \rightarrow \left(\begin{array}{ccc|ccc} \frac{3}{2} & -2 & -2 & 1 & 0 & 0 \\ \frac{1}{2} & -1 & 0 & 0 & 1 & 0 \\ -1 & 2 & 2 & 0 & 0 & 1 \end{array}\right)$$

Fig. 6-2: Modified Gauss-Jordan Elimination Algorithm.

It is apparent from Fig. 6-2 that the action of this modified algorithm does not affect the structure of the right block of the augmented matrix; it remains the identity matrix, while the actual inverted matrix A^{-1} is computed in the left block over top of the space where the original matrix A was placed. Hence, we can simply ignore the right block and work solely with the left block, that is, just with the matrix A itself.

As described, the algorithm makes n passes to the right, interleaved with n passes downward, which performed naively would require $2n^2$ steps to complete. However these passes can actually be pipelined, so that several passes are in progress at any given time. The downward pass could begin at an element in the matrix immediately after the rightward pass has gone by, and the next rightward pass can begin at that element immediately after that downward pass has gone by. To achieve this pipelining, the rightward waves and the downward waves must be skewed, so that the rightward wave in one row passes one step before the same wave passes in the row below it, and likewise that the downward wave in one column passes one step before the same wave passes in the next column. Fig. 6-3 depicts several steps of the skewed motion of the pipelined algorithm, showing the waves of the two-phase cycles of the algorithm (a rightward wave followed by a downward wave) separated by dashed lines.

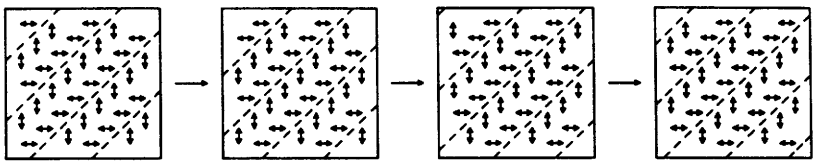


Fig. 6-3: Skewed wavefronts of the pipelined algorithm.

The systolic algorithm presented in [KRA83] executes this pipelined algorithm on an $n \times n$ orthogonal array of processors with each processor having data streams coming in from and going out to each of its four neighbouring processors. Each processor has two registers, one to hold an element of the matrix being processed, and the other to hold values which pass through the processor. Thus, the processor in the i^{th} row and the j^{th} column of the array would hold the value of a_{ij} of the matrix in one register, and the value of a_{i1} in the other when the value of a_{i1} passes through the processor on the rightward pass of the algorithm, to use it in the row subtraction step of the downward pass of the algorithm. [KRA83] describes the function of each of the processors, describing their actions in terms of four states of the computation, corresponding on whether the processor is communicating with the processor to its left, the processor to its right, the processor above it, or the processor below it, depending on the position of the waves with respect to the processor. The action which a processor performs is exactly what occurs in the pipelined algorithm performed on the matrix being inverted, as described above, but is complicated to define, as it cycles through four phases, doing four different computations, and because the processors are not homogeneous; there are nine different processor types to consider: one for each of the four corners, one for each of the four edges, and one for the interior processors.

Returning back to Fig. 6-3, you can see that the wave fronts are exactly four steps apart, and that a wave takes $2(n-1)$ steps to pass completely through the matrix from the upper left corner to the lower right corner. The algorithm starts with the first wave front originating at the upper left corner of the matrix (processor array), and continues until the n^{th} wave has passed completely through the matrix to the lower right corner of the matrix. After $4n-2$ steps, the downward pass of the n^{th} wave enters the matrix,

and after $2(n-1)$ more steps, it has pass completely over the matrix. Hence this systolic algorithm computes the inverse of an $n \times n$ matrix in $6n-4$ steps, using n^2 processors.

This algorithm was the first systolic algorithm directly applied to the problem of matrix inversion. The method described in Section 4 merely combined several different algorithms to achieve the goal of inverting a matrix. [KRA83] only applied this algorithm to the problem of inverting matrices, but in fact it is a true parallelization of the Gauss-Jordan Elimination algorithm, and as such can be generalized to solve the other problems engulfed by the definition of the Algebraic Path Problem.

7. Rote Solution to the General Algebraic Path Problem

Up until now, we have only considered systolic algorithms which were applied only to some subproblem of the Algebraic Path Problem, either the problem of matrix inversion or the problem of computing transitive and reflexive closure. [ROT83] is the first treatment of a systolic algorithm for solving the general Algebraic Path Problem, that is, for executing the algorithm given in Fig. 3-1 in its most general setting, so that it could be applied to any problem enveloped by the definition of the Algebraic Path Problem. The algorithm appearing here is described using the concepts defined in Section 3, *ie.*, the general definitions of the operations of addition (+), multiplication ($\times$), and closure (*). To interpret them to solve a given subproblem of the Algebraic Path Problem, it is only necessary to interpret these operations as they are defined for the given subproblem.

Unlike the algorithms described in the two previous sections which use orthogonal arrays, [ROT83] returns to using a hexagonal array of processors to solve the problem. Fig. 7-1 shows the array used for processing a 4×4 matrix. To process an $n\times n$ matrix, you would use an $(n+1)\times(n+1)$ systolic array of hexagonal processors similar to the array used for matrix multiplication and for LU decomposition. In fact, most of the processors in the array, the interior (A) processors, perform the same inner product step as the processors in the previous matrix algorithms. The operations performed by the various processor types are described in Fig. 7-1. They all perform simple operations, either corresponding to one of the four types of assignment statements as described in the algorithm in Section 3, or simply an identity operation, whereas the processor just passes its input on unaltered. The type A processor performs the inner product calculation $c' = c + a\times b$, where a, b, and c are the input values for the processor. The type B_I processor either calculates just the identity function, if it is receiving input from below (that is, if input is entering into the array), or it calculates the function $b' = a\times b$, where a and b are the input values for the processor, if it is receiving input from the diagonals. The type B_O processor calculates the function $c' = c\times b$, where b and c are the input values for the processor. The type C processor calculates the closure operation $c' = c^*$ where c is the input value to the processor. The type D_I processor simply calculates the identity function, either passing on the value it receives from

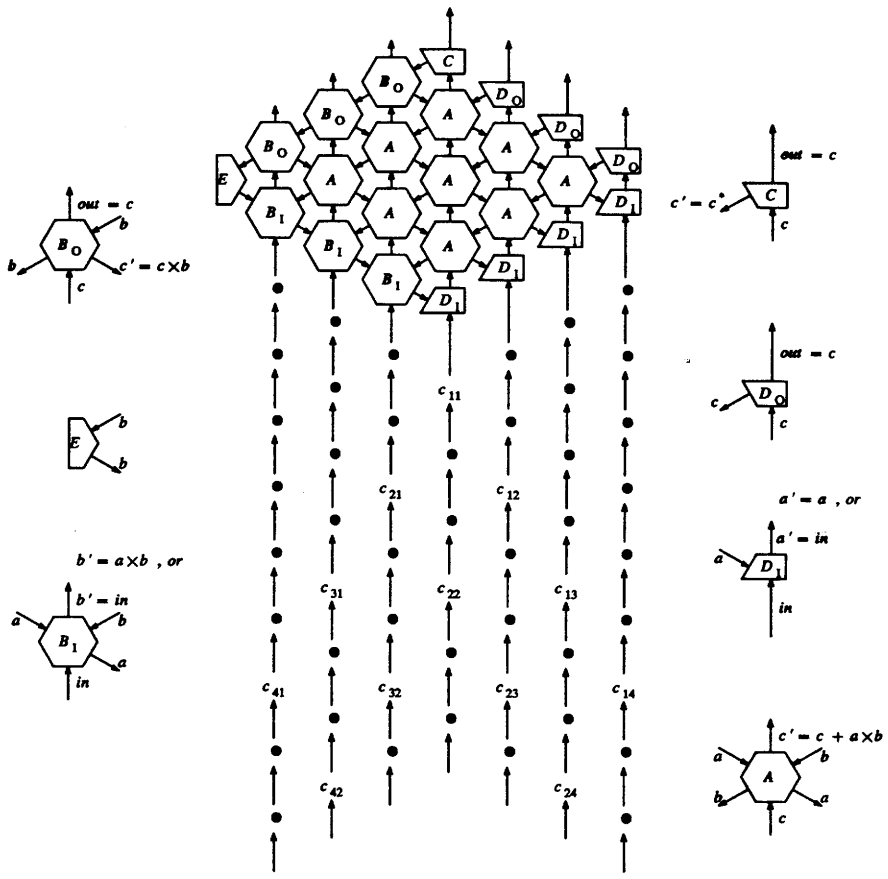


Fig. 7-1: Systolic Array for solving the Algebraic Path Problem.

below, if input is entering the array, or the value it receives from the diagonal. Finally, the type D_O and type E processors simply calculate the identity function, passing on the values they receive unaltered.

Initially, the processors are assumed to be cleared to zero. The initial array $C = M^{(0)}$ is input in a staggered fashion from below as shown in Fig. 7-1, and each element c_{ij} follows a set path through the array of processors, changing its value along the

way according to the algorithm given in Fig. 3-1, and then flows out at the top of the array as an element of the matrix which defines the solution to the Algebraic Path Problem.

The path followed by an element c_{ij} of the input matrix is as follows: first the element enters the array from below and travels upward until it hits the top of the array. The element will then reflect either down and to the right, if it hit the top left edge of the array, or else down and to the left, if it hit the top right edge of the array. The element will continue travelling through the array until it hits a bottom edge, whence it will reflect and travel straight up the array again. Upon reaching a top edge for a second time, the element will again reflect down through the array as before, and when it hits a bottom edge again, it will reflect and travel straight up the array once more. This time, when it hits the top of the array, it will just pass straight through the processor and become part of the output solution, rather than reflecting again. The paths of three typical elements are depicted in Fig. 7-2.

From looking at Fig. 7-2, it is clear that the path followed by an element is exactly $3(n-1)$ steps longer than if it were to travel straight through the array. Hence the output at the top of the array is of the same shape and organization as the input at the bottom. The first element input into the array, c_{11}, is output $4n+1$ steps after it is input, and the last element input into the array, c_{nn}, is input exactly $3n$ steps after the first element is input, so the algorithm described using this array of processors takes $7n-2$ steps to complete, from the time the first element enters the array, until the time the last element exits the array. Hence any problem which can be formulated as an Algebraic Path Problem can be solved using this systolic algorithm in $7n-2$ steps.

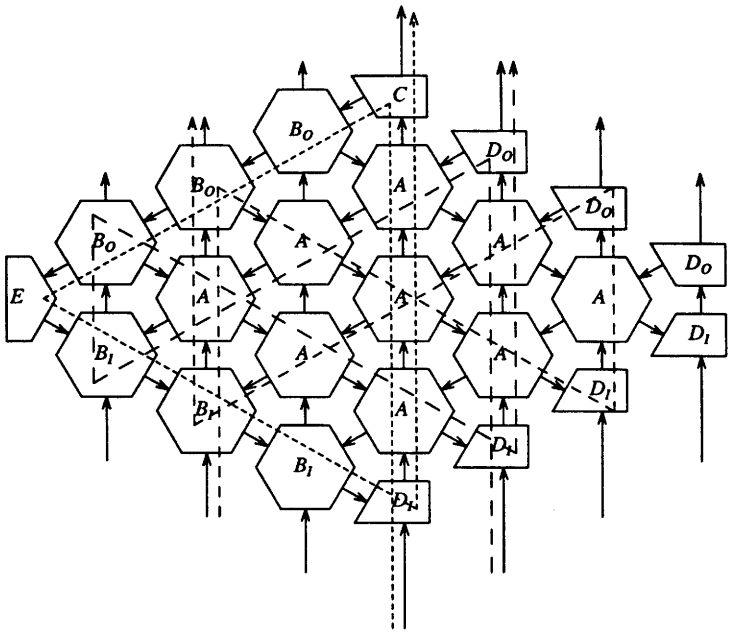


Fig. 7-2: Paths taken by three matrix elements.

7.1. Analysis and Improvements

In Section 2.3 of this paper, when we were analyzing examples of systolic algorithms, we saw in the cases of matrix-vector multiplication and matrix-matrix multiplication that we could run several tasks on the same systolic array to achieve a better average running time for our algorithms. This was done to make better use of the processors, as they often are not used all at the same time when processing a single task. Such improvement can be attained in the present algorithm being considered.

First of all, you can use a form of pipelining similar to that used by the matrix multiplication system, where you have three $n \times n$ matrices being processed simultaneously. Since the elements of a matrix are input into the array staggered three steps

apart, you can enter the elements of a second matrix immediately after the corresponding elements of the first matrix, and follow the second matrix in a similar fashion by a third matrix. In this way, you can process three matrices at the same time, and have the processors occupied by some element at every step of the computation.

Once a matrix is started being processed by the array, the processing continues for $7n-2$ steps. However, you need not wait until the processing of the first matrix is completely finished before starting to process another. The first element of a new matrix can enter the array at the bottom processor immediately after the last element of the old matrix has left this processor for the last time. Hence, if you are processing three matrices simultaneously, you can begin processing another three matrices exactly $6n$ steps after the first three matrices started being processed. Hence, three matrices are processed (that is, $3n^3$ assignments are performed, corresponding to the $M_{ij}^{(k)}$ computed in the algorithm of Fig. 3-1 for each value of i,j,k) for every $6n$ steps of the computation by the n^2 processors in the array which actually do the assignments (A,B_I,B_O,C). The maximum number of assignments which can be done by the n^2 processors in $6n$ steps is $6n^3$, so the processors are being used to exactly one half utilization. The other half of the time, the processors are passing on their inputs unaltered.

To see how we can attain further improvement, note that update assignments are only performed when the elements are travelling up the array. When an element enters the array, it may travel up through the first few processors unaltered, but at some point the processors will start altering the element, and the element will continue being altered at each processor as it travels in an upward direction, until it again reaches the first processor which altered its value, at which point its value will have been changed n times, and so it will be unaltered from that point on. With this in mind, you can see that a second matrix can be entered into the array exactly $3n$ steps after the first matrix was entered. An element of the second matrix will join the corresponding element of the first matrix for its first pass up the array. The new element will be passed unaltered by the processors and the old element will be changed for the first few steps in the upward pass. However at some point, the old element will reach the processor which first altered its value and will thus continue up the array unaltered from that point on. But this is the exact point at which the new element starts having its value

altered.

Using the above pipelining ideas, we are able to attain 100% utilization of the processors in the array, whereas we started off with only very sparse utilization. With these tricks employed in our systolic system, we can finish processing an $n \times n$ matrix with every n steps. Clearly this is the best possible result you can hope for with n^2 processors, as using the algorithm of Fig. 3-1, n^3 update assignments must be performed.

There is a more serious problem with this systolic array, as well as for all of the other systolic arrays for processing matrices which we have discussed in this paper, than that of efficient processor use. Presumably, we would like to process matrices of various sizes, but it appears at the moment that we would need a different array for each different size of matrix that we would like to process. The other methods never discuss this problem, but [ROT83] outlines some ways of getting around this problem with respect to the systolic system described in this section.

One solution to this problem is found by noting that by using an array for processing an $n \times n$ matrix, you can in fact use that array to process any $m \times m$ matrix for $m \leq n$. Fig. 7-3 gives a skeletal description of the data flow of a matrix being processed in an oversized array. In viewing this diagram, the following points can be noted. The cluster at the top of the array corresponds to phase 1 of the algorithm, where the LU decomposition of the matrix is performed, in the matrix inversion setting of the problem; the L matrix travels down and to the right and the U matrix travels down and to the left. This part of the computation is simply the LU decomposition system described in Section 4.1. The cluster on the sides correspond to phase 2 of the algorithm, where the L and U triangular matrices are inverted. They correspond precisely to the triangular matrix inversion system described in Section 4.3, although it is now formulated on a hexagonal array rather than an orthogonal array as before. The cluster at the bottom corresponds to phase 3 of the algorithm, and is simply the matrix multiplication system described in Section 2.3.2. The U^{-1} matrix enters the system from the upper left and the L^{-1} matrix enters from the upper right. These two matrices are multiplied together to give the inverse of the matrix in question, which is then passed up and out of the system. Hence, the system being described in this section is really not a new

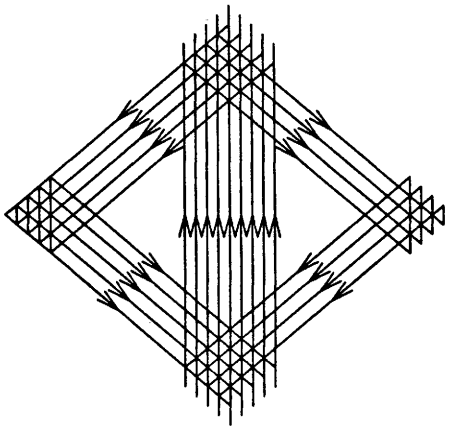


Fig. 7-3: Outline of the data flow of the elements of a matrix through an oversized array.

system, but a combination of previous systems, as was described in Section 4.

There are still problems with this solution to processing various sizes of matrices. If you only have one array of processors, what size should it be? On the one hand, if you make its size n too small, the array will be useless for processing many matrices, whose size is $m>n$. If on the other hand you make its size n very large, then it will be inefficient for processing matrices of size $m<<n$, as it is clear that the processing time of a matrix of size m is proportional to the size of n; in fact it would take exactly $4n+3m-2$ steps to compute.

One attempt at solving this problem is by using an array of a reasonable size n, and then for matrices of size $m>n$, using techniques of linear algebra to transform the matrix into a matrix in block diagonal form. In this way you can process the blocks separately, as depicted in Fig. 7-4.

A better approach to solving this problem is by using the technique of *folding* ([CHO83]). You can fold the array along the diagonal as shown in Fig. 7-5, and have the center (ie, right edge) reflect the values which encounter it.

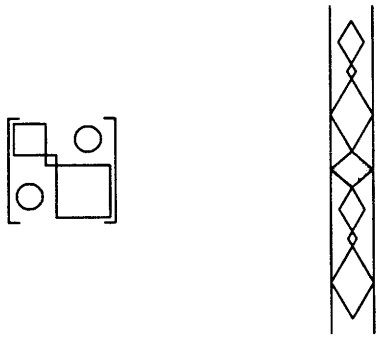


Fig. 7-4: Processing the blocks of a large matrix.

Using this type of array, you need not input the matrix at the right, but rather input it at the extreme left, so that it only uses as much of the array as necessary. A signal can be propagated up with the first element to mark the right edge of the subarray being used, so that the elements can be reflected at that edge. With this, you get a (potentially semi-infinite) wedge-shaped array of processors, which eliminates the problem of trading off the size of the array (ie, the maximum size of matrix which can be processed) with the efficiency of processing smaller matrices; It is clear that $m \times m$ matrices are again processed in $7m-2$ steps. An example of using such an array is given in Fig. 7-6, where a 3×3 matrix is being processed.

The skeletal diagram depicted in Fig. 7-7 shows more clearly how different sized matrices can be processed in the array. This diagram also shows how you can process varying sized matrices most efficiently. You can still begin processing one matrix before another has completely finished being processed.

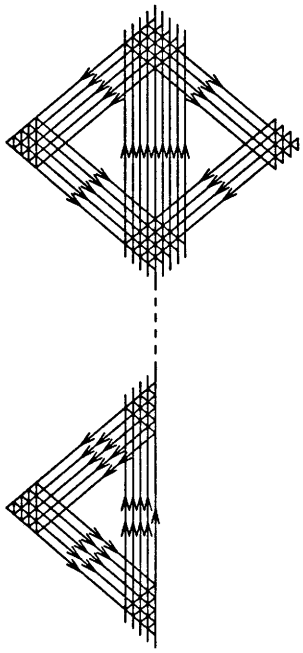


Fig. 7-5: Folding of the systolic array.

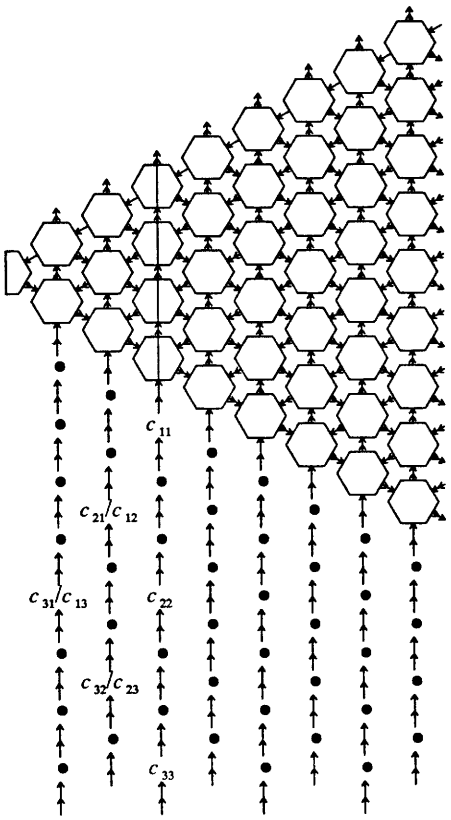


Fig. 7-6: Processing a 3×3 matrix on a wedge-shaped array of processors. A signal is propagated up the array with the c_{11} element to define the right edge of the portion of the array to be used.

Fig. 7-7: Processing varying-sized matrices.

8. Discussion

Several systolic systems have been described for solving both the general Algebraic Path Problem, as well as certain subproblems of it. After developing a solution to the general problem, the systolic algorithms which have been discovered for solving the problem were outlined in chronological order. The algorithm developed for solving the general problem consists of n^3 assignment statements, and thus would require $O(n^3)$ steps to execute on a sequential machine. All of the systolic systems described for solving the problem consist of $O(n^2)$ processors running in parallel, and they all succeed in solving the problem (or some subproblem) in linear time, which makes them all equivalent and optimal in that sense. However, some of the algorithms are less desirable due to a high constant in the linear number of steps it required, or because of the complexity of the operations of the individual processors.

The first method described was that of combining the systolic systems of Kung for computing the LU decomposition of a matrix, inverting triangular matrices, and multiplying matrices. This method, although sufficient, is undesirable due to the fact that it is so disjointed. Using this method, you have to process your matrix through three different arrays of processors to solve the problem. Because of this, it requires $15n-8$ steps to process a matrix, which is more than twice the number of steps needed using any of the later systems.

The next method described was that of Guibas, Kung and Thompson. This was the first system developed which actually addressed some subproblem of the Algebraic Path Problem. It uses an orthogonal array of processors, and requires $6n-2$ steps to compute the solution. Unfortunately, the system is restricted to only a subset of subproblems of the Algebraic Path Problem. It was developed as a system for computing the transitive closure of a binary relation, but in fact it could be used for some other subproblems, such as the problem of determining the shortest paths between vertices of a weighted directed graph whose edges are weighted with nonnegative values. However, it cannot solve the Algebraic Path Problem in its fullest generality, and in fact cannot even solve the problem of matrix inversion. Also, it requires that the leftmost processors be connected to the rightmost processors, and that the top processors be

connected to the bottom processors, so that the actual topology of the system is that of a torus. Furthermore, it requires that three versions of the input matrix exist and pass through the array, one to remain resident in the processors, one to pass horizontally through the array, and one to pass vertically through the array.

The next method described was that of Kramer and van Leeuwen. Like the previous method, it operates on an orthogonal array of processors, and solves its problem in $6n-2$ steps. However, although it was only described as a solution to matrix inversion by Kramer and van Leeuwen, it can in fact be generalized to solve any subproblem of the Algebraic Path Problem. Also, it does not require the end processors to be connected to form a toroidal topology, nor does it require three copies of the matrix to be flowing through the matrix. In this sense, it is superior to the previous method. However, the problem with this system lies in the complexity of the functions which the individual processors have to compute. First of all, there are nine different types of processors which have to be dealt with. Secondly, each of the processors goes through a four-phase cycle in which it in turn interacts with the processor to its left, the processor to its right, the processor above it, and the processor below it. Also, there must be some type of counter incorporated into the system, as the algorithm requires a set action to occur precisely n times when processing an $n \times n$ matrix.

The final method described was that of Rote. This method returned to the hexagonal array of processors which is used for other matrix computations, and because of its simplicity suggests that hexagonal arrays are the most suited for matrix computations. The previous method of Kramer and van Leeuwan which uses an orthogonal processor array actually simulates a third direction of data flow through the array as the wavefronts described in the method travel diagonally through the array from the top left to the bottom right. As was noted in Section 2., a hexagonal array of processors is simply an orthogonal array with diagonal connections added in one direction, so this diagonal motion would not have to be simulated on a hexagonal array. Rote actually had in mind the solution of the general Algebraic Path Problem when he developed his system, and indeed his system does solve the general problem. It does involve a somewhat nontrivial data flow through the array, but all of the processor functions are simply defined and executed. Most of them perform nothing more than a multiplication or an

inner product step; all of the others simply perform the identity operation on their arguments, passing their inputs straight through unaltered. The system processes an $n \times n$ matrix in $7n-2$ steps, which is actually a little worse than the previous two methods. However, Rote also presented a unidirectional variation of his system that can solve the problem in $5n-2$ steps, which he proves is optimal when you have only one version of the input matrix flowing through the processor array. This variation was not included in this paper as it requires a nonregular data flow with various matrix elements reflecting at different places within the interior of the processor array. Also because of this, processor functions are no longer fixed for the processors, as for example the closure operation occurs at various places in the array at various times. Hence the improvement in the number of steps required to solve a problem does not seem to justify the greater complexity in the data flow and processor actions.

It was decided that the first method, that of combining the systolic arrays for solving the problems of computing the *LU* decomposition of a matrix, inverting a triangular matrix, and multiplying two matrices, was a poor approach to solving the Algebraic Path Problem. However, the method of Rote seems like the best method. Five years separated the developments of these systolic systems, during which time the other systems for solving subproblems of the Algebraic Path Problem were developed. In fact, Rote's method is precisely the combination of the three systolic systems developed in the beginning, as was discovered in Section 7. By looking again at the *LU* decomposition processor array depicted in Fig. 4-2, you can see that it is precisely the system developed five years later by Rote. The data flow was simply stopped prematurely after the *LU* decomposition was computed. It took that long for the *LU* decomposition system to be analyzed closely enough to realize that it could in fact be made to execute the whole Gauss-Jordan Elimination algorithm for matrix inversion, and in fact for the general Algebraic Path Problem.

Bibliography

[AHO74] Aho, A.V., J.E. Hopcroft, J.D. Ullman, The Design and Analysis of Computer Algorithms, *Addison-Wesley*, (1974).

[BAC75] Backhouse, R.C., B.A. Carré, *"Regular Algebra Applied to Path-finding Problems"*, Journal Inst. Maths Applics 15 (1975), pp161-186.

[CAR71] Carré, B.A., *"An Algebra for Network Routing Problems"*, J. Inst. Maths Applics 7 (1971), pp273-294.

[CHO83] Choffrut, C., K. Culik II, *"Folding of the Plane and the Design of Systolic Arrays"*, Information Processing Letters 17 (1983), pp149-153.

[CUL84] Culik II, K., Fris, I., *"Topological Transformations as a Tool in the Design of Systolic Networks"*, Department of Computer Science, University of Waterloo, Research Report CS-84-11 (1984).

[DIJ59] Dijkstra, E.W., *"A Note on Two Problems in Connection with Graphs"*, Numer. Math. 1 (1959), pp269-271.

[FLO62] Floyd, Robert W., *"Algorithm 97: Shortest Path"*, Comm. ACM 5 (1962), p345.

[GUI79] Guibas, L.J., H.T. Kung, C.D. Thompson, *"Direct VLSI Implementation of Combinatorial Algorithms"*, Proceedings of Caltech Conference on VLSI (1979), pp509-525.

[KAN78] Kant, Rajani M., Takayuki Kimura, *"Decentralized Parallel Algorithms for Matrix Computation"*, Fifth Annual Symposium on Computer Architecture, Conference Proceedings (1978), pp96-100.

[KLE56] Kleene, S.C., *"Representation of Events in Nerve Nets and Finite Automata"*, Annals of Mathematics Studies 34 - Automata Studies, C.E. Shannon, J. McCarthy (eds), (1956), pp3-41.

[KRA83] Kramer, M.R., J. van Leeuwen, *"Systolic Computation and VLSI"*, Foundations of Computer Science IV, Part 1, Algorithms and Complexity, J.W. DeBakker, J. van Leeuwen (eds), (1983), pp75-103.

[KUN78] Kung, H.T., C.E. Leiserson, *"Systolic Arrays (for VLSI)"*, Sparse Matrix Proceedings (Symposium on Sparse Matrix Computation, Knoxville, Tennessee), I.S. Duff, G.W. Stewart (eds), (1978), pp256-282.

[KUN79] Kung, H.T., *"Let's Design Algorithms for VLSI Systems"*, Proceedings of Caltech Conference on VLSI (1979), pp65-90.

[KUN80] Kung, H.T., *"The Structure of Parallel Algorithms"*, Advances in Computers Vol. 19 (1980), M.C. Yovits (ed), pp65-103.

[KUN82] Kung, H.T., *"Why Systolic Architecture"*, Computer Magazine 15 (1982), pp37-46.

[LEH77] Lehmann, Daniel J., *"Algebraic Structures for Transitive Closure"*, Theoretical Computer Science 4 (1977), pp59-76.

[LEI79] Leiserson, C.E., *"Systolic Priority Queues"*, Proceedings of Caltech Conference on VLSI (1979), pp199-214.

[LEV72] Levitt, K.N., W.H. Kautz, *"Cellular Arrays for the Solution of Graph Problems"*, Comm. ACM 15 (1972), pp789-801.

[MAH84] Mahr, B., *"Iteration and Summability in Semirings"*, Annals of Discrete Mathematics 19 - Combinatorial and Algebraic Methods in Operations Research, R.E. Burkard, R.A. Cuninghame-Green, U. Zimmermann

(eds), (1984), pp229-256.

[MEA80] Mead, C.A., L. Conway, Introduction to VLSI, *Addison-Wesley*, (1980).

[PEA67] Pease, Marshall C., *"Matrix Inversion Using Parallel Processing"*, Journal ACM 14 (1967), pp757-764.

[ROT83] Rote, Günter, *"A Systolic Array for the Algebraic Path Problem"*, Technische Universitat Graz, Institut fur Mathematik, Austria (1983).

[ROY59] Roy, B., *"Transitivité et Connexité*, C.R. Acad. Sci. 249 (1959), p216.

[SAM78] Sameh, A.H., D.J. Kuck, *"On Stable Parallel Linear System Solvers"*, Journal ACM 25 (1978), pp81-91.

[TAR75] Tarjan, Robert Endre, *"Solving Path Problems on Directed Graphs"*, Stanford University Report STAN-CS-75-528 (1975).

[TAR81] Tarjan, Robert Endre, *"A Unified Approach to Path Problems"*, Journal ACM 28 (1981), pp577-593.

[TAR81B] Tarjan, Robert Endre, *"Fast Algorithms for Solving Path Problems"*, Journal ACM 28 (1981), pp594-614.

[ULL84] Ullman, J.D., Computational Aspects of VLSI, *Computer Science Press* (1984).

[WAR62] Warshall, Stephen, *"A Theorem on Boolean Matrices"*, Journal ACM 9 (1962), pp11-12.

[ZIM81] Zimmermann, U., *"Linear and Combinatorial Optimization in Ordered Algebraic Structures"*, Annals of Discrete Mathematics 10 (1981).