A Survey of Sequential and Systolic Algorithms for the Algebraic Path Problem

Eugene Fink
School of Computer Science
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213, USA
eugene@cs.cmu.edu

under the supervision of
Jo C. Ebergen

Research Report

Abstract

This report summarizes some results on the *algebraic path problem*, and describes different sequential and *systolic* algorithms for solving this problem.

A *systolic algorithm* is a parallel algorithm where computations are performed by a planar mesh of connected processors. All processors perform the same algorithm and every processor exchanges data only with its neighbours.

The *algebraic path problem* is the problem of performing a special unary operation, called *closure*, over a square matrix, whose entries are elements of a semiring. This problem generalizes some matrix problems (such as computing the inverse of a real-valued matrix), graph problems (e.g. transitive closure and reduction, shortest path, the stochastic communication network problem), and regular language problems (e.g. the proof of the correspondence between regular expressions and finite automata).

We describe two different approaches to an algebraic path problem and four special cases of the general problem: computing the closure of a matrix over a Dijkstra semiring, the transitive closure problem, the shortest path problem, and computing the inverse of a real-valued matrix. For each problem we present both sequential and systolic algorithms. Also, we describe a history of sequential and systolic solutions of each problem.

To Linda Wood

Acknowledgement

I owe many thanks to Jo Ebergen under whose supervision I wrote this report. The report would not have been possible without his untiring direction, support, and inspiration. Jo Ebergen advised me on the presentation of the material and improving the style of the report. He painstakingly read the entire document, gave many comments, and helped me to find and correct inaccuracies.

Contents

Chapter 1

Introduction

1.1 The algebraic path problem

Matrix computations and problems of finding paths in networks are among the most important computational problems. Researchers tried to solve them from the very beginning of the Computer Age, when computers just had come to existence and were mostly viewed as simple number mills. Algorithms for many such problems were developed even earlier, in pre-computer medieval ages. People tried to solve these problems on analog machines or just by hand and sometimes found very efficient solutions. The first algorithms for matrix multiplication, matrix inverse, and solving systems of linear equations were designed in the time of Gauss. Research on routing problems was launched by travel agencies and telephone companies in the mid Forties, and some algorithms for analog machines were found before von Neumann constructed his digital computer [von Neumann, 1945].

With the development of computers, more and more efforts were applied to finding efficient solutions for both matrix and graph problems, and hundreds of papers were written on almost every problem of this kind during the first two decades of the Computer Age (1950–1970). For example, according to the statistics of [Pape, 1974], more than 200 articles on the shortest path problem had been published before 1972. As new parallel models of computations were developed, researchers tried to solve the old problems on parallel machines, and sometimes even created new models of computations specially designed for matrix computations. In some sense, systolic arrays were born

this way: H. Kung developed a formal model of VLSI computations designed to solve matrix and combinatorial problems.

In the mid Fifties researchers began to notice that matrix methods may be used to solve routing problems. Shimbel, Bellman, and Kalaba used matrix elimination techniques for designing efficient routing algorithms[1].

An old rule states:

> The existence of analogies between central features of various theories implies the existence of a general theory which underlies the particular theories and unifies them with respect to those central features. (E. H. Moore. *Introduction to a Form of General Analysis*, New Haven Mathematics Colloquium, Yale University Press, 1910.)

And the purpose of science is to find these central features. So, having noticed the similarities between matrix and graph problems, scientists began to look for a general theory that would lead to universal methods for solving all these problems.

The first small success in this direction was achieved in 1956 by Ford and Fulkerson, who showed the isomorphism between two graph problems (the shortest path and network capacity problems), and two years later described a more general problem, which included the first two problems as special cases. However, the generalization remained in the area of graph algorithms, and did not show any connection between graph and matrix problems.

The next, *big* success, was achieved by Dijkstra, who presented the first algorithm for solving the transitive closure and shortest path problems by computing the closure of a matrix over a special kind of semiring, later called a *Dijkstra semiring*. Dijkstra's algorithm was not only a direct ancestor of the algebraic path problem, but also presented an efficient solution for both single-source and all-pairs transitive closure and the shortest path problem, three years earlier than Warshall's transitive closure algorithm and Floyd's all-pairs shortest path algorithm were published. It is unfortunate to see that, as time passes, people remember Dijkstra's algorithm less and less,

[1] At present I am purposefully avoiding technical details such as describing the problems solved by these researchers, mentioning the techniques used, and referring to particular publications. All these details will be described, mentioned, and referred to in the following chapters, while the introduction is intended to be informal.

while Warshall's and Floyd's algorithms, which are less general and not more efficient than Dijkstra's solution, become more and more famous.

Finally, the work of Carre, Lehmann, and several other researchers led to formulating the algebraic path problem, the most general form of which was stated in 1977. The problem is concerned with a special algebraic structure, called a *closed semiring*, with three operations on the elements of this structure, generalized addition, generalized multiplication, and a unary operation called *closure*, similar to the inverse operation in the usual arithmetic. These operations are defined not only on single elements but also on square matrices over a closed semiring. The *algebraic path problem* is the problem to compute the closure of a matrix over an arbitrary closed semiring.

The problem described a general setting for solving a lot of different matrix and graph problems, and also problems on regular languages. Scientists were pleased to learn that so many important problems may be solved by developing a single algorithm for the algebraic path problem, and since 1977 much work has been done to design efficient algebraic path algorithms. In almost all cases, algorithms were developed by modifications of old methods, initially designed for special cases of the algebraic path problem, such as Warshall's transitive closure algorithm and the Gauss-Jordan matrix inversion method.

Of course, the formulation of the general problem did not make research on its special cases, such as transitive closure or matrix inversion, completely meaningless, because special cases often allow us to design more efficient solutions than the solution of the general problem by using some unique properties of each special case. For example, while designing a transitive closure algorithm, we may use non-matrix representations of a graph, and while solving matrix inversion, we may use commutativity of addition (in a general semiring addition is not necessarily commutative). However, the general solutions of the algebraic path problem are surprisingly efficient, and most special cases of the algebraic path problem do *not* have more efficient solutions than the general problem. The only exceptions are transitive closure and reduction, which may be solved more efficiently than other instances of the algebraic path problem.

The running time of all known algebraic path algorithms is $O(N^3 \cdot T)$, where N is the size of an input matrix and T is the time necessary to perform generalized addition, multiplication, and closure over elements of a semiring (it may not be constant time). It is not known whether this running time is

optimal, and no non-trivial lower bound is known. Ironically, the lower bound for the time complexity is not known for *any* instance of the algebraic path problem, even for simplest and most well-studied of them, such as matrix inversion and transitive closure.

1.2 Systolic arrays

Now let us describe informally systolic-array computations, which we are going to use for solving the algebraic path problem and its instances. Systolic arrays were introduced in 1978 by H. Kung and Leiserson as a theoretical model for VLSI parallel computations. The purpose was to design a model of computations such that

- The model obeys restrictions that allow an efficient hardware implementation.
- In spite of these severe restrictions, it is still able to solve some useful problems.
- The model is theoretically well-formalized to allow theoreticians to design algorithms and prove their correctness without thinking of hardware.

Hardware chips are always flat and have a regular structure, and so an efficiently implementable model must have the same properties: it has to be a planar regular mesh of processors. Since the speed of data exchange between processors is in reverse proportion to their distance from each other, the communication between neighbouring processors is much more efficient than between distant processors. H. Kung and Leiserson took this into account, and allowed data exchange only between neighbours in a mesh. And thus we have come to the definition of a systolic array:

> A *systolic array* is a regular planar (or linear) mesh of processors where each processor may exchange data only with neighbouring processors, but neither with distant processors, nor with the main memory.

Systolic arrays did not come from absolutely nowhere, and were not something conceptually new. Similar computational models, called cellular automata, were already known in the early Seventies[2]. However, systolic arrays better satisfied the three requirements stated above than the previous models, and for this reason they quickly became popular among computer scientists.

Systolic arrays are especially well fit for designing matrix and graph algorithms, as if they had been purposefully designed for these algorithms (and probably they were). The very first problems solved on systolic arrays all were special cases of the algebraic path problem. No wonder that researchers tried to design the general algebraic path algorithm for systolic arrays, and the first success was achieved by Rote, who presented a systolic solution of an almost general algebraic path problem in 1985.

Within the year after Rote's publication several independent efficient solutions of the general problem were found, and much research in this area has been performed until nowadays. Many different systolic arrays for the algebraic path problem have been designed. Different issues of this problem were considered, such as computing the transitive closure on systolic arrays of restricted size and on systolic arrays with some additional computational restrictions. Also, results on the lower bound of the space-time complexity of the problem have been obtained, and optimal solutions have been designed based on these results.

1.3 Overview of the report

We begin Chapter 2 with historical notes on the algebraic path problem. Then we present two different approaches to formulating the problem, and describe a less general problem stated in 1959 by Dijkstra. Finally, we give several examples of closed semirings that show how the algebraic path problem may be applied to specific graph and matrix problems.

In Chapter 3, we describe sequential solutions of the general algebraic path problem and three of its special cases, transitive closure, shortest path, and matrix inversion. We proceed in the same order as researchers proceeded while developing algorithms: we first show a solution for a special case and then methods to generalize this solution to the algebraic path problem. Then

[2]See for example [A. Smith, 1971] and [Kautz and Levitt, 1972].

we present Dijkstra's algorithm for solving single-source graph problems. Historical notes are presented for each special case of the algebraic path problem.

Chapter 4 introduces the notion of systolic arrays and describes the systolic solutions of all problems discussed in Chapter 3. We again use the method of going from special-case solutions to algorithms for the general algebraic path problem. We present historical notes on the development of systolic arrays, and on systolic solutions for each of the discussed problems.

Conclusions and some general remarks are presented in Chapter 5. Also, in this chapter we give references to articles on non-systolic parallel algorithms, for the reader who is interested in studying the subject further.

Chapter 2

Description of the problem

2.1 Historical notes

The history of the algebraic path problem began in 1956, when Kleene published his book on finite automata and regular languages [Kleene, 1956]. Kleene's proof that every regular language can be represented by a regular expression was the first algorithm for the algebraic path problem. However, Kleene did not realize that there was a connection between his proof and graph problems, nor that the proof contained an algorithm.

In the same year, Ford and Fulkerson demonstrated the duality between the network capacity problem and the shortest path problem [Ford and Fulkerson, 1956]. This probably was the first attempt to formalize similarity between different graph problems. They further developed their model in [Ford and Fulkerson, 1958], where they described a general flow problem, which included the shortest path and network capacity problems as special cases. However, this generalization was concerned only with graphs and did not describe the connection between graph and matrix algorithms.

In 1958 Bellman adapted the Jacobi method (see [Varga, 1962]) of matrix computations for solving the shortest path problem [Bellman, 1958], and Kalaba showed that the same method may be used to solve stochastic routing problems [Kalaba, 1958]. These two results were further steps toward the algebraic path problem, for they showed that the same technique may be used for several different matrix and graph problems.

In 1959 Dijkstra published an algorithm for finding the closure of a ma-

trix over a Dijkstra semiring. Even though a Dijkstra semiring was a less general structure than the semirings suggested later by Carre, Yoeli, and Lehmann, Dijkstra's algorithm generalized Roy's transitive closure algorithm (Roy, 1959) and both the single-source and all-pairs shortest path algorithms [Dijkstra, 1959]. It is interesting to notice that while algorithms for more and more general models of the algebraic path problem have been suggested, the efficiency of Dijkstra's original algorithm has not been improved, and today the algebraic path problem algorithms still have the same running time as Dijkstra's algorithm.

Unfortunately, at present Dijkstra's article is almost forgotten, and seldom mentioned in the publications on the algebraic path problem. Warshall's algorithm [Warshall, 1962], which is often referred to as the first algorithm on a transitive closure problem and whose name is given to half of the algebraic-path algorithms, was published three years later than Dijkstra's algorithm, was less general and had the same efficiency as Dijkstra's algorithm.

In 1961 Yoeli described a structure called *Q-semirings* [Yoeli, 1961], which was more general than Dijkstra semirings and could be used to design the algebraic-path algorithm. However, Yoeli was a mathematician and did not design algorithms. While the results presented by Yoeli were correct, his proofs and some definitions were not quite correct, because he used infinite sums without properly defining them.

A year later Ford and Fulkerson made one more step toward the algebraic-path algorithm by demonstrating how to use the Gauss-Seidel method for solving the shortest-path problem [Ford and Fulkerson, 1962]. The Gauss-Seidel method was originally designed for operations over matrices (see [Varga, 1962]), and thus the work of Ford and Fulkerson showed the similarity between solutions of graph and matrix problems.

An attempt to design a general algebraic setting for solving systems of linear equations and path problems in graph theory was made in [Carre, 1971]. Carre described *regular algebras*, algebraic structures close to Yoeli's Q-semirings. While Carre's structures allow us to solve a wider range of problems than Q-semirings, nobody knows whether Q-semirings are a special case of Carre's algebras. It seems that they are not. Neither Carre himself nor any other researcher has tried to compare these two structures in terms of their generality. Carre showed that the problem to find the closure of a matrix over a regular algebra generalizes several graph and matrix problems, and adapted two methods (Jacobi and Gauss-Seidel) for computing this closure.

Carre's publication is well-known and sometimes (mistakenly) referred to as the first publication on algebraic structures for the algebraic path problem.

The work of Carre was further developed in [Aho *et al.*, 1974]. Backhouse and Carre first showed the similarity between algebraic path algorithms and Kleene's proof of the correspondence between regular languages and regular expressions. Their article [Backhouse and Carre, 1975] presents an algebraic structure that generalizes both previous models for the algebraic path problem and regular languages. Both publications of Carre and the discussion on the algebraic path problem in [Aho *et al.*, 1974] inherited Yoeli's inaccuracy of not properly defining infinite sums. Some results on the algebraic path problem and algorithms for solving it were presented in [Tarjan, 1975] and [Tarjan, 1976]. Tarjan considered the *single-source algebraic path problem*, that is the problem of finding the shortest paths from some fixed vertex to all other vertices.

Finally between 1977 and 1981 three different approaches to the algebraic generalization of path problems, matrix problems, and regular languages stemmed from the work described above.

The first approach, suggested by Lehmann, generalizes graph problems to performing a special unary operation, called the *closure* operation, on a matrix over a semiring [Lehmann, 1977]. Lehmann compiled all the work on the generalized graph and matrix problems published before, corrected all inaccuracies, and slightly generalized the previous models. He formally proved the correctness of presented models and algorithms.

The second approach, developed throughout several papers ([Carre, 1971], [Backhouse and Carre, 1975], [Aho *et al.*, 1974], [Kam and Ullman, 1976], [Fredman, 1976], [D. Johnson, 1977], [Zimmermann, 1981], [Gondran and Minoux, 1984], and some others), and most clearly summarized in [Rote, 1985] and [Cormen *et al.*, 1990], does *not* replace a graph with a matrix, but treats the weights of edges of a graph as elements of an arbitrary semiring. This approach is slightly less general than Lehmann's approach. A third, quite different approach was developed by Tarjan [Tarjan, 1981a], [Tarjan, 1981b], who continued work on comparison of graph problems and regular languages, started by Backhouse and Carre. Tarjan treats a graph as a finite automaton, and a path as a string in the corresponding regular language. Then he shows that the set of all paths between two vertices may be expressed as a regular expression in this language, and defines a function from regular expressions to an arbitrary semiring. Tarjan has presented an algorithm for computing

this function for any regular expression. Tarjan's approach generalizes not only path problems in directed graphs, but also Kleene's work on the correspondence between regular languages and finite automata [Kleene, 1956]. We do not discuss Tarjan's approach in this paper.

Below we present first two approaches to defining the algebraic path problem and compare them with each other.

2.2 Definition of a semiring

A *semiring* is a structure $(S, \oplus, \otimes, \bar{0}, \bar{1})$, where S is a set, $\oplus$ (a generalized addition) and $\otimes$ (a generalized multiplication) are binary operations on S, and $\bar{0}$ and $\bar{1}$ are elements of S, and for all elements a, b, and c of S the following properties hold[1]

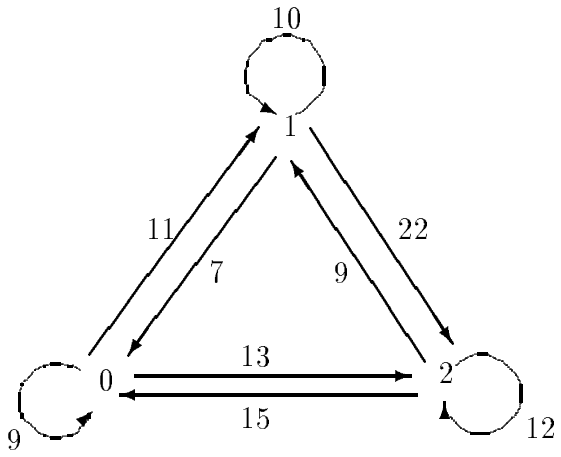
(a)	$a \oplus (b \oplus c) = (a \oplus b) \oplus c$	(associativity of addition)
(b)	$a \oplus b = b \oplus a$	(commutativity of addition)
(c)	$a \oplus \bar{0} = a$	($\bar{0}$ is the neutral element for addition)
(d)	$a \otimes (b \otimes c) = (a \otimes b) \otimes c$	(associativity of multiplication)
(e)	$a \otimes \bar{1} = \bar{1} \otimes a = a$	($\bar{1}$ is the neutral element for multiplication)
(f)	$a \otimes (b \oplus c) = a \otimes b \oplus a \otimes c$ $(b \oplus c) \otimes a = b \otimes a \oplus c \otimes a$	(distributivity of multiplication over addition)

Even though the operations $\oplus$ and $\otimes$ are called *addition* and *multiplication*, they may be quite different from the usual addition and multiplication in applications of the algebraic path problem to a particular graph problem. For example, if we use this algebraic setting to solve the shortest path problem, then $\oplus$ is replaced by **min**, and $\otimes$ is replaced by the usual addition.

2.3 Graph approach

In the *graph approach* to the algebraic path problem, we assume two additional properties of a semiring:

[1] We assume that $\otimes$ has a higher priority than $\oplus$ in the arithmetic expressions, that is $a \otimes b \oplus a \otimes c = (a \otimes b) \oplus (a \otimes c)$.



(a) A weighted graph.

	0	1	2
0	9	11	13
1	7	10	22
2	15	9	12

(b) The corresponding adjacency matrix

Figure 2.1: Weighted graph and its adjacency matrix

for all $a \in S$,

(g) $a \oplus a = a$ (idempotence of addition)

(h) $a \otimes \bar{0} = \bar{0} \otimes a = \bar{0}$ (absorption rule)

We consider a *weighted graph* $G = (V, w)$, where V is a finite set of vertices and $w : V \times V \to S$ is a function that assigns a weight from a semiring S to each (ordered) pair of vertices in the graph. Intuitively, a weighted graph G contains *all* possible edges, including *loops* (that is edges which begin and end in the same vertex), and w is the set of the weights of edges. Figure 2.1a shows an example of a weighted graph. While dealing with "real" graphs, we usually assign non-zero weights to the pairs of vertices connected by an edge, and zero weight to the other pairs.

A *weighted path* from i to j in a weighted graph is an arbitrary sequence of vertices $(i, k_1, k_2, .., k_m, j)$ that begins with i and ends with j. We define the weight of a weighted path as the product of the weights of all edges of the path in order (recall that a generalized multiplication is not necessarily commutative):

$$w(i, k_1, k_2, .., k_m, j) = w(i, k_1) \otimes w(k_1, k_2) \otimes .. \otimes w(k_m, j)$$

Observe that if the weights of non-existent edges of a real graph equal $\bar{0}$, then the weights of all weighted paths containing non-existent edges (that is non-existent paths) also equal $\bar{0}$. The weight of a path of length 0, which begins and ends in the same vertex and does not contain edges, is defined to be equal to $\bar{1}$. (Paths of length 0 should not be confused with loops. A loop also begins and ends in the same vertex, but a loop contains an edge, while the path of length 0 does *not* contain edges.)

> The *algebraic path problem* is the problem to compute the sum of all weighted paths from i to j for all pairs (i, j):
>
> $$d(i,j) = \bigoplus_{\text{all paths}} w(i \to j)$$

We call $d(i,j)$ the *sum-weight function* of the vertices i and j. At this point we encounter the difficulty of computing an infinite sum, because the number of different weighted paths from i to j may be countably infinite[2]. To overcome this difficulty, we assume two more properties of our semiring [Mahr, 1984]:

- (*Infinite distributivity*) If A and B are *countable*, that is either finite or countably infinite, subsets of S and the sums $\bigoplus_{a \in A} a$ and $\bigoplus_{b \in B} b$ are both defined, then the sum $\bigoplus_{a \in A, b \in B} a \otimes b$ is also defined and

  $$\bigoplus_{a \in A, b \in B} a \otimes b = (\bigoplus_{a \in A} a) \otimes (\bigoplus_{b \in B} b)$$

- (*Infinite associativity*) Let A be a countable subset of S, and $\{A_k \mid k \in K\}$ be a disjoint partition of A. If for all A_k, $a_k = \bigoplus_{a \in A_k} a$ is defined, and if $\bigoplus_{k \in K} a_k$ is defined, then $\bigoplus_{a \in A} a$ is also defined, and

  $$\bigoplus_{a \in A} a = \bigoplus_{k \in K} a_k$$

A semiring satisfying these two properties is called *partially complete*. If the sum is defined for every countable subset of S, a semiring is called *complete*.

[2]For the definition of *countably infinite* sets, see [Enderton, 1977]. Informally, the set is countably infinite if all its elements may be enumerated by natural numbers.

These two axioms are sufficient for our purposes. A more rigorous discussion of summability in semirings may be found in [Mahr, 1984].

To describe the algebraic path problem in terms of matrix computations, we introduce an $N \times N$ matrix $W = (w_{ij})$, whose rows and columns are numbered from 0 to $N-1$, such that for all $i, j \in [0..N)$, the element w_{ij} of W contains the weight of the edge (i, j):

$$(\forall i, j \in [0..N))\ w_{ij} = w(i, j)$$

The matrix W is called the *adjacency matrix* of a corresponding weighted graph. An example of an adjacency matrix is shown in Figure 2.1b.

We define addition and multiplication of matrices over a semiring as the usual matrix addition and multiplication: for square matrices $A = (a_{ij})$ and $B = (b_{ij})$ of size $N \times N$, the sum and product of A and B are the matrices of the size $N \times N$ such that

$$A \oplus B = (a_{ij} \oplus b_{ij})$$
$$A \otimes B = (c_{ij}), \quad \text{where } c_{ij} = \bigoplus_{n \in [0..N)} a_{in} \otimes b_{nj}$$

It is straightforward to verify that addition of matrices is commutative and associative, and multiplication is associative and distributes over addition.

Let us now look at a successive powers of a square matrix W:

$$W^2 = W \otimes W = (w_{ij(2)}),$$
$$\text{where } w_{ij(2)} = \bigoplus_{n \in [0..N)} w_{in} \otimes w_{nj}$$
$$W^3 = W \otimes W \otimes W = (w_{ij(3)}),$$
$$\text{where } w_{ij(3)} = \bigoplus_{n_1, n_2 \in [0..N)} w_{in_1} \otimes w_{n_1 n_2} \otimes w_{n_2 j}$$
$$\vdots$$
$$W^k = (w_{ij(k)}),$$
$$\text{where } w_{ij(k)} = \bigoplus_{n_1, .., n_{k-1} \in [0..N)} w_{in_1} \otimes w_{n_1 n_2} \otimes .. \otimes w_{n_{k-1} j}$$

In other words, an entry $w_{ij(k)}$ of W^k contains the sum of the weights of all k-edge paths from i to j. Therefore, the sum-weight function of i and j may be computed as

$$I = \begin{pmatrix} \bar{1} & \bar{0} & \bar{0} & ... & \bar{0} & \bar{0} \\ \bar{0} & \bar{1} & \bar{0} & ... & \bar{0} & \bar{0} \\ \bar{0} & \bar{0} & \bar{1} & ... & \bar{0} & \bar{0} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \bar{0} & \bar{0} & \bar{0} & ... & \bar{1} & \bar{0} \\ \bar{0} & \bar{0} & \bar{0} & ... & \bar{0} & \bar{1} \end{pmatrix}$$

Figure 2.2: The identity matrix

$$d(i,j) = w_{ij} \oplus w_{ij(2)} \oplus w_{ij(3)} \oplus ... \quad \text{where } i \neq j$$
$$d(i,i) = \bar{1} \oplus w_{ii} \oplus w_{ii(2)} \oplus w_{ii(3)} \oplus ...$$

Now if we define $D = (d_{ij})$ as the $N \times N$ matrix whose entries are values of the sum-weight functions for all pairs of vertices i and j, we may express D in terms of W:

$$D = W^* = \bigoplus_{k \geq 0} W^k = I \oplus W \oplus W^2 \oplus W^3 \oplus ... \quad (*)$$

where I is the *identity matrix*, that is all diagonal elements of I equal $\bar{1}$, and all other elements equal $\bar{0}$ (see Figure 2.2). Thus, the algebraic path problem may be stated as the problem to compute the matrix W^* for a given matrix W, where W^* is defined by the above expression. The operation * is called the *closure* of a matrix.

2.4 Matrix approach

Now we consider the approach suggested in [Lehmann, 1977]. Lehmann defined the closure operation on matrices over a closed semiring using induction on the size of a matrix, without relating this operation to the weights of paths in a graph. Lehmann's approach is more general, because it does not require distributivity and associativity over infinite sets, nor idempotence of addition.

2.4.1 Closed semiring

Lehmann defines a new unary operation *, called *closure*, on the elements of a semiring such that

$$(\forall a \in S)\ a^* = \bar{1} \ \oplus\ a \otimes a^* = \bar{1} \ \oplus\ a^* \otimes a \qquad \text{(closure property)}$$

A semiring with a closure operation is called a *closed semiring*.

If a closure operation is defined only on some (not all) elements of S, the semiring is called a *partial closed semiring*. A partial closed semiring may be extended to a closed semiring by adding a new element x and extending the functions $\oplus$, $\otimes$, and * as follows:

for all elements a of S,
(1) $x \ \oplus\ a = a \ \oplus\ x = x$
(2) $x \otimes a = a \otimes x = x$
(3) if a^* is not defined in the initial semiring, then $a^* = x$
(4) $x^* = x$

It is straightforward to verify that the addition and multiplication extended to the new domain $S \cup \{x\}$ still satisfy properties (a)–(f) (see Section 2.2), and the closure operation satisfies the closure property. Moreover, if multiplication is commutative in an initial partial closed semiring, then it is still commutative in the extension of the semiring.

We define the closure of a matrix W of the size $N \times N$ as follows:

- If $N = 1$, then $(a)^* = (a^*)$, that is the closure of a single-element matrix is equal to the matrix containing the closure of this single element.

- If $N > 1$, we divide the matrix W into four submatrices: W_{11} of the size $n \times n$, W_{12} of the size $n \times (N-n)$, W_{21} of the size $(N-n) \times n$, and W_{22} of the size $(N-n) \times (N-n)$, where $0 < n < N$:

$$W = \begin{pmatrix} W_{11} & W_{12} \\ W_{21} & W_{22} \end{pmatrix}$$

Then the closure of W is defined as follows

$$W^* = \begin{pmatrix} W_{11}^* \ \oplus\ W_{11}^* \otimes W_{12} \otimes X^* \otimes W_{21} \otimes W_{11}^* & W_{11}^* \otimes W_{12} \otimes X^* \\ X^* \otimes W_{21} \otimes W_{11}^* & X^* \end{pmatrix},$$

where $X = W_{22} \ \oplus\ W_{21} \otimes W_{11}^* \otimes W_{12}$

The proof that the closure of a matrix is well-defined, that is the result of the computation of W^* described above does not depend on the size of sub-matrices of W, boils down to computing the closure of a matrix with nine sub-matrices in two different ways:

$$\left(\begin{array}{c|cc} W_{11} & W_{12} & W_{13} \\ \hline W_{21} & W_{22} & W_{23} \\ W_{31} & W_{32} & W_{33} \end{array}\right) \quad \text{and} \quad \left(\begin{array}{cc|c} W_{11} & W_{12} & W_{13} \\ W_{21} & W_{22} & W_{23} \\ \hline W_{31} & W_{32} & W_{33} \end{array}\right)$$

and verifying nine identities. The verification is trivial using commutativity and associativity of matrix addition, associativity of matrix multiplication, and distributivity of matrix addition over matrix multiplication.

Lehmann has shown that the closure operation on matrices satisfies the closure property:

$$W^* = I + W \otimes W^* = W^* \otimes W + I$$

where I is the identity matrix (see Figure 2.2).

2.4.2 Simple semiring

A *simple semiring* is a closed semiring with the additional property that

for all $a \in S$, $a \oplus \bar{1} = \bar{1}$

Intuitively, you may view the operation $\oplus$ in a simple semiring as **min**, and $\bar{1}$ as the smallest element of the semiring. However, you should use this intuition carefully, since a semiring may not be totally ordered, that is it may happen that $a \oplus b$ is neither a nor b. (A definition of a partial order and the minimum (infinum) operation in a partially ordered set may be found in [Enderton, 1977].) A good example of $\oplus$ in a simple semiring, which may help you to develop an intuitive idea for this operation, is a greatest common divisor operation, that is S is the set of naturals, and

$a \oplus b =$ the greatest common divisor of a and b

Then divisibility may be viewed as a partial order:

$a \preceq b$ if a divides b

and $\oplus$ as the minimum-operation for this partial order:

$$a \ \oplus \ b = (\ \mathbf{max}\, c : c \preceq a, b : c)$$

that is $a \oplus b$ is the maximal element that is less than a and b w.r.t. the order $\preceq$. It is easy to see that for every natural number a in this semiring, $a \oplus 1 = 1$.

Simple semirings are exactly the *Q-semirings* described in [Yoeli, 1961]. The *regular algebras* described in [Carre, 1971] and [Backhouse and Carre, 1975] are very similar to simple semirings. They do not assume $a \oplus \bar{1} = \bar{1}$, but assume $a \oplus a = a$; their axiomatization makes extensive use of the order ($a \preceq b$ if and only if $a \ \oplus \ b = b$), and this seems to take us away from linear algebra.

It is easy to verify that the following properties hold for all elements of a simple semiring [Yoeli, 1961]:

$$\begin{array}{ll} a \ \oplus \ a = a & \text{(addition is idempotent)} \\ \bar{0} \otimes a = a \otimes \bar{0} = \bar{0} & \text{(absorption rule)} \\ a^* = \bar{1} & \end{array}$$

It might be shown based on these properties that a simple semiring is a special case of the graph approach. Simple semirings have a lot of "good" properties, described in [Conway, 1971]. In particular,

- The definition of the closure of a matrix may be simplified to

$$W^* = \begin{pmatrix} (W_{11} \ \oplus \ W_{12} \otimes W_{22}^* \otimes W_{21})^* & W_{11}^* \otimes W_{12} \otimes X^* \\ X^* \otimes W_{21} \otimes W_{11}^* & X^* \end{pmatrix}$$

- If V is the matrix obtained from W by interchanging rows i and j and columns i and j, then V^* is obtained from W^* by the same interchanges.

Lehmann has proved that the closure of a matrix over a simple semiring may be computed by the formula

$$W^* = \bigoplus_{k \in [0..N)} W^k = I \ \oplus \ W \ \oplus \ W^2 \ \oplus \ W^2 \ \oplus \ .. \ \oplus \ W^{N-1}$$

You may observe that this formula is slightly different from formula $(*)$ in the previous section. Formula $(*)$ is also correct for computing the closure of a matrix over a simple semiring, that is

$$\bigoplus_{k \in [0..N)} W^k = \bigoplus_{k \geq 0} W^k$$

We do not present the formal proof of this equality, but it may be interesting to suggest an intuitive explanation. The operation $\oplus$ in a simple semiring may be viewed as a generalization of the **min** operation, and the problem to find the closure of a given matrix as a generalization of the shortest path problem. Intuitively, all elements of a simple semiring are "positive", and the last equality states that, in order to find the shortest path between two vertices, it is enough to consider paths that contain at most $N-1$ edges. All longer paths have cycles, and since weights of all edges are positive, none of them may be shortest.

2.4.3 Dijkstra semiring

A special case of a simple semiring was described in [Dijkstra, 1959] (of course at that time nobody knew that this was a special case of a simple semiring). A *Dijkstra semiring* is a simple semiring with an additional property that for all elements a and b,

$$a \oplus b = \text{either } a \text{ or } b$$

Intuitively, a Dijkstra semiring may be viewed as a totally ordered semiring, where the order between two elements is defined by the following rule

$$a \preceq b \quad \text{if} \quad a \oplus b = a$$

(It is straightforward to show that the order $\preceq$ in a Dijkstra semiring is indeed a total order.) In the next section we will see that Dijkstra's algorithm for computing the closure W_0^* of a weight-matrix W_0 over a Dijkstra semiring allows us to find quickly a single row of W_0^*, that is the set of sum-weight functions $d(m,i)$ for a *fixed* vertex m and all vertices $i = 0, 1, .., N-1$. This may be used for solving such problems as finding all vertices reachable from a fixed vertex in a directed graph and the single-source shortest path problem.

Table 2.1 presents the models for the algebraic path problem described in this section and compares them in terms of generality.

2.5 Examples of closed semirings

In this section we consider several examples of semirings that show how the algebraic path problem generalizes several important problems on graphs

closed semiring	[Lehmann, 1977]	more general
graph approach	([Rote, 1985])	↑
simple semiring	[Yoeli, 1961]	↓
Dijkstra semiring	[Dijkstra, 1959]	less general

Table 2.1: Models for the algebraic path problem

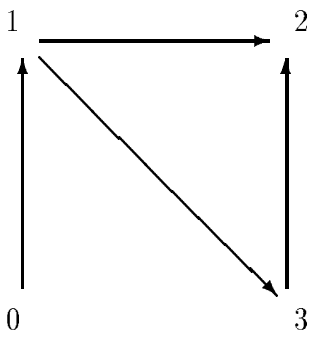
and matrices. A good summary of applications of the algebraic path problem may be found in [Carre, 1979] (Chapters 3 and 4), [Zimmermann, 1981] (Chapter 8), and [Gondran and Minoux, 1984] (Section 3.3).

Before presenting examples, we introduce the notion of a directed graph. A *directed graph* $G = (V, E)$ is a set of *vertices* V, some of which are connected with *directed edges* (see Figure 2.3a). As before, we denote the vertices of the graph by natural numbers $0, 1, .., N-1$. An edge from vertex i to vertex j is denoted by (i, j), and the set of all edges in the graph is denoted by E. Formally, E is an arbitrary subset of $V \times V$. A (real) *path* from i_1 to i_m is a sequence of vertices $(i_1, i_2, .., i_m)$ such that there is an edge from i_1 to i_2, from i_2 to i_3, and so on. Formally,

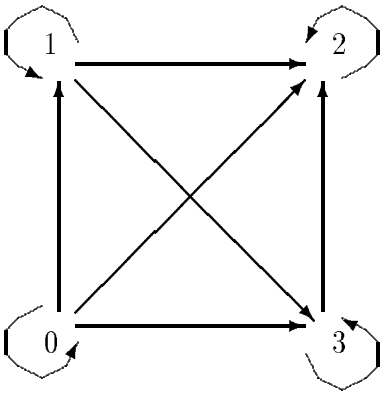
$$(i_1, i_2, .., i_m) \text{ is a (real) path if } (\forall k : 1 < k \leq m : (i_{k-1}, i_k) \in E)$$

For example, edges $(0, 1)$, $(1, 3)$, and $(3, 2)$ in Figure 2.3a form the path $(0, 1, 3, 2)$. A *zero-edge path* (i) is the path that begins and ends in the same vertex i and does not have edges. By convention we assume that every vertex of a directed graph is connected with itself by a zero-edge path. We use the word "real" when we need to emphasize the distinction between a *real* path in a graph and a *weighted* path in the corresponding weighted graph. We denote a path from i to j by $(i \rightarrow j)$.

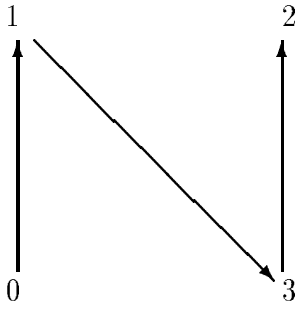
A *cycle* is a path of the form $(i_1, i_2, .., i_1)$, which leads back to its first vertex. For example, a graph in Figure 2.4a contains a cycle $(0, 1, 2, 0)$. A *loop* is a cycle consisting of a single edge. In other words, a loop is an edge of the form (i, i), which begins and ends in the same vertex (see Figure 2.4b). A loop differs from a zero-edge path since a loop contains one edge, while a zero-edge path contains no edges. A graph is called *acyclic* if it does not contain cycles. For example, the graphs in Figures 2.3a and 2.3c are acyclic.



(a) Directed graph G_0



(b) Transitive closure of G_0

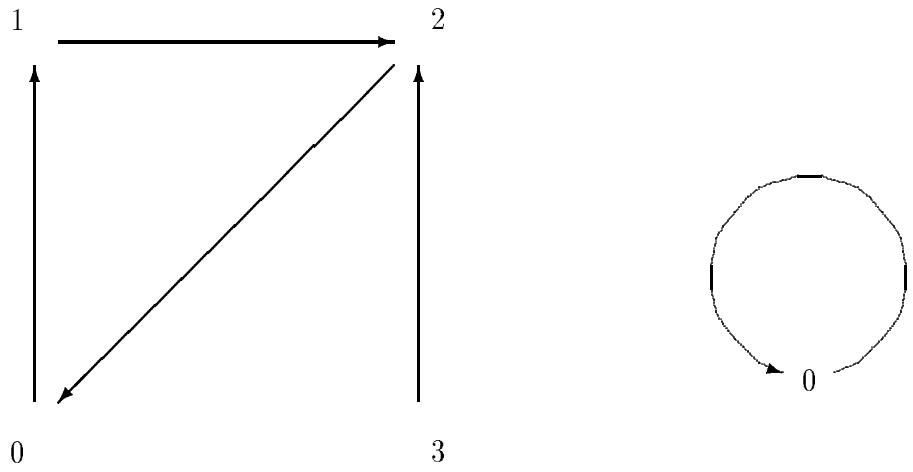


(c) Transitive reduction of G_0

	0	1	2	3
0	0	1	0	0
1	0	0	1	1
2	0	0	0	0
3	0	0	1	0

(d) Adjacency matrix W_0 of G_0

Figure 2.3: Example of an acyclic directed graph



(a) A graph with cycle. (b) A loop.

Figure 2.4: Examples of cycles

On the other hand, the graph in Figure 2.3b is not acyclic, since it contains loops.

2.5.1 Transitive closure

Consider some arbitrary graph $G_0 = (V, E_0)$, and imagine that for every two vertices i and j connected by some path $(i \to j)$, we draw a direct edge (i, j). Then we obtain a graph with the same vertices as the initial graph, and the set of edges E_{cl} such that

$$(i, j) \in E_{\text{cl}} \text{ if and only if there is a path } (i \to j) \text{ in } E_0$$

This new graph is called the *transitive closure* of the initial graph $G_0 = (V, E_0)$. Since every vertex i is connected with itself by a zero-edge path, the transitive closure of a graph contains an edge (i, i) for every vertex i. Figure 2.3b shows an example of the transitive closure.

To show that the transitive closure problem is a special case of the algebraic path problem, we consider a Boolean semiring with two elements, 0 and 1, disjunction for $\oplus$ and conjunction for $\otimes$:

	0	1	2	3
0	1	1	1	1
1	0	1	1	1
2	0	0	1	0
3	0	0	1	1

Figure 2.5: The closure W_0^* of the matrix W_0

$S = \{0, 1\}$
$0 \oplus 0 = 0$ and $0 \oplus 1 = 1 \oplus 0 = 1 \oplus 1 = 1$
$0 \otimes 0 = 0 \otimes 1 = 1 \otimes 0 = 0$ and $1 \otimes 1 = 1$

Observe that 0 is the neutral element for addition, and 1 is the neutral element for multiplication. It is straightforward to check that this structure satisfies the properties of a Dijkstra semiring. We use the graph approach to the algebraic path problem, and build a weighted graph (V, w_0) such that

$$w_0(i,j) = \begin{cases} 1 & \text{if } (i,j) \in E_0 \\ 0 & \text{if } (i,j) \notin E_0 \end{cases}$$

The corresponding matrix W_0 (see Figure 2.3d) is called the *adjacency matrix* of the graph G_0.

We claim that the closure W_0^* of W_0 is the adjacency matrix of the transitive closure G_{cl} of G_0. In other words, for any vertices i and j of G_0, the sum-weight function $d(i,j)$ equals 1 if and only if there is a path from i to j. For example, Figure 2.5 shows the closure of a matrix W_0 from Figure 2.3. One may verify that this is the adjacency matrix of the graph G_{cl} from Figure 2.3. To prove this claim, we consider two cases.

Case 1: there is no real path from i to j.

Then $i \neq j$ and for any weighted path $(i, k_1, .., k_m, j)$, there exists some vertices k_n and k_{n+1} in this sequence such that $(k_n, k_{n+1}) \notin E_0$, and therefore $w(k_n, k_{n+1}) = 0$. Then,

$$\begin{aligned} w_0(i, k_1, .., k_m, j) &= w_0(i, k_1) \otimes .. \otimes w_0(k_n, k_{n+1}) \otimes .. \otimes w_0(k_m, j) \\ &= w_0(i, k_1) \otimes .. \otimes 0 \otimes .. \otimes w_0(k_m, j) \\ &= 0 \end{aligned}$$

and thus the weight of every weighted path from i to j is 0. Since $d(i,j)$ is equal to the conjunction of the weights of all weighted paths from i to j, we conclude that $d(i,j) = 0$.

Case 2: there is a real path from i to j.
If $i = j$ then, by the definition of $d(i,i)$,

$$d(i,i) = 1 \oplus w_{ii} \oplus w_{ii(2)} \oplus ... = 1$$

If $i \neq j$ and $(i, k_1, .., k_m, j)$ is some real path from i to j, then the weights of all edges of the corresponding weighted path equal 1, and therefore the weight of this path also equals 1. Since $\oplus$ is defined as disjunction, this means that the sum of all paths from i to j in our weighted graph equals 1.

2.5.2 Transitive reduction

> To the best of my knowledge, nobody has suggested a semiring for reducing transitive reduction to the algebraic path problem. The semiring described in this section is found by myself.

Now let us take some *acyclic* graph and consider the reverse problem: we wish to remove as many edges as possible, while preserving at least one path between every two edges connected by a path in the initial graph. In other words, we wish to construct a new graph $G_{\text{red}} = (V, E_{\text{red}})$ with the same set of vertices V as the initial graph G_0 and the set of edges E_{red} such that

- E_{red} is a subset of E_0,
- E_{red} has as few edges as possible, and
- for any vertices i and j, if there is a path from i to j in G_0, then there is a path from i to j in G_{red}.

The resulting graph G_{red} is called the *transitive reduction* of G_0. Observe that the transitive closure of G_{red} is the same as the transitive closure of G_0. The following theorem [Aho *et al.*, 1972] guarantees that such G_{red} exists.

Theorem 1 *For any acyclic graph G_0, there exists a unique graph G_{red} such that*

1. Closure(G_{red})=Closure(G_0), *and*

2. *for every graph* $G' = (V, E')$ *such that* E' *is a proper subset of* E_{red}, Closure(E')$\neq$Closure(E_0)

Proof. To construct E_{red}, we consider every edge (i, j) in E_0, and remove it if i and j are connected by a multi-edge path:

$$E_{\text{red}} = \{(i, j) \in E \mid \text{there is no multi-edge path from } i \text{ to } j \text{ in } G_0\}$$

where *multi-edge path* is a path containing two or more edges.

Claim 1: Closure(G_{red})=Closure(G_0)

We need to show that if there is a path from i_1 to i_m in G_0, then there is also a path $(i_1 \to i_m)$ in G_{red}. Consider a path $p_0 = (i_1, .., i_m)$ in G_0. If for some edge (i_{k-1}, i_k) in the path p_0 there is a multi-edge path $(i_{k-1}, j_1, .., j_s, i_k)$ from i_{k-1} to i_k, we replace the edge (i_{k-1}, i_k) in p_0 by this multi-edge path, thus obtaining a new path $p_1 = (i_1, .., i_{k-1}, j_1, .., j_s, i_k, .., i_m)$ (see Figure 2.6). If some edge in p_1 may be replaced by a multi-edge path, we replace it too, and so on. Finally, the replacing process will terminate, because the length of the path grows at each step, and a path in an *acyclic* graph cannot contain more than N vertices. (This is the place where we need the acyclic property of the graph. For a graph with cycles this claim does not hold, and the definition of E_{red} in the beginning of the proof does not make sense.) Since no edge in the resulting path $p_{\text{fin}} = (i_1, .., i_m)$ may be replaced by a longer path, p_{fin} belongs to G_{red}.

Claim 2: If E' is a proper subset of E_{red}, then Closure(G_{red})$\neq$Closure(G_0).

If E' is a proper subset of E_{red}, there exists some $(i, j) \in E_{\text{red}}$ such that $(i, j) \notin E'$. Then, since there is no multi-edge path from i to j in E_{red}, there is no path from i to j in E', and therefore $(i, j) \notin$ Closure(E'), while on the other hand $(i, j) \in$ Closure(E_{red}).

Claim 3: Consider a graph $G'' = (V, E'')$ such that $E'' \subseteq E_0$ and Closure(G'')=Closure(G_0). Then $E_{\text{red}} \subseteq E''$.

Assume there exists an edge $(i, j) \in E_{\text{red}}$ such that $(i, j) \notin E''$. Then $(i, j) \in E_0$, and since Closure(G'')=Closure(G_0), there exists a multi-edge path $(i \to j)$ in E''. Since $E'' \subseteq E_0$, this multi-edge path is also contained in E_0, and therefore, by the definition of E_{red}, $(i, j) \notin E_{\text{red}}$. Contradiction.

It trivially follows from the last claim that the graph G_{red} described in the

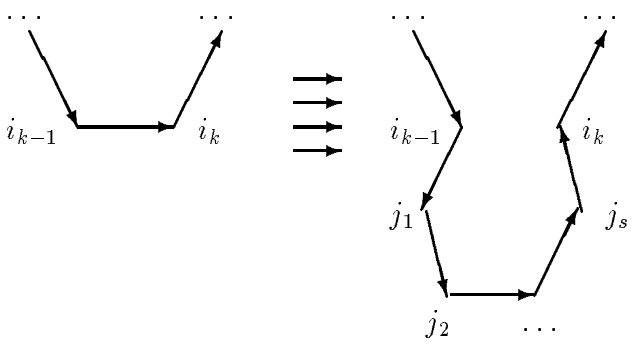


Figure 2.6: Replacement of an edge with a multi-edge path

statement of the theorem is unique. □

Observe that the proof suggests a way to find the transitive reduction of a graph: for every edge (i, j), we need to remove it if and only if there is a multi-edge path from i to j. An example of the transitive reduction of an acyclic graph is shown in Figure 2.3c.

To show that the transitive reduction problem may be viewed as a special case of the algebraic path problem, we consider a semiring with four elements, 0, a, 1, and 2, where 0 is the neutral element for addition and a (*not* 1) is the neutral element for multiplication, and addition and multiplication are defined by the following tables:

$\oplus$	0	a	1	2
0	0	a	1	2
a	a	a	a	a
1	1	a	2	2
2	2	a	2	2

$\otimes$	0	a	1	2
0	0	0	0	0
a	0	a	1	2
1	0	1	2	2
2	0	2	2	2

One may verify that this structure is a simple semiring. We have introduced the neutral element for addition, a, because every semiring must have such an element, but we are not going to use this element to label edges in the graph. Intuitively you may think that $\otimes$ is the usual addition, 2 stands for any natural number larger than 1, and 0 stands for infinity.

For an acyclic graph $G_0 = (V, E_0)$, we build a weighted graph (V, w_0) the same way as in the case of the transitive closure problem:

	0	1	2	3
0	a	1	2	2
1	0	a	2	1
2	0	0	a	0
3	0	0	1	a

	0	1	2	3
0	0	1	0	0
1	0	0	0	1
2	0	0	0	0
3	0	0	1	0

Figure 2.7: Closure of W_0 (left) and the corresponding matrix of the transitive reduction of G_0 (right)

$$w_0(i,j) = \begin{cases} 1 & \text{if } (i,j) \in E_0 \\ 0 & \text{if } (i,j) \notin E_0 \end{cases}$$

We claim that for all i and j, where $i \neq j$, the edge (i,j) belongs to the transitive reduction of G_0 if and only if the sum-weight function $d(i,j)$ equals 1. The edge (i,j) does not belong to the transitive reduction if either $d(i,j) = 0$ or $d(i,j) = 2$. Also, by definition, no loop (i,i) belongs to the transitive reduction. Thus, the adjacency matrix of the transitive reduction of G_0 may be obtained from the closure W_0^* of W_0 by replacing all 2's and all diagonal elements with 0's.

For example, consider the graph in Figure 2.3a. The closure of its adjacency matrix in our semiring and the corresponding adjacency matrix of the transitively reduced graph are shown in Figure 2.7. You may check that this matrix corresponds to the transitive reduction of the graph presented in Figure 2.3c.

To prove our claim for distinct vertices i and j, we divide it into three cases:

1. If there is no path from i to j, then $d(i,j) = 0$.

2. If G_0 contains an edge (i,j) and does not contain a multi-edge path from i to j, then $d(i,j) = 1$.

3. If G_0 contains a multi-edge path from i to j, then $d(i,j) = 2$.

Case 1: there is no path from i to j.
Then we may use the same proof as in the previous section to show that $d(i,j) = 0$.

Case 2: the edge (i,j) is in G_0, and there is no multi-edge path from i to j. Then every weighted multi-edge path from i to j has a zero-weight edge, and therefore the weight of every such path is 0. On the other hand, the weight of the single-edge path from i to j is 1. Thus, $d(i,j)$ is the sum of 1 and several 0's, and therefore $d(i,j) = 1$.

Case 3: there is a multi-edge path from i to j.
Let $(i, k_1, .., k_m, j)$ be a multi-edge path from i to j. Then the weight of the corresponding weighted path is

$$\begin{aligned} w(i,k_1,..,k_m,j) &= d(i,k_1) \otimes d(k_1,k_2) \otimes .. \otimes d(k_m,j) \\ &= \underbrace{1 \otimes 1 \otimes .. \otimes 1}_{\text{2 or more}} \\ &= 2 \end{aligned}$$

and, since the sum-weight function $d(i,j)$ is computed as the maximum of all paths from i to j, we conclude that $d(i,j) = 2$.

2.5.3 All-pairs shortest path

In this problem, we are given a directed graph $G_0 = (V, E_0)$, every edge of which has some length, expressed as a non-negative real number. The length of a path is the sum of the lengths of its edges. The problem is to find the length of the shortest paths between all pairs of vertices.

To solve this problem, we define a semiring, where S is the set of non-negative real numbers together with ∞, $\otimes$ is the usual addition, and $\oplus$ is the minimum-operation. We build a weighted graph (G, w_0), where for all i and j,

$$w_0(i,j) = \begin{cases} \text{the length of } (i,j) & (i,j) \in E_0 \\ \infty & (i,j) \notin E_0 \end{cases}$$

It is straightforward to check that our semiring is indeed a semiring, that is it satisfies all properties of a closed semiring, and the sum-weight $d(i,j)$ equals the length of the shortest path from i to j. Also, one may verify that this semiring is a Dijkstra semiring.

2.5.4 Tunnel problem

This problem is also called the *network capacity problem.* The problem is similar to the all-pairs shortest path problem. We are given a graph with some real-valued cost assigned to every edge. The cost of a path is defined as the minimum of the costs of its edges. Intuitively, we may think that each edge is a road with a tunnel, and the cost of an edge is the height of the tunnel. The cost of a path is the maximal height of a truck that you may drive along this path. The problem is to find the maximal height of a truck that may be driven from i to j, for all pairs of vertices i and j. This time we define $\otimes$ as the minimum-operation, $\oplus$ as the maximum-operation, and weights of edges by

$$w_0(i,j) = \begin{cases} \text{the cost of } (i,j) & (i,j) \in E_0 \\ 0 & (i,j) \notin E_0 \end{cases}$$

Again, one may check that all properties of a Dijkstra semiring hold, and $d(i,j)$ equals the maximal height of a truck that may travel from i to j.

2.5.5 Matrix inversion

This example is intended to show that the algebraic path problem may be used to solve some problems from linear algebra. Given a real-valued non-singular matrix W, we wish to find its *multiplicative inverse*, that is a matrix W^{-1} satisfying

$$W \cdot W^{-1} = W^{-1} \cdot W = I$$

where I denotes the identity matrix.

This time we use Lehmann's matrix approach to the algebraic path problem. Our semiring is the set of real numbers, with the normal addition for $\oplus$ and normal multiplication for $\otimes$. We define a closure of a real number by

$$x^* = \frac{1}{1-x}$$

One may check that the resulting structure satisfies all properties of a partial closed semiring, where 1 is the only element whose closure is not defined. Now

from the equality $W^* = I + W \otimes W^*$ we derive the formula for computing the inverse of the matrix via the closure:

$$\begin{aligned} W^* = I + W \cdot W^* &\equiv W^* - W \cdot W^* = I \\ &\equiv W^* \cdot (I - W) = I \\ &\equiv W^* = (I - W)^{-1} \end{aligned}$$

Now we substitute $(I - W)$ instead of W into the last formula, and receive

$$(I - W)^* = (I - (I - W))^{-1}$$

which implies that the inverse of W may be computed using the formula

$$W^{-1} = (I - W)^*$$

Observe that the semiring for matrix inversion is *not* a simple semiring, and cannot be described by Rote's graph approach, since the addition is not idempotent, and a^* may not equal 1.

Chapter 3

Sequential algorithms

In this section we present two sequential algorithms for solving the algebraic path problem, the Warshall-Floyd algorithm and the Gauss-Jordan algorithm. Warshall's algorithm was initially designed for solving the transitive closure problem, Floyd's algorithm for solving the shortest path problem, and the Gauss-Jordan algorithm for computing the inverse of a real matrix. Lehmann adapted these algorithms for solving the algebraic path problem.

We begin the description of each algorithm by presenting its initial version for solving the corresponding instance of the algebraic path problem, and then show how the algorithm may be modified to solve the general problem. Finally, we present Dijksta's algorithm for solving the single-source algebraic path problem in a Dijkstra semiring.

3.1 Warshall-Floyd algorithm

3.1.1 Warshall's transitive-closure algorithm

Warshall's algorithm for the transitive closure problem [Warshall, 1962] is well-known and often referred to as the first efficient transitive closure algorithm. Warshall's algorithm gives a nice method for computing the transitive closure, which may be used in many other problems, including the general algebraic path problem. Also, Warshall's algorithm is convenient for adapting to parallel computations, in particular to systolic arrays. However, Warshall's algorithm was not the first efficient algorithm for computing the transitive

closure, nor even the second.

The earliest algorithm for the transitive closure problem that I found was presented in [Moore, 1957]. Moore described the *single-source* transitive closure problem, that is the problem to find all vertices reachable from some fixed vertex, in an undirected graph. Moore's article is written in the style of recreational mathematics, without presenting an algorithm in a pseudocode or any specific computer language. However, it is straightforward to use Moore's method for writing a single-source transitive closure algorithm for a directed graph with running time $O(N^2)$. In other words, Moore's method is as efficient as Warshall's algorithm. In 1959 several papers were published with solutions of problems more general than the transitive closure problem, e.g. [Bellman, 1958] described an algorithm for the shortest path problem, and [Kalaba, 1958] described an algorithm for the *stochastic communication network problem*, which is the problem of finding the longest path between two vertices. These algorithms may be easily adapted for solving the transitive closure problem in $O(N^3)$ time.

A year later Roy published a transitive closure algorithm, whose running time was also $O(N^3)$ (Roy, 1959)[1]. In the same year Dijkstra published his article on the algebraic path problem (the term *algebraic path problem* did not yet exist that time), where he presented a generalized algorithm for solving the transitive closure problem and the shortest path problem. Dijkstra's algorithm solves the single-source problem in $O(N^2)$ time.

In 1962 Warshall published his beautiful algorithm in the article *"A theorem on Boolean matrices"* [Warshall, 1962]. (The name of the article shows us that even in those ancient times some people already viewed algorithms as theorems, whose correctness must be proved.) This was only a one-and-a-half-page article, but the algorithm quickly became famous and was used to solve a lot of similar problems. The running time of Warshall's algorithm was also $O(N^3)$.

An increasing interest in the transitive closure and shortest path problems may be observed in the late Sixties. A lot of variations on the transitive closure algorithm were published that time. Examples are [Thorelli, 1966], [Arlazarov *et al.*, 1970], [Purdom, 1970]. Knuth has included a transitive

[1] Roy's article was published in French. Unfortunately, I did not find an English translation of Roy's article, and learned about the running time of Roy's algorithm from references in other papers.

closure algorithm into Volume 1 of his famous book *"The Art of Computer Programming"* [Knuth, 1968]. The worst-case time of all these algorithms is $\Theta(N^3)$.

Furman in 1970 has shown how to reduce the transitive closure problem to the Boolean matrix multiplication problem [Furman, 1970], and Fischer and Meyer in 1971 found the reverse reduction [Fischer and Meyer, 1971]. (The cost of both reductions is $O(N^2)$.) These two results prove that the optimal algorithms for the transitive closure and Boolean matrix multiplication have the same running time[2]. Furman modified Strassen's matrix multiplication algorithm [Strassen, 1969] for computing the transitive closure in $O(N^{\log_2 7})$ time.

Aho, Garey, and Ullman in 1972 have shown how to reduce the transitive-reduction problem to the transitive-closure problem and vice versa in $O(N^2)$ time [Aho *et al.*, 1972]. Thus, these two problems also have the same time complexity.

Bloniarz, Fischer, and Meyer in 1976 suggested some techniques to improve the average-case running time of the transitive closure algorithm [Bloniarz *et al.*, 1976]. The average-case time of the algorithm presented in their paper is $O(N^2 \cdot \log N)$. The work in this direction was continued by Schnorr, who designed an algorithm with average-case running time $O(N+|E|)$, where $|E|$ is the expected number of edges in the transitive closure of a graph [Schnorr, 1978]. Goralcikowa and Koubek in 1979 found a method to compute the transitive closure of an acyclic graph in $O(N \cdot |E_{\rm red}|)$ time, where $|E_{\rm red}|$ is the number of edges in the transitive reduction of the graph [Goralcikowa and Koubek, 1979]. Finally, Simon in 1985 found an algorithm that runs in $O(k \cdot |E_{\rm red}|)$ time, where k is the number of paths that cover all vertices of the graph, and proved that the average-case time of this algorithm is $O(N^2 \cdot \log\log N)$ [Simon, 1985]. For most models of random graphs the average-case time of Simon's algorithm is $O(N^2)$. At present, this is the fastest known sequential algorithm for computing the transitive closure.

The optimal running time for the transitive-closure algorithm is still un-

[2]Strictly saying, this is true only if the optimal running time of both algorithms is $\Omega(N^2)$. Clearly, if we represent an initial graph by the adjacency matrix, then any transitive closure algorithm takes $\Omega(N^2)$ time, because the size of the input is $\Theta(N^2)$. However, the *adjacency list* (see [Cormen *et al.*, 1990], pages 455–456) allows us to represent a graph with only $\Theta(N+|E_0|)$ values, and it may happen that there is a transitive closure algorithm using the adjacency-list representation with $\Theta(N + |E_0|)$ running time.

known. The only thing we know is that the optimal running time for the transitive closure, transitive reduction, and Boolean matrix multiplication are the same, provided that this optimal running time is $\Omega(N^2)$.

Below we describe Warshall's transitive closure algorithm. We represent a graph G_0 by an adjacency matrix $W_0[0..N-1, 0..N-1]$, whose elements are as described in Subsection 2.5.1:

$$W_0[i,j] = \begin{cases} 1 & \text{if there is an edge from } i \text{ to } j \\ 0 & \text{otherwise} \end{cases}$$

We wish to find an adjacency matrix of the transitive closure of the graph G_0, that is the matrix $W_{\rm cl}$ such that

$$W_{\rm cl}[i,j] = \begin{cases} 1 & \text{if there is a path from } i \text{ to } j \text{ in } G_0 \\ 0 & \text{otherwise} \end{cases}$$

We use a predicate $(i \rightarrow^{<n} j)$ to denote the presence of a path from i to j in G_0, all intermediate nodes of which are less than n:

$$(i \rightarrow^{<n} j) \quad \equiv \quad (\exists \text{ path } (i, k_1, k_2, .., k_m, j) \text{ such that } k_1, .., k_m < n)$$

Figure 3.1 shows an example of a path from vertex 6 to 7, all intermediate nodes of which are less than 4. The existence of such a path is denoted by $(6 \rightarrow^{<4} 7)$. A predicate $(i \rightarrow^{0} j)$ means that i and j are connected with a path that does not have intermediate vertices, that is either there is a direct edge from i to j, or $i = j$ and the implied path is the zero-length path connecting a vertex with itself.

The algorithm uses a Boolean matrix $W[0..N-1, 0..N-1]$. Initially, $W = W_0 \vee I$, that is the initial value of W is obtained from W_0 by replacing all diagonal values with 1's. This replacement represents the fact that each vertex is connected with itself by the zero-edge path. After the execution of the algorithm $W = W_{\rm cl}$. We use the invariant

$$P(n) \quad \equiv \quad (\forall i, j : 0 \le i, j < N : W[i,j] = (i \rightarrow^{<n} j))$$

that is after the n-th execution of the main loop, $W[i,j]$ equals 1 if the original graph contains a path from i to j all intermediate nodes of which are less than n. It is easy to verify that

$$P(0) \quad \equiv \quad W = W_0 \vee I, \text{ and}$$
$$P(N) \quad \equiv \quad W = W_{\rm cl}$$

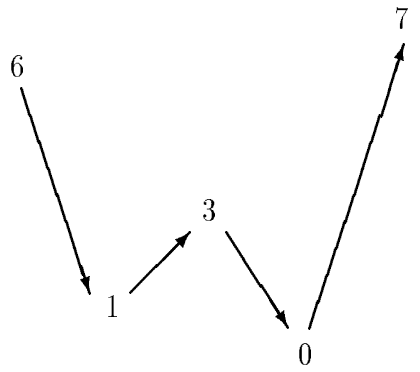


Figure 3.1: Example of a path ($6 \rightarrow^{<4} 7$)

$$\begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \rightarrow \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \rightarrow \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

W_0 W_1 W_2

$$\rightarrow \begin{pmatrix} 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \rightarrow \begin{pmatrix} 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

W_3 $W_4 = W_{\mathrm{cl}}$

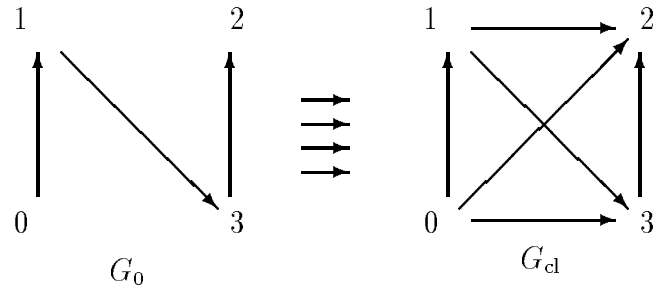


Figure 3.2: Steps is computing the transitive closure

To compute W_{cl}, we need to perform N steps: first we find the matrix W_1 satisfying $P(1)$, then the matrix M_2 satisfying $P(2)$, and so on until we compute W_N (see Figure 3.2). We denote the adjacency matrix satisfying $P(n)$ by W_n. The following derivation shows how to compute W_{n+1} via W_n:

$$\begin{array}{ll} & P(n+1) \\ \equiv & (\forall i,j :: W[i,j] = (i \rightarrow^{<n+1} j)) \\ \equiv & (\forall i,j :: W[i,j] = (i \rightarrow^{<n} j) \vee ((i \rightarrow^{<n} n) \wedge (n \rightarrow^{<n} j))) \\ \equiv & (\forall i,j :: W[i,j] = W_n[i,j] \vee (W_n[i,n] \wedge W_n[n,j])) \end{array}$$

which may be rewritten as

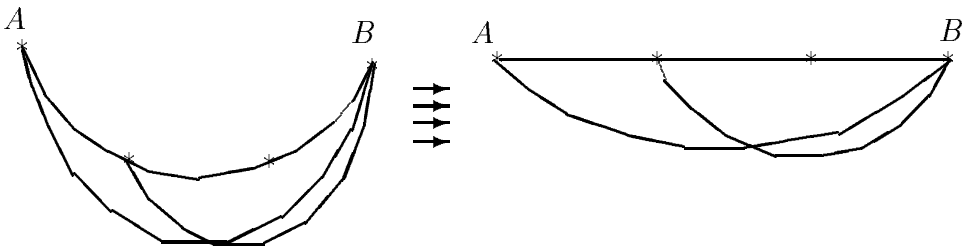
$$(\forall i,j :: W_{n+1}[i,j] = W_n[i,j] \vee (W_n[i,n] \wedge W_n[n,j]))$$

This gives rise to the following algorithm:

$$\begin{array}{l} |[\textbf{con } N\text{: int } \{N > 0\}; \\ \quad W[0..N-1, 0..N-1]\text{: } \textbf{array of } \text{Boolean}; \\ \quad \{P(0) \equiv (W = W_0)\} \\ \qquad n := 0 \\ \qquad ;\textbf{do } n < N \\ \qquad \rightarrow \forall i,j :: W_{n+1}[i,j] := W_n[i,j] \vee (W_n[i,n] \wedge W_n[n,j]) \\ \qquad\quad \{P(n+1)\} \\ \qquad\quad ;n := n+1 \\ \qquad \textbf{od} \\ \quad \{P(N) \equiv (W = W_{\mathrm{cl}})\} \\ ]| \end{array}$$

3.1.2 Floyd's algorithm for all-pairs shortest path

It seems that the shortest path problem is the most popular among graph problems. Research on this problem was started by engineers of the Bell Telephone Company in the early Fifties [Wilkinson, 1956]. However, the articles written by Bell's engineers are hard to understand for computer scientists, because their authors were sloppy in theory and mixed theoretical aspects of the shortest path problem with hardware implementation and financial issues. Several methods were suggested for solving the shortest path problem on analog machines [Rapaport and Abramson, 1959]. The simplest of



Build a string model of a graph, where knots represent vertices and string lengths represent distances. To find the shortest path from A to B, seize knot A in you left hand, and B in your right hand, and pull them apart. The path that stretches tight is the shortest path from A to B. To solve the single-source problem, pick the model by the source vertex and weight all other vertices.

Table 3.1: Minty's analog solution for shortest path

the analog solutions, which works only for undirected graphs, is shown in Table 3.1 [Minty, 1957].

The earliest formal solution of the all-pairs shortest path problem that I was able to find is presented in [Shimbel, 1954]. Shimbel used a matrix approach to this problem. At the same time several researchers tried to solve the shortest path problem with any real (not only non-negative) lengths of edges and found that this problem is equivalent to the travelling salesperson problem [Dantzig *et al.*, 1954], [I. Heller, 1955]. Today we know that this implies NP-completeness.

[Moore, 1957] described informally an algorithm for finding a single-source shortest path problem for an undirected graph with all edges of equal length. The limitations of Moore's algorithm were probably due to the failure to state the more general problem, for it is straightforward to use Moore's method for solving the single-source path problem in a directed graph with arbitrary-length edges in $O(N^2)$ time. Seventeen years later Pape implemented Moore's algorithm and showed experimentally that this implementation runs faster then implementations of most other shortest path algorithms known that time [Pape, 1974].

Dantzig, also in 1957, showed that the single-source shortest path problem may be viewed as a linear programming problem of finding the minimum of a linear function of several variables whose values are restricted by a set of linear equations. In this representation, the problem may be solved by an application of the simplex method [Dantzig, 1957]. Dantzig's method was the first mathematically well-formalized method for solving the single-source shortest path problem.

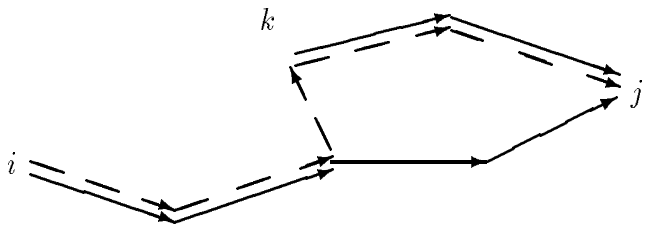
[Bellman, 1958] formulated a computational algorithm for the all-pairs shortest path problem that is essentially equivalent to Shimbel's solution. Bellman's algorithm is based on the Jacobi method for solving systems of linear equations, which uses an iterative technique. However, Bellman failed to prove that the iterations always converge. This fault was corrected by Bentley and Cooke, who proved that Bellman's method converges for any graph with positive-length edges [Bentley and Cooke, 1965].

[Dijkstra, 1959] described a single-source shortest path algorithm with running time $O(N^2)$, and showed the correctness of his algorithm. Dijkstra's algorithm is discussed in the end of this chapter, in Section 3.3.

At approximately the same time several methods were presented for finding the n-th shortest path. The earliest efficient algorithm for this problem was proposed in [Hoffman and Pavley, 1959]. A brief overview of their technique is presented in Table 3.2. A modification of their technique was presented in [Bellman and Kalaba, 1960]. This modification is more efficient than the method of Hoffman and Pavley for finding the n-th paths for all pairs of vertices. The problem was further developed in [Pollack, 1961a], [Pollack, 1961b], and [Clarke *et al.*, 1963].

Pollack and Weibenson presented the first review of different shortest path algorithms in [Pollack and Wiebenson, 1960]. This article describes a simple efficient method, designed by Minty, that is still used for computing the single-source shortest path [Cormen *et al.*, 1990]. (Minty himself did not publish this method). The method is presented in Table 3.3. An algorithm based on this method is described in [Whiting and Hiller, 1960]. A formal treatment of an algorithm based on Minty's method is given in [Ford Fulkerson, 1962] (pages 130–134), together with a proof that the assumption about non-negative lengths of edges may be replaced by the less restrictive assumption that the sum of lengths around any directed cycle is non-negative.

Finally, in 1962 Floyd published his well-known shortest-path algorithm [Floyd, 1962], based on Warshall's shortest path algorithm. It is hard to say



A *deviation* from the shortest path is a path that coincides with the shortest path from its origin to some node j, then deviates directly to some node k and finally proceeds from k to the terminal node via the shortest path from k. It is shown that the second shortest path is always a deviation from the shortest path.

To find the second shortest path from i to j, we first find the shortest paths from all nodes to j. Then we determine all deviations from the shortest path between i and j and choose the shortest of the deviations. A generalization of this technique allows us to find the n-th shortest path from i to j.

Table 3.2: Hoffman and Pavley's method for finding the second shortest path

1. Label the source vertex with the distance 0.
2. Look at all edges with their "tails" labelled and their "heads" unlabelled. For each such edge, form the sum of the label and the length. Select an edge making this sum minimal, and label the "head" vertex of this edge with the sum.
3. If some vertices have yet to be labelled, return to the beginning of 2.

Table 3.3: Minty's method for the single-source shortest path problem

why Floyd's algorithm is known better than any algorithm published before. The algorithm was not more efficient than the previous algorithms, nor better formalized, nor did it contain any new ideas. Probably the reason is that it was the first shortest path algorithm published in *Communications of the ACM*.

In almost all early publications on the shortest path problem, authors did not present an evaluation of the time complexity of their algorithms. However, most algorithms are *"efficient"* or can be made efficient by simple modifications, where *efficient* means that they solve the all-pairs problem in $O(N^3)$ time, and the single-source problem in $O(N^2)$ time. The most efficient is Minty's method (Table 3.3), which may be implemented as an algorithm on a graph represented by an adjacency list with $O(N + |E_0|)$ running time, where $|E_0|$ is the number of edges.

In the early Sixties, the shortest path problem became very popular. A huge number of different shortest path algorithms were published between the early Sixties and the mid Seventies, and we will not try to survey all these algorithms in our report. Different variations of the problem were considered, such as finding a shortest path in a *sparse graph* (that is a graph with a small number of edges) [Wagner, 1976], [D. Johnson, 1977], determining the fastest path through a network with travel times depending on the departure time [Cooke and Halsey, 1966], finding the shortest path through specified intermediate nodes [Saksena and Kumar, 1968][3], [Dreyfus, 1969], and so on. An overview of some algorithms may be found in [Dreyfus, 1969]. A bibliography of most shortest path algorithms published between 1956 and 1974 is presented in [Pierce, 1975].

[Spira, 1973] presented a modification of Dijkstra's algorithm for the all-pairs problem with average-case running time $O(N^2 \cdot (\log N)^2)$. The worst-case running time of Spira's algorithm was still $O(N^3)$. The first all-pairs algorithm with running time $o(N^3)$ was suggested in [Fredman, 1976]. This algorithm runs in $O(\frac{N^3 \cdot (\log\log N)^{1/3}}{(\log N)^{1/3}})$ time and performs $O(N^{5/2})$ comparisons and additions. At present this is the most efficient algorithm for solving the all-pairs shortest path problem. The lower bound for the all-pairs problem is still unknown.

The lower bound for the single-source problem was found by Spira and Pan, who proved that this problem requires $\Omega(N^2)$ comparisons in the worst

[3]The solution of Saksena and Kumar is incorrect.

case [Spira and Pan, 1973]. Spira and Pan did not consider the lower bound as a function of both the number of vertices N and the number of edges $|E_0|$, but it is straightforward to show that this lower bound is $\Omega(N + |E_0|)$ (e.g. see [Cormen *et al.*, 1990]).

To solve the shortest path problem, we represent the weights of edges in the graph by a matrix $W_0[0..N-1, 0..N-1]$, as described in Subsection 2.5.3. We wish to compute the matrix $D[0..N-1, 0..N-1]$, where for all vertices i and j, $D[i,j]$ equals the length of the shortest path from i to j if there is a path from i to j, and ∞ otherwise.

Below we present Floyd's shortest path algorithm, which is similar to Warshall's transitive-closure algorithm. It uses a real-valued matrix $W[0..N-1, 0..N-1]$. The initial value W_0 of the matrix W is obtained from the adjacency matrix of a given graph by replacing all diagonal elements with zeros. This replacement represents the fact that each vertex is connected with itself by the zero-edge path, the length of which equals zero. After the execution the matrix W is equal to D. Instead of the literal $(i \rightarrow^{<n} j)$, the invariant of Floyd's algorithm uses a real-valued function $w(i \rightarrow^{<n} j)$, which is equal to the length of the shortest path from i to j, all intermediate vertices of which are less than n:

$$
\begin{aligned}
& w(i \rightarrow^{<n} j) \\
= \; & (\,\mathbf{min}\, m, k_1, .., k_m : 0 \le k_1, .., k_m < n : w(i, k_1, .., k_m, j)) \\
= \; & (\,\mathbf{min}\, m, k_1, .., k_m : 0 \le k_1, .., k_m < n : w(i, k_1) + .. + w(k_m, j))
\end{aligned}
$$

We use the following invariant (notice its similarity to Warshall's invariant):

$$Q(n) \quad \equiv \quad (\forall i, j : 0 \le i, j < N : W[i,j] = w(i \rightarrow^{<n} j))$$

Again, one may verify that

$$
\begin{aligned}
Q(0) \quad & \equiv \quad W = W_0, \text{ and} \\
Q(N) \quad & \equiv \quad W = D
\end{aligned}
$$

To derive the formula for computing W_{n+1} via W_n (where W_n is the matrix satisfying $Q(n)$), we proceed as follows

$$
\begin{aligned}
& Q(n+1) \\
\equiv \; & (\forall i, j :: W[i,j] = w(i \rightarrow^{<n+1} j)) \\
\equiv \; & (\forall i, j :: W[i,j] = w(i \rightarrow^{<n} j)\, \mathbf{min}\, (w(i \rightarrow^{<n} n) + w(n \rightarrow^{<n} j))) \\
\equiv \; & (\forall i, j :: W[i,j] = W_n[i,j]\, \mathbf{min}\, (W_n[i,n] + W_n[n,j]))
\end{aligned}
$$

which may be rewritten as

$$(\forall i, j :: W_{n+1}[i,j] = W_n[i,j] \,\mathbf{min}\, (W_n[i,n] + W_n[n,j]))$$

This invariant gives rise to an algorithm similar to the Warshall's algorithm.

3.1.3 Warshall-Floyd algorithm for algebraic path

You have probably noticed the similarity of Warshall's and Floyd's invariants. The only difference between them is that $\vee$ in Warshall's invariant is replaced by **min**, and $\wedge$ is replaced by $+$. Moreover, as one may verify, the substitution of **max** instead of $\vee$ and **min** instead of $\wedge$ into Warshall's algorithm will produce the algorithm for solving the tunnel problem, and the same substitution in the correctness proof of Warshall's algorithm will produce the correctness proof for the tunnel-problem algorithm.

Perhaps we may use the same method to solve the algebraic path problem and obtain a general algorithm by replacing $\vee$ with the generalized multiplication $\otimes$, and $\wedge$ with the generalized addition $\oplus$? For the graph approach to the algebraic path problem, suggested by Rote, this is indeed the case. So the algorithms described above may be used to compute the closure of a matrix in Rote's model of the algebraic path problem, in particular in a simple semiring. (Recall that a simple semiring is a special case of Rote's graph approach.) However, in the case of Lehmann's matrix approach, an algebraic path algorithm is a little bit more complex, since we need to take into account the closure operation *.

The algorithm discussed below was designed by Lehmann based on Warshall's transitive closure algorithm [Warshall, 1962] and Floyd's all-pairs shortest path algorithm [Floyd, 1962], and using the ideas presented in Kleene's proof that every regular language can be represented by a regular expression [Kleene, 1956].

Given a matrix $W_0[0..N-1, 0..N-1]$ over an arbitrary closed semiring, we wish to compute its closure W_0^*. Again, the computation is performed in N steps using a matrix $W[0..N-1, 0..N-1]$ such that initially $W = W_0$ and after the execution of the algorithm $W = W_0^*$. To develop an invariant for the Warshall-Floyd algorithm using Lehmann's approach, we introduce the notation $W[i..j, k..m]$, which denotes the submatrix of W consisting of rows i to j and columns k to m (see Figure 3.3a). The interval $i..i$ is abbre-

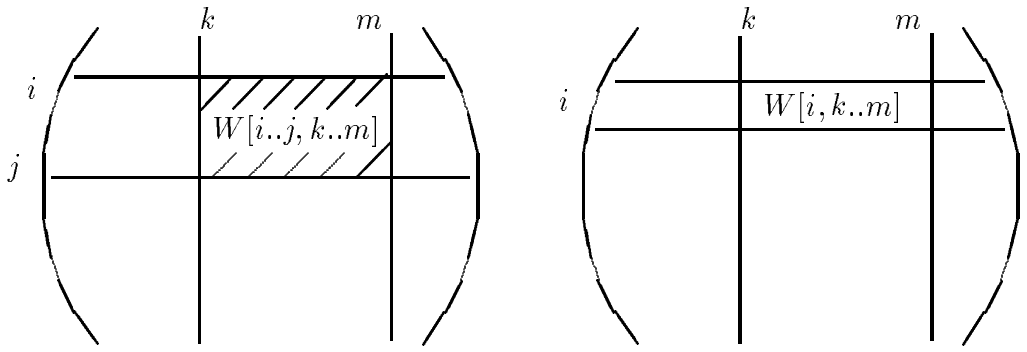


Figure 3.3: Submatrices of the matrix W

viated to i, that is $W[i, k..m]$ denotes the one-column matrix $W[i..i, k..m]$ (see Figure 3.3b).

The invariant suggested by Lehmann is as follows:

$$\begin{array}{lcl} P(n) & \equiv & W = W_0 \oplus W_0[0..N-1, 0..n-1] \\ & & \otimes (W_0[0..n-1, 0..n-1])^* \\ & & \otimes W_0[0..n-1, 0..N-1] \end{array}$$

As usual, we denote the matrix satisfying $P(n)$ by W_n. By convention we assume that the product

$$W[0..N-1, 0..-1] \otimes (W_0[0..-1, 0..-1])^* \otimes W_0[0..-1, 0..N-1]$$

should just be ignored, and receive the initial invariant

$$\begin{array}{lcl} P(0) & \equiv & W = W_0 \oplus W_0[0..N-1, 0..-1] \\ & & \otimes (W_0[0..-1, 0..-1])^* \\ & & \otimes W_0[0..-1, 0..N-1] \\ & \equiv & W = W_0 \end{array}$$

The invariant after the execution of all N steps gives

$$\begin{array}{lcl} P(N) & \equiv & W = W_0 \oplus W_0[0..N-1,0..N-1] \\ & & \quad \otimes (W_0[0..N-1,0..N-1])^* \\ & & \quad \otimes W_0[0..N-1,0..N-1] \\ & \equiv & W = W_0 \oplus W_0 \otimes W_0^* \otimes W_0 \end{array}$$

Recall that for any matrix W_0, the following equality holds in a closed semiring

$$W_0^* = I \oplus W_0 \otimes W_0^* = I \oplus W_0^* \otimes W_0$$

Therefore, if we compute $W_{N+1} = I \oplus W_N$, then

$$\begin{array}{rcl} W_{N+1} & = & I \oplus W_N \\ & = & I \oplus W_0 \oplus W_0 \otimes W_0^* \otimes W_0 \\ & = & I \oplus W_0 \otimes (I \oplus W_0^* \otimes W_0) \\ & = & I \oplus W_0 \otimes W_0^* \\ & = & W_0^* \end{array}$$

It remains to design an algorithm that leads from $P(n)$ to $P(n+1)$, that is to find a formula for computing W_{n+1} via W_n. The formula found by Lehmann is

$$W_{n+1} = W_n \oplus W_n[0..N-1,0..n] \otimes (W_n[n,n])^* \otimes W_n[0..n,0..N-1]$$

To prove that the computation using this formula preserves the invariant, we use induction. (This proof may be skipped at first reading.) For brevity, we introduce the following notation (see Figure 3.4):

$$\begin{array}{l} A = W_0[0..n-1,0..n-1] \\ P = W_0[n,0..n-1] \\ Q = W_0[0..n-1,n] \\ B_n = W_n[n,n] \end{array}$$

Assume that the invariant $P(n)$ holds for W_n, that is

$$\begin{array}{rcl} W & = & W_0 \oplus W_0[0..N-1,0..n-1] \\ & & \quad \otimes (W_0[0..n-1,0..n-1])^* \\ & & \quad \otimes W_0[0..n-1,0..N-1] \end{array}$$

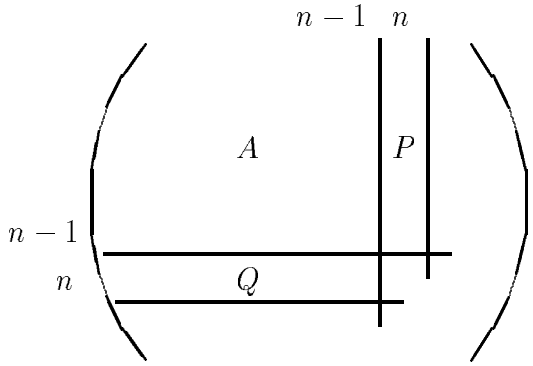


Figure 3.4: Submatrices of the matrix W_0

from which, by the definition of the matrix multiplication, we derive

$$\begin{aligned}
W_n[0..N-1, n-1] &= W_0[0..N-1, n-1] \\
&\quad \oplus W_0[0..N-1, 0..n-1] \otimes A^* \otimes Q \\
W_n[n-1, 0..N-1] &= W_0[n-1, 0..N-1] \\
&\quad \oplus P \otimes A^* \otimes W_0[0..n-1, 0..N-1] \\
B_n \;=\; W_n[n,n] &= W_0[n,n] \;\oplus\; P \otimes A^* \otimes Q
\end{aligned}$$

Substituting these equalities into the formula for computing W_{n+1}, we obtain

$$\begin{aligned}
W_{n+1} \;=\;& W_n \\
\oplus\;& W_n[0..N-1, 0..n] \\
\otimes\;& (W_n[n,n])^* \\
\otimes\;& W_n[0..n, 0..N-1] \\
=\;& W_0 \;\oplus\; W_0[0..N-1, 0..n-1] \otimes A^* \otimes W_0[0..n-1, 0..N-1] \\
\oplus\;& (W_0[0..N-1, n] \;\oplus\; W_0[0..N-1, 0..n-1] \otimes A^* \otimes Q) \\
\otimes\;& B_n^* \\
\otimes\;& (W[n, 0..N-1] + P \otimes A^* \otimes W_0[0..n-1, 0..N-1]) \qquad (*)
\end{aligned}$$

On the other hand, by the definition of the closure operation

$$W_0[0..n, 0..n] \;=\; \begin{pmatrix} A & Q \\ P & W_0[n,n] \end{pmatrix}^*$$

$$= \begin{pmatrix} A^* \oplus A^* \otimes Q \otimes B^* \otimes P \otimes A^* & A^* \otimes Q \otimes B^* \\ B^* \otimes P \otimes A^* & B^* \end{pmatrix}$$

and

$$\begin{aligned}
&W_0[0..N-1, 0..n] \otimes (W_0[0..n, 0..n])^* \otimes W_0[0..n, 0..N-1] \\
&= W_0[0..N-1, 0..n-1] \\
&\qquad \otimes (A^* \oplus A^* \otimes Q \otimes B^* \otimes P \otimes A^*) \\
&\qquad \otimes W_0[0..n-1, 0..N-1] \\
&\quad \oplus W_0[0..N-1, n] \otimes B^* \otimes P \otimes A^* \otimes W_0[0..n-1, 0..N-1] \\
&\quad \oplus W_0[0..N-1, 0..n-1] \otimes Q \otimes B^* \otimes P \otimes W_0[n, 0..N-1] \\
&\quad \oplus W_0[0..N-1, n] \otimes B^* \otimes W_0[n, 0..N-1]
\end{aligned}$$

Comparing with $(*)$ gives us

$$W_{n+1} = W_0 \oplus W_0[0..N-1, 0..n] \otimes (W_0[0..n, 0..n])^* \otimes W_0[0..n, 0..N-1]$$

which means that the invariant $P(n+1)$ holds for W_{n+1}.

Thus, the following algorithm computes the closure of a given matrix W_0.

$$\begin{aligned}
&|[\textbf{con } N\text{: int } \{N > 0\}; \\
&\quad W[0..N-1, 0..N-1]\text{: } \textbf{array of } \text{Semiring-Values}; \\
&\quad \{P(0) \equiv (W = W_0)\} \\
&\qquad n := 0 \\
&\qquad ;\textbf{do } n < N \\
&\qquad \rightarrow \{P(n)\} \\
&\qquad\quad W := W \oplus W[0..N-1, n] \otimes (W[n, n])^* \otimes W[n, 0..N-1] \\
&\qquad\quad \{P(n+1)\} \\
&\qquad\quad ;n := n+1 \\
&\qquad \textbf{od} \\
&\qquad \{P(N) \equiv W = W_0 \oplus W_0 \otimes W_0^* \otimes W_0)\} \\
&\qquad ;W := I \oplus W \\
&\quad \{W = W_0^*\} \\
&]|
\end{aligned}$$

We obtain the final algorithm by replacing matrix addition and multiplication by elementwise operations. The running time of the algorithm is $O(N^3 \cdot (T_\oplus + T_\otimes + T_*))$, where $T_\oplus$, $T_\otimes$, and T_* are running times required to compute respectively the sum, product, and closure of elements of a semiring. These times are *not* necessarily constant. For example, the closure of an element is often computed as a convergent series [Cormen *et al.*, 1990]:

$$a^* = \bar{1} \oplus a \oplus (a \otimes a) \oplus (a \otimes a \otimes a) \oplus \ldots$$

3.2 Gauss-Jordan algorithm

3.2.1 Matrix inversion

In this subsection we present the Gauss-Jordan method for finding the inverse W_0^{-1} of a non-singular matrix W_0 over real numbers. (A matrix is called *non-singular* if it does have an inverse.) The formal proof of the generalized Gauss-Jordan method is presented in the next subsection, and for now we give only an informal description of the algorithm. (For the formal description of the Gauss-Jordan algorithm see, for example, [Forsythe and Moller, 1967], [Bulirsch and Stoer, 1980], or [Press *et al.*, 1989].)

To find the inverse of a non-singular matrix $W_0[0..N-1, 0..N-1]$, we use the matrix C_0 of size $N \times (2 \cdot N)$ composed of the matrix W_0 and the identity matrix,

$$C_0 = (W_0, I)$$

which pictorially looks as follows:

$$C_0 = \begin{pmatrix} W_0[0,0] & W_0[0,1] & \ldots & W_0[0,N-1] & 1 & 0 & \ldots & 0 \\ W_0[1,0] & W_0[1,1] & \ldots & W_0[1,N-1] & 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ W_0[N-1,0] & W_0[N-1,1] & \ldots & W_0[N-1,N-1] & 0 & 0 & \ldots & 1 \end{pmatrix}$$

The algorithm operates with $N \times (2 \cdot N)$ matrix C, which initially equals C_0. The algorithm aims to reduce C to the matrix (I, W_0^{-1}) by multiplying it by the N elementary Jordan matrices $J_0, J_1, .., J_{N-1}$ so as to obtain

$$J_{N-1} \cdot \ldots \cdot J_1 \cdot J_0 \cdot C_0 = (I, W_0^{-1})$$

Let us denote the matrix obtained after n multiplications by C_n,

$$C_n = J_{n-1} \cdot C_{n-1}$$

We choose J_{n-1} such that the first n columns of C_n are the same as the first n columns of the identity matrix, that is

$$C_n = (I[0..N-1, 0..n-1], C_n[0..N-1, n..2 \cdot N-1])$$

which pictorially may be shown as follows

$$\begin{pmatrix}
1 & 0 & ... & 0 & C_n[0,n] & C_n[0,n+1] & ... & C_n[0,2\cdot N-1] \\
0 & 1 & ... & 0 & C_n[1,n] & C_n[1,n+1] & ... & C_n[1,2\cdot N-1] \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
0 & 0 & ... & 1 & C_n[n-1,n] & C_n[n-1,n+1] & ... & C_n[n-1,2\cdot N-1] \\
0 & 0 & ... & 0 & C_n[n,n] & C_n[n,n+1] & ... & C_n[n,2\cdot N-1] \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
0 & 0 & ... & 0 & C_n[N-2,n] & C_n[N-2,n+1] & ... & C_n[N-2,2\cdot N-1] \\
0 & 0 & ... & 0 & C_n[N-1,n] & C_n[N-1,n+1] & ... & C_n[N-1,2\cdot N-1]
\end{pmatrix}$$

The matrix J_n only differs from the identity matrix I in its n-th column. If $C_n[n,n] = 0$, then the initial matrix W_0 does not have the inverse. If $C_n[n,n]$ is not equal to 0, then the n-th column of J_n may be computed by the following algorithm:

$J_n[n,n] := 1/C_n[n,n]$
;$i := 0$
;**do** $i < N$
$\rightarrow$ **if** $i \neq n \rightarrow J_n[i,n] := -C_n[i,n]/C_n[n,n]$
[] $i = n \rightarrow$ skip
fi
;$i := i + 1$
od

It may be shown that if we use this algorithm to compute J_n, then for all $n \in [0..N-1]$, C_n has the form indicated above, that is

$$C_n = (I[0..N-1, 0..n-1], C_n[0..N-1, n..2 \cdot N-1])$$

and

$$C_N = (I, W_0^{-1})$$

Now let us consider the method for performing all computations within the array C. Since we know in advance the values of the first n columns of the array C_n, we do not store these values. Instead, we use column 0 of C_n to

store column 0 of J_0, column 1 of C_n to store column 1 of J_1, and so on till column $(n-1)$ of C_n, which is used to store column $(n-1)$ of J_{n-1}, that is

$$C_n = (J_0[0..N-1,0], J_1[0..N-1,1], .., J_{n-1}[0..N-1,n-1], C_n[0..N-1,n..2\cdot N-1])$$

which pictorially looks as follows

$$\begin{pmatrix} J_0[0,0] & \ldots & J_n[0,n-1] & C_n[0,n] & \ldots & C_n[0,2\cdot N-1] \\ J_0[1,0] & \ldots & J_n[1,n-1] & C_n[1,n] & \ldots & C_n[1,2\cdot N-1] \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ J_0[n-1,0] & \ldots & J_n[n-1,n-1] & C_n[n-1,n] & \ldots & C_n[n-1,2\cdot N-1] \\ J_0[n,0] & \ldots & J_n[n,n-1] & C_n[n,n] & \ldots & C_n[n,2\cdot N-1] \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ J_0[N-2,0] & \ldots & J_n[N-2,n-1] & C_n[N-2,n] & \ldots & C_n[N-2,2\cdot N-1] \\ J_0[N-1,0] & \ldots & J_n[N-1,n-1] & C_n[N-1,n] & \ldots & C_n[N-1,2\cdot N-1] \end{pmatrix}$$

Since we do not need the first n columns of C_n, we compute only the last $(2\cdot N-n)$ columns of the product $J_{n-1}\cdot C_{n-1}$. Below we present the algorithm that computes C_N (that is the last N columns of C_N) in N steps, using the described method. For the sake of clarity the algorithm uses $(N+1)$ different arrays, $C_0, C_1, .., C_N$. However, it is easy to modify the algorithm so that it performs the same computations using one array C.

```
for n := 0 to N − 1 do
begin
  {Compute J_n, store J_n[0..N − 1, n] in C_{n+1}[0..N − 1, n]}
   C_{n+1}[n, n] := 1/C_n[n, n];
   for i := 0 to N − 1 do
      if i ≠ n
         then C_{n+1}[i, n] := −C_n[i, n]/C_n[n, n];
  {Compute C_{n+1}[0..N − 1, n + 1..2 · N − 1]}
   for j := n + 1 to 2 · N − 1
   begin
      C_{n+1}[n, j] := C_{n+1}[n, n] · C_n[n, j];
      for i := 0 to N − 1 do
         if i ≠ n
            C_{n+1}[i, n] := C_n[i, j] + C_{n+1}[i, n] · C_n[n, j]
   end
end
```

One may verify that the running time of the algorithm is $O(N^3)$, if real addition and multiplication are performed in constant time.

3.2.2 Gauss-Jordan algorithm for algebraic path

The Gaussian elimination method was used for solving the algebraic path problem in [Carre, 1971] and [Tarjan, 1976], and then modified by Lehmann for his most general setting for this problem. In this subsection we present Lehmann's generalization of the Gauss-Jordan method for computing the closure of a matrix over an arbitrary closed semiring. This algorithm is a straightforward translation of the Gauss-Jordan method described above.

In the following algorithm, $W_0[0..N-1, 0..N-1]$ is the initial matrix, and after the execution of the algorithm, $G = W_0^*$. For the sake of simplicity, we keep in memory N matrices of the size $N \times N$, and store the result of each step of the computation as a separate matrix. It is not difficult to write a program that performs the same computations using only two $N \times N$ matrices.

1. **for** $n := 0$ **to** $N-1$ **do**
2. **for** $i := n$ **to** $N-1$ **do**
3. **for** $j := 1$ **to** $N-1$ **do**
4. $W_{n+1}[i,j] := W_n[i,j] \oplus W_n[i,n] \otimes (W_n[n,n])^* \otimes W_n[n,j]$
5. **for** $i := 0$ **to** $N-1$ **do**
6. **for** $j := 0$ **to** $N-1$ **do**
7. $G[i,j] := W_i[i,j]$;

$\{G = G'\}$

8. **for** $i := N-2$ **downto** 0 **do**
9. **for** $j := 0$ **to** $N-1$ **do**
10. **for** $n := i+1$ **to** $N-1$ **do**
11. $G[i,j] := G[i,j] \oplus W_{i+1}[i,n] \otimes G[n,j]$;
12. **for** $i := 0$ **to** $N-1$ **do**
13. $G[i,i] := G[i,i] \oplus \bar{1}$

It is straightforward to verify that the running time of the algorithm is $O(N^3 \cdot (T_\oplus + T_\otimes + T_*))$, where $T_\oplus$, $T_\otimes$, and T_* are the running times required to compute respectively the sum, product, and closure of elements of a semiring. Thus the algorithm has the same running time as the Warshall-Floyd algorithm. The advantage of the Gauss-Jordan method is apparent when for every element a of a semiring, $\bar{0} \otimes a = a \otimes \bar{0} = \bar{0}$, and the input matrix W_0 is *sparse*, that is W_0 contains a large number of zeros. In this case zeros stay longer in the Gauss-Jordan algorithm than in the Warshall-Floyd algorithm, and this reduces the number of multiplication operations (recall that

the multiplication in a semiring may take more than a constant time). It is shown in [Tarjan, 1975] under assumptions close to Lehmann's assumptions, but seemingly incompatible with them[4], that with a suitable data representation the Gauss-Jordan method may be implemented in a number of basic steps that is almost linear in the number of non-zero entries in W_0 for a large class of matrices with restricted zero-nonzero structure.

We shall now proceed with showing that the above algorithm computes the closure of the input matrix W_0. We are going to use the notation introduced in the previous section. (If you are not interested in the formal proof of correctness of the Gauss-Jordan algorithm, you may skip the proof and go directly to the next section.)

By comparison of the first pass of the algorithm (lines 1–4) with the Warshall-Floyd algorithm, one may verify that lines 1–4 compute the sequence of N matrices $W_1, W_2, .., W_N$ such that

$$\text{for all } n \in [1..N],\ \ W_n[n-1..N-1, 0..N-1] = W'_n[n-1..N-1, 0..N-1])$$

where W'_n is defined as the matrix computed at the n-th step of the Warshall-Floyd algorithm, that is

$$\begin{array}{rcll} W'_n & = & W_0 & \oplus\ W_0[0..N-1, 0..n-1] \\ & & & \otimes\ (W_0[0..n-1, 0..n-1])^* \\ & & & \otimes\ W_0[0..n-1, 0..N-1] \end{array}$$

Then lines 5–7 compute the matrix G' such that

$$\text{for all } n \in [0..N-1],\ \ G'[n, 0..N-1] = W_{n+1}[n, 0..N-1]$$

which pictorially may be shown as

$$G' = \begin{pmatrix} W_1[0,0] & W_1[0,1] & \dots & W_1[0,N-1] \\ W_2[1,0] & W_2[1,1] & \dots & W_2[1,N-1] \\ \dots & \dots & \dots & \dots \\ W_N[N-1,0] & W_N[N-1,1] & \dots & W_N[N-1,N-1] \end{pmatrix}$$

Then lines 8–11 compute the sequence of row vectors $g_{N-1}, .., g_0$ such that

[4]The task to compare Tarjan's and Lehmann's models for the algebraic path problem in terms of generality is probably not hard, but involves a lot of tedious technical work. I did not find such a comparison in the literature.

$$g_{N-1} = G'[N-1, 0..N-1]$$

and for all $n \in [0..N-2]$,

$$g_n = G'[n, 0..N-1] \ \oplus \ G'[n, n+1..N-1] \otimes \begin{pmatrix} g_{n+1} \\ g_{n+2} \\ \vdots \\ g_{N-1} \end{pmatrix}$$

We claim that

$$\text{for all } n \in [0..N-1], \ \ g_n = W'_N[n, 0..N-1]$$

We prove the claim by backward induction on n. For $n = N-1$,

$$g_{N-1} = W'_N[N-1, 0..N-1]$$

Now assume that the claim holds for $n+1$. Then, by induction hypothesis,

$$\begin{aligned} g_n &= G'[n, 0..N-1] \ \oplus \ G_N[n, n+1..N-1] \otimes \begin{pmatrix} g_{n+1} \\ g_{n+2} \\ \vdots \\ g_{N-1} \end{pmatrix} \\ &= W'_{n+1}[n, 0..N-1] \ \oplus \ W'_{n+1}[n, n+1..N-1] \\ &\qquad \otimes W'_{n+1}[n+1..N-1, 0..N-1] \end{aligned}$$

Let us now consider a partition of W into four submatrices:

$$W'_{n+1} = \begin{pmatrix} A & B \\ C & D \end{pmatrix}$$

such that the size of A is $(n+1) \times (n+1)$, that is

$$\begin{aligned} A &= W'_{n+1}[0..n, 0..n] \\ B &= W'_{n+1}[0..n, n+1..N-1] \\ C &= W'_{n+1}[n+1..N-1, 0..n] \\ D &= W'_{n+1}[n+1..N-1, n+1..N-1] \end{aligned}$$

We have proved in the previous section that

$$\begin{array}{rcll} W'_{n+1} & = & W_0 & \oplus \quad W_0[0..N-1, 0..n] \\ & & & \otimes \quad (W_0[0..n, 0..n])^* \\ & & & \otimes \quad W_0[0..n, 0..N-1] \end{array}$$

and

$$W'_N = W_0 \ \oplus \ W_0 \otimes W_0^* \otimes W_0$$

By substituting the partition of W into this formula, we derive:

$$\begin{array}{rcl} W'_{n+1} & = & \begin{pmatrix} A & B \\ C & D \end{pmatrix} \oplus \begin{pmatrix} A \\ C \end{pmatrix} \otimes A^* \otimes \begin{pmatrix} A, & B \end{pmatrix} \\ & = & \begin{pmatrix} A \ \oplus \ A \otimes A^* \otimes A & B \ \oplus \ A \otimes A^* \otimes B \\ C \ \oplus \ C \otimes A^* \otimes A & E \end{pmatrix} \end{array}$$

where $E = D \ \oplus \ C \otimes A^* \otimes B$.

Using the equality

$$W^* = I \ \oplus \ W \otimes W^* = I \ \oplus \ W^* \otimes W$$

that holds for every matrix over a closed semiring, one may easily prove that the following four equalities hold:

$$\begin{array}{l} W'_{n+1} \ \oplus \ W'_{n+1} \otimes (W'_{n+1})^* \otimes W'_{n+1} = (W'_{n+1})^* \otimes W'_{n+1} \\ A \otimes A^* \otimes A = A^* \otimes A \\ B \ \oplus \ A \otimes A^* \otimes B = A^* \otimes B \\ C \ \oplus \ C \otimes A^* \otimes A = C \otimes A^* \end{array}$$

By substituting these equalities into the expression for W'_{n+1}, we obtain:

$$W'_{n+1} = \begin{pmatrix} A^* \otimes A & A^* \otimes B \\ C \otimes A^* & E \end{pmatrix}$$

and, using the definition of the closure of W_0,

$$\begin{array}{rcl} W'_N & = & W_0 \ \oplus \ W_0 \otimes W_0^* \otimes W_0 \\ & = & (W'_{n+1})^* \otimes W'_{n+1} \\ & = & \begin{pmatrix} A^* \ \oplus \ A^* \otimes B \otimes E^* \otimes C \otimes A^* & A^* \otimes B \otimes E^* \\ E^* \otimes C \otimes A^* & E^* \end{pmatrix} \otimes \begin{pmatrix} A & B \\ C & D \end{pmatrix} \end{array}$$

Then

$$W_{n+1}[n, 0..N-1] = \Big(\ (A[n, 0..N-1])^* \otimes A, \ \ (A[n, 0..N-1])^* \otimes B \ \Big)$$

$$W_{n+1}[n, n+1..N-1] = (A[n, 0..N-1])^* \otimes B$$

$$W_N[n, n+1..N-1]$$
$$= \Big(\ E^* \otimes C \otimes A^* \otimes A \ \oplus \ E^* \otimes C, \ \ E^* \otimes C \otimes A^* \otimes B \ \oplus \ E^* \otimes D \ \Big)$$
$$= \Big(\ E^* \otimes C \otimes A^*, \ \ E^* \otimes E \ \Big)$$

and then

$$\begin{aligned} g_n &= W'_{n+1}[n, 0..N-1] \ \oplus \ W'_{n+1}[n, n+1..N-1] \\ &\qquad \otimes W'_{n+1}[n+1..N-1, 0..N-1] \\ &= \Big((A[n, 0..N-1])^* \otimes A \ \oplus \ (A[n, 0..N-1])^* \otimes B \otimes E^* \otimes C \otimes A^*, \\ &\qquad (A[n, 0..N-1])^* \otimes B \ \oplus \ (A[n, 0..N-1])^* \otimes B \otimes E^* \otimes E \Big) \\ &= \Big((A[n, 0..N-1])^* \otimes A \ \oplus \ (A[n, 0..N-1])^* \otimes B \otimes E^* \otimes C \otimes A^*, \\ &\qquad (A[n, 0..N-1])^* \otimes B \otimes E^* \Big) \end{aligned}$$

By comparing this expression with the formula for W'_N, we obtain

$$\begin{aligned} W'_N[n, 0..N-1] &= \Big((A[n, 0..N-1])^* \otimes A \\ &\qquad \oplus (A[n, 0..N-1])^* \otimes B \otimes E^* \otimes C \otimes A^* \otimes B \\ &\qquad \oplus (A[n, 0..N-1])^* \otimes B \otimes E^* \otimes C, \\ &\qquad\qquad (A[n, 0..N-1])^* \otimes B \\ &\qquad\qquad \oplus (A[n, 0..N-1])^* \otimes B \otimes E^* \otimes C \otimes A^* \otimes B \\ &\qquad\qquad \oplus (A[n, 0..N-1])^* \otimes A \\ &\qquad\qquad \oplus (A[n, 0..N-1])^* \otimes B \otimes E^* \otimes D \Big) \\ &= g_n \end{aligned}$$

as desired.

It follows from the claim that after the execution of the Gauss-Jordan algorithm,

$$G = I \oplus \begin{pmatrix} g_0 \\ g_1 \\ \vdots \\ g_{N-1} \end{pmatrix} = I \oplus W'_N = W_0^*$$

3.3 Dijkstra's algorithm

The algorithm suggested by Dijkstra works only for Dijkstra semirings, but allow us to solve the *single-source algebraic path problem*, that is to find the set of sum-weight functions $d(m,i)$ for paths from a fixed vertex m to all other vertices $i = 0, 1, .., N-1$ in $O(N^2 \cdot (T_\oplus + T_\otimes))$ time, which is N times faster than in the case of finding the sum-weight functions for all pairs of vertices. For example, Dijkstra's algorithm may be used to find the lengths of shortest paths from a fixed vertex to all other vertices in $O(N^2)$ time. To solve the all-pairs algebraic path problem, we just run Dijkstra's algorithm N times, once for every vertex.

As we have shown in Subsection 2.4.3, an addition operation $\oplus$ in a Dijkstra semiring may be viewed as the minimum-operation, and all elements of the semiring are totally ordered. Also, it is convenient to view the multiplication $\otimes$ in a Dijkstra semiring as the usual addition. These two substitutions reduce the algebraic path problem in a Dijkstra semiring to the shortest path problem. While this problem is less general than Dijkstra's original problem, it has exactly the same solution, while the correctness proof for the shortest path problem is more intuitive and therefore easier to understand. To obtain the correctness proof for Dijkstra's general algorithm, it is enough to replace **min** by $\oplus$, $+$ by $\otimes$, and 0 by $\bar{1}$ in the proof suggested below. (We should keep in mind that while $+$ is commutative, a generalized operation $\otimes$ may not be commutative. However, we do not use commutativity of addition in the correctness proof.)

In the following algorithm, $W_0[0..N-1, 0..N-1]$ is a matrix of weights of edges, and m is a fixed vertex for which we compute the lengths of the shortest paths to other vertices. Array $D[0..N-1]$ initially keeps the weights of edges from m to other vertices, and after the execution of the algorithm contains the lengths of the shortest paths from m. In other words, after the execution of the algorithm, $D[0..N-1]$ is equal to the m-th row of the closure W_0^* of W_0. The set T is used to keep the indices of the already processed

vertices.

1. $T := \{m\}$;
2. **for** $n := 0$ **to** $N-1$ **do** $D[n] := W_0[m,n]$;
3. $D[m] := 0$;
4. **for** $n := 1$ **to** $N-1$ **do**
   begin
   $\{P(T)\}$
5. Choose j such that $D[j]$ is the minimal
   element of $\{D[i] \mid 0 \leq i < N$ **and** $i \notin T\}$;
6. **for** $i := 0$ **to** $N-1$ **do**
   $D[i] := D[i]$ **min** $(D[j] + W_0[j,i])$;
7. $T := T \cup \{j\}$
   $\{P(T \cup \{j\})\}$
   end

To prove the correctness of Dijkstra's algorithm, we denote the length of the shortest path from m to i by $d(i)$, and the index of the smallest element of the set $\{D[i] \mid 0 \leq i < N$ **and** $i \notin T\}$ by $\mathrm{Min}(T)$, that is $\mathrm{Min}(T)$ is a natural number such that

$$0 \leq \mathrm{Min}(T) < N \ \wedge \ \mathrm{Min}(T) \notin T, \ \text{and}$$
$$(\forall i : 0 \leq i < N \ \wedge \ i \notin T : D[i] \geq D[\mathrm{Min}(T)])$$

(Observe that $\mathrm{Min}(T)$ is *not* the minimal element of the set T, so the notation is somewhat misleading. However, a more self-explaining notation would be inconveniently long.) The invariant is

$$\begin{aligned} P(T) \ \equiv \quad & (\forall i : 0 \leq i < N : D[i] \geq d(i)) \\ \textbf{and} \quad & T = \{i \mid d(i) \leq d(\mathrm{Min}(T)\} \\ \textbf{and} \quad & (\forall i : i \in T : D[i] = d(i)) \\ \textbf{and} \quad & D[\mathrm{Min}(T)] = d(\mathrm{Min}(T)) \end{aligned}$$

To say this in English, $\mathrm{Min}(T)$ is the closest vertex to m among the vertices which do not belong to T, and

1. for any vertex i, $D(i)$ is no less than the shortest distance from m to i,

2. T is the set of vertices which are at least as close to m as vertex $\mathrm{Min}(T)$ (in other words, any vertex from T is at least as close to m as any vertex which is not in T),

3. for any vertex i in T the shortest distance is already found, that is $D[i] = d(i)$,

4. for the vertex $\mathrm{Min}(T)$, the shortest distance is also already found, that is $D[\mathrm{Min}(T)] = d(\mathrm{Min}(T))$.

First we have to prove that the invariant holds before the execution of the main loop, that is after executing lines 1–4.

(1) Since for any i, $D[i] = W_0[m, i]$ (assigned in line 2), and clearly the shortest path from m to i is no longer than the length of the direct edge (m, i), we may conclude that $D[i] \geq d(i)$.

(2) $T = \{m\}$ (assigned in line 1), and m is certainly the closest vertex to m among all vertices.

(3) $D[m] = 0$ (assigned in line 3), and therefore $D[m] = d(m)$ (recall that the distance from a vertex to itself is always 0).

(4) To prove that $D[\mathrm{Min}(T)] = d(\mathrm{Min}(T))$, we observe that $\mathrm{Min}(T)$ is the vertex adjacent[5] to m such that the edge from m to $\mathrm{Min}(T)$ is the shortest of edges outcoming from m. This means that any multi-edge path starting from m is at least as long as the length of the edge $(m, \mathrm{Min}(T))$, and therefore the shortest path from m to $\mathrm{Min}(T)$ is the edge $(m, \mathrm{Min}(T))$. Since initially $D[\mathrm{Min}(T)]$ is equal to the length of $(m, \mathrm{Min}(T))$, we conclude that $D[\mathrm{Min}(T)]$ is equal to the length of the shortest path from m to $\mathrm{Min}(T)$.

Now we need to show that the main loop of the algorithm preserves the invariant. So assume that $P(T)$ holds in the beginning of the loop, before line 5. Then after executing line 5, $j = \mathrm{Min}(T)$.

(1) To prove that line 6 preserves part 1 of the invariant, we observe that for every i, $d(i) \leq d(j) + W_0[j, i]$, and by the invariant $P(T)$, $D[i] \geq d(i)$ and $D[j] \geq d(j)$ (it is straightforward to see that $D[j]$ itself is not changed while executing line 6). Therefore, after executing the loop in line 6,

$$\begin{aligned} D[i] &= D[i] \,\mathbf{min}\, (D[j] + W_0[i, j]) \\ &\geq d(i) \,\mathbf{min}\, (d(j) + W_0[i, j]) \end{aligned}$$

[5] We say that i is *adjacent* to m if there is an edge from m to i.

$$\begin{aligned} &\geq d(i) \,\mathbf{min}\, d(i) \\ &= d(i) \end{aligned}$$

(2) Since $j = \mathrm{Min}(T)$ is the closest vertex to m among the vertices that do not belong to T, and, by part 2 of $P(T)$, any vertex from T is closer to m than any vertex not from T, we conclude that any vertex from the set $T \cup \{j\}$ is closer to m than any vertex that does not belong to $T \cup \{j\}$. Thus, part 2 of the invariant holds for the set $T \cup \{j\}$.

(3) By part 4 of the invariant $P(T)$, $D[j] = d(j)$, and by part 3, for all $i \in T$, $D[i] = d(i)$. Therefore, we may conclude that for all $i \in (T \cup \{j\})$, $D[i] = d(i)$, and therefore part 3 of the invariant holds for the set $T \cup \{j\}$.

(4) It remains to show that after executing the loop in line 6, part 4 of the invariant holds for the set $T \cup \{j\}$, that is

$$D[\mathrm{Min}(T \cup \{j\})] = d(\mathrm{Min}(T \cup \{j\})) \quad (*)$$

Let us denote (for the brevity of notation) $\mathrm{Min}(T \cup \{j\})$ by k, that is k is the closest vertex to m among vertices that do not belong to $T \cup \{j\}$. If $D[k] = d(k)$ before the execution of line 6, then it is not changed, because the loop in line 6 does not increase $D[k]$, and part 1 of the invariant guarantees that $D[k]$ cannot become less than $d(k)$. So assume that $D[k] > d(k)$ before line 7, and let $(m, i_1, .., i_a, k)$ be the shortest path from m to k. Then

$$d(k) = d(i_a) + W_0[i_a, k]$$

Then $d(i_a) < d(k)$[6], and therefore either $i_a = j$ or $i_a \in T$. If $i_a = j$, then while executing line 6, we assign

$$\begin{aligned} D[k] &= D[k] \,\mathbf{min}\, (D[j] + W_0[j, k]) \\ &= D[k] \,\mathbf{min}\, (d(j) + W_0[j, k]) \\ &= d(k) \end{aligned}$$

On the other hand, if $i_a \in T$, then the similar assignment was made on the previous stage of the algorithm, when i_a was added to T, and thus

[6]Also, it may happen that $d(i_a) = d(k)$, which leads us to a case analysis that would increase the length of the proof by a page. We do not consider this case here. The interested reader may consider this situation himself or refer to [Lehmann, 1977] for a more rigorous proof. (Lehmann's proof is different from the proof presented here. It uses generalized matrix operations and is similar to the proof in Subsection 3.1.3.)

$D[k] = d(k)$ before executing line 6. Thus, we have shown that the invariant $P(T \cup \{j\})$ holds after executing line 6. Therefore after updating T in line 7, $P(T)$ holds.

Observe that Dijkstra's algorithm is a generalization of both Warshall's algorithm and Floyd's algorithm. It is a generalization in two respects: first, it solves the algebraic path problem in a Dijkstra semiring, which is a generalization of both the transitive closure problem and the algebraic path problem; second, Dijkstra's algorithm allows us not only to solve the all-pairs problem, but also to find efficiently the shortest paths from a single vertex to all other vertices. Moreover, Dijkstra's algorithm is as efficient as Warshall's and Floyd's algorithm.

Chapter 4

Systolic algorithms

4.1 Systolic arrays

4.1.1 Introduction into systolic algorithms

The notion of systolic arrays was introduced by H. Kung and Leiserson in 1978 [H. Kung and Leiserson, 1978a], [H. Kung and Leiserson, 1978b], [H. Kung and Leiserson, 1978c] as a formalization of parallel programming for VLSI. Since that time much effort has been spent on designing efficient systolic algorithms. The systolic model of computation has been found appropriate for matrix computations [H. Kung and Leiserson, 1978b], [H. Kung, 1982], [Kramer and Leeuwen, 1983], dynamic programming [Guibas *et al.*, 1979], priority queues [H. Kung, 1979], [Leiserson, 1979], [H. Kung, 1980], [Kaldewaij and Udding, 1992], and some other problems that may be solved by performing parallel computations on a regular data structure. Several textbooks on systolic programming have been published [S. Kung, 1988], [Quinton, 1990], [Ullman, 1984]. In recent years, a lot of research has been performed on developing theoretical methods for an efficient design of systolic algorithms [S. Kung *et al.*, 1984], [Quinton, 1984], [Culik and Fris, 1984], [Miranker and Winkler, 1984], [Fortes *et al.*, 1985], [Ibarra *et al.*, 1986], [Quinton, 1987], [Bertolazzi *et al.*, 1988], [Martin, 1989], and on methods for efficient translation of sequential and different kinds of parallel algorithms into systolic algorithms [Umeo, 1985], [Navarro *et al.*, 1987], [S. Kung *et al.*, 1987], [Ibarra and Sohn, 1989], and different kinds of systolic algorithms into each other [Kumar and Tsai, 1988]. A good brief summary of different methods of sys-

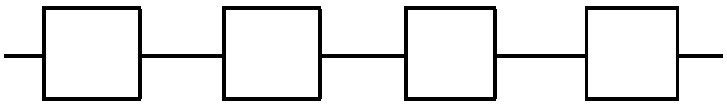


Figure 4.1: Linear systolic array

tolic design and the comparison of systolic arrays with other models of parallel computations may be found in [S. Kung, 1987]. Also, there are several publications on the theory of space-time complexity of systolic computations [Thompson, 1979], [Leighton, 1983], [Li and Wah, 1985].

A *systolic array*[1] is a regular, usually *linear* (Figure 4.1), *orthogonal* (Figure 4.2), or *hexagonal* (Figure 4.3) mesh of processors simultaneously executing *the same* program. Each processor may communicate only with neighbouring processors; it cannot communicate with distant processors, nor with RAM. This is probably the main difference between the systolic model and most other models of parallel computations, such as PRAM [Fortune and Wylie, 1970], [Cormen *et al.*, 1990] (Chapter 30), SIMD [Flynn, 1972], [Kuck, 1977], MIMD [Flynn, 1966], [B. Smith, 1978], [Gosch, 1979], and a probabilistic model of unrestricted computation [Fortune and Wylie, 1970], [Greenberg and Fischer, 1982]: all these models allow direct data exchanges between any two processors in constant time, while the systolic model allows data exchange only between neighbours. This makes the systolic architecture less flexible, but allows a more efficient hardware implementation.

Each processor in the systolic array may keep some values in its own memory, but the size of its memory is $O(1)$, and generally it is considered a good style[2] to keep in each processor an amount of data as small as possible. The data is passed to some processors of a systolic array from outside (presumably, from RAM), and the output is read from some other processors. It is usually best to feed data only into boundary processors, not into inner processors of a mesh, and also to read output only from boundary processors.

[1] The word *systole* is a physiological term referring to the rhythmically recurrent contractions of the heart and arteries. It was probably chosen because data in a systolic array is rhythmically passed from processor to processor, like blood in arteries.

[2] By *good style* we mean that algorithms that obey this rule are considered more valuable. However, the good style may be violated if it prevents us from finding an efficient solution of a problem.

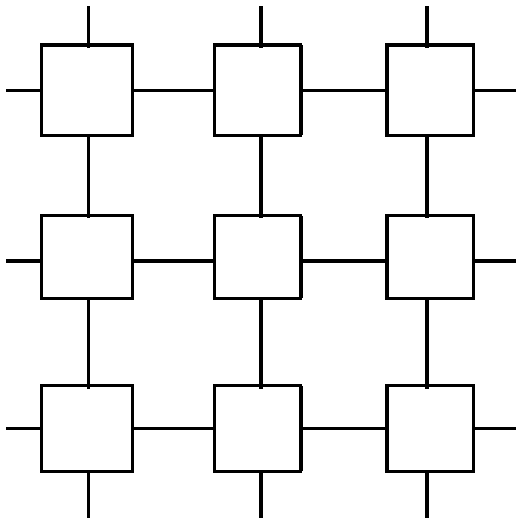


Figure 4.2: Orthogonal systolic array

Each processor in an array knows whether it is on the boundary or not, but it is usually best to assume that inner processors do not know their coordinates in the mesh. The size of a mesh is (usually) *not* constant: it depends on the size of a problem. For example, if the input is an array of the size N, then the mesh usually contains $\Theta(N)$ processors.

A classical example of a systolic algorithm is computing *convolution*: given two sequences of numbers of equal length, say $a_0, .., a_{N-1}$ and $b_0, .., b_{n-1}$, we need to compute the third sequence, $c_0, c_1, .., c_{2N-1}$, each element of which is defined as follows:

$$c_i = \sum_{j=0}^{N-1} a_j \cdot b_{i-j}$$

(If the two initial sequences are the coefficients of two polynomials, then the resulting sequence contains the coefficients of the product of these polynomials.) The solution of this valuable exercise is given in almost all textbooks and technical reports on systolic arrays (see for example [Ullman, 1984] or [S. Kung, 1988]), and here we refrain from repeating it once more. It may be solved on a linear systolic array with N processors in $O(N)$ time. (The

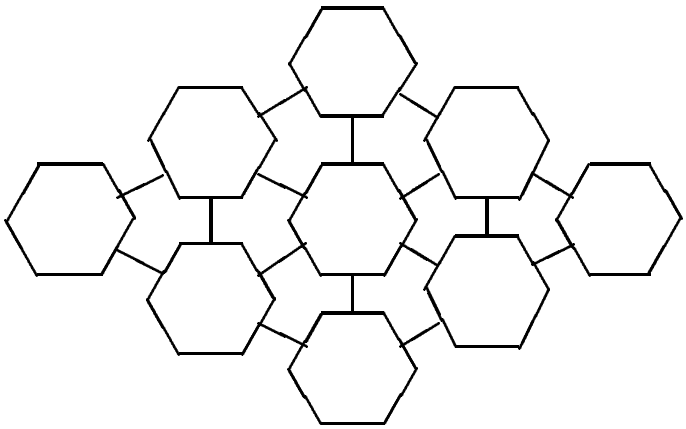


Figure 4.3: Hexagonal systolic array

running time of a systolic computation is the time from the moment when the first piece of data is fed into the systolic array to the moment when the last piece of output is received.) Below we present a little bit less trivial example of a systolic computation that should give you an idea how systolic arrays work.

4.1.2 Multiplying a matrix by a vector

Given a square matrix (a_{ij}) of the size $N \times N$, and a vector (b_k) of the size N, we wish to compute their product:

$$\begin{pmatrix} a_{00} & a_{01} & \dots & a_{0(N-1)} \\ a_{01} & a_{11} & \dots & a_{1(N-1)} \\ \vdots & \vdots & \vdots & \vdots \\ a_{(N-1)1} & a_{(N-1)2} & \dots & a_{(N-1)(N-1)} \end{pmatrix} \times \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_{N-1} \end{pmatrix} = \begin{pmatrix} \sum_{n=0}^{N-1} a_{0n} \cdot b_n \\ \sum_{n=0}^{N-1} a_{1n} \cdot b_n \\ \vdots \\ \sum_{n=0}^{N-1} a_{(N-1)n} \cdot b_n \end{pmatrix}$$

The systolic algorithm for the matrix-vector multiplication, designed by Leiserson, appeared in the very first publication on systolic arrays [H. Kung and Leiserson, 1978b], and was presented later in detail in Leiserson's PhD thesis [Leiserson, 1981].

The algorithm uses a linear array with $(2 \cdot N - 1)$ processors. Figure 4.4

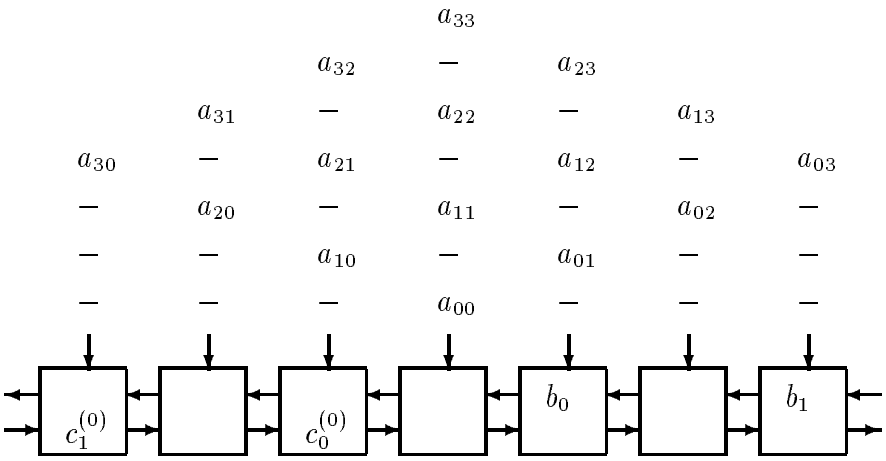


Figure 4.4: Systolic array for multiplying a matrix by a vector

shows the array for the case $N = 4$. We compute the product (c_m) of (a_{ij}) and (b_k) in N steps, using the following recursive equation:

$$c_m^{(0)} = 0$$
$$c_m^{(n+1)} = c_m^n + a_{mn} \cdot b_n$$
$$c_m = c_m^{(N)}$$

The algorithm operates as follows:

- The coefficients of the matrix (a_{ij}) enter the array from above, as shown on Figure 4.4. Each processor receives one new element every two beats.
- The components of the vector (b_k) flow unchanged from right to left.
- The components of the vector (c_m) flow from left to right. Their initial values are 0, and, passing through the array, each c_m accumulates its partial products $a_{m1} \cdot b_1, a_{m2} \cdot b_2, .., a_{m(N-1)} \cdot b_{N-1}$.

So, at each step, each processor performs the following operations:

1. Get a_{ij} from the upper input channel, b_j from the right neighbour, and c_i from the left neighbour. (It is straightforward to check that if we pass the data in the indicated order, then b_j and c_i indeed come to the processor simultaneously with a_{ij}.)

2. Compute $c_i := c_i + a_{ij} \cdot b_j$.

3. Pass b_j to the right neighbour and c_i to the left neighbour.

All values are moved through the systolic array with *the same speed*, that is at the *beat* of some clock all processors simultaneously pass data to their neighbours, and receive new data from their neighbours and from upper channels. Then each processor executes a step of the algorithm and waits for the next beat to pass and read the data again.

To present the algorithms executed by processors of the systolic array in a formal fashion, we use the notation developed by Hoare [Hoare and Jones, 1989] for parallel algorithms. In this notation

left? a

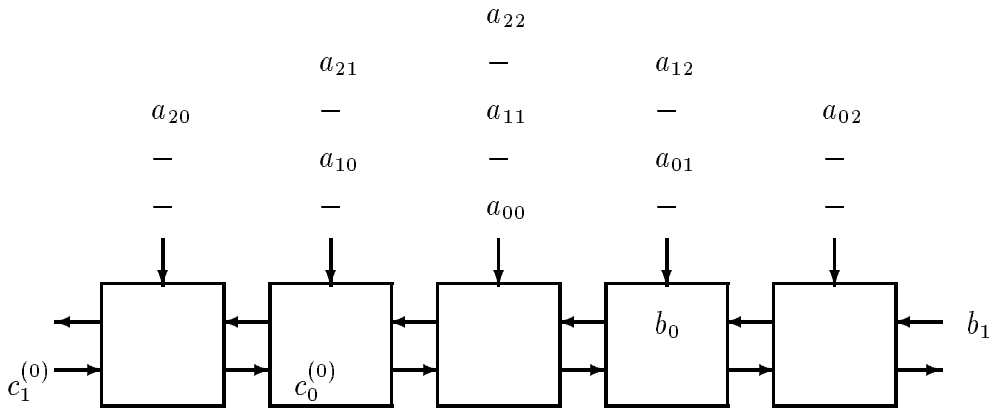
means that a processor obtains the value a by reading the input from its left neighbour (or, if the processor does not have the left neighbour, from outside). If there is nothing to read, the processor is blocked until it receives the desired input. Similarly, the operator

right! a

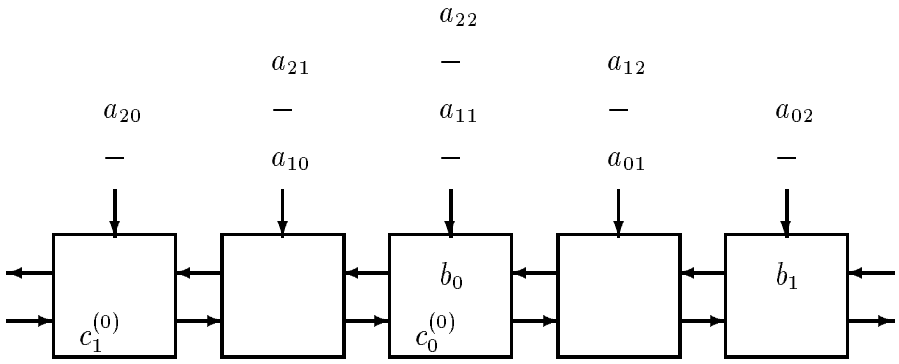
means that a processor passes the value of a to its right neighbour. If the right neighbour is not ready to read, the processor is blocked until the value is read. The other notation is the same as the notation for the sequential algorithms. The memory of each processor in our array consists of three variables, a, b, and c, that keep respectively the current value of a_{ij}, b_j, and c_i. Below we present the formal description of the algorithm.

```
|[var a, b, c: Real;
  right? b; left? c; upper? a
  ;c := c + a · b
  ;left! b; right! c
]|
```

One may check that the total number of clock beats required to compute the product is $(4 \cdot n - 3)$. The steps of the algorithm for $N = 3$ are shown on Figure 4.5.

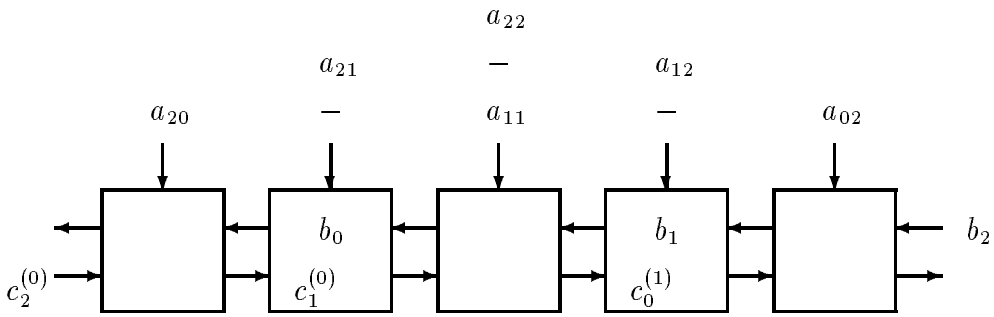


Beat 1

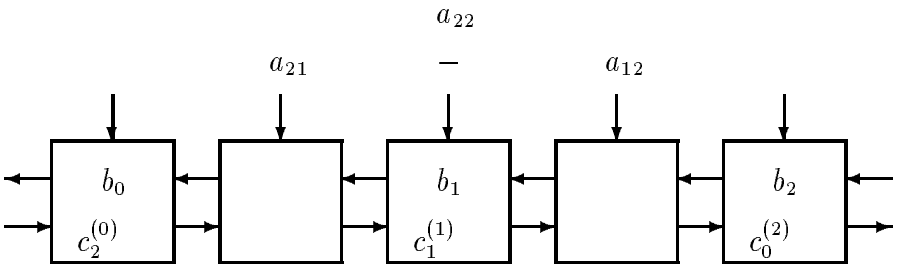


Beat 2

Figure 4.5: Steps of matrix-vector multiplication

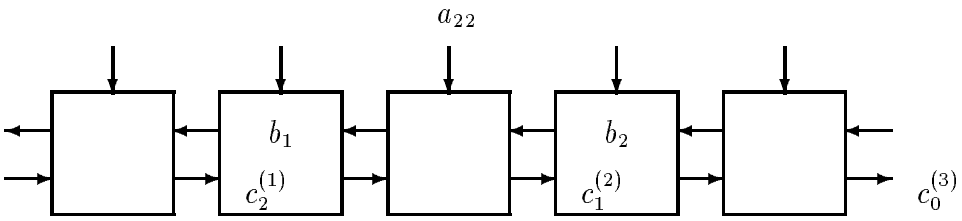


Beat 3

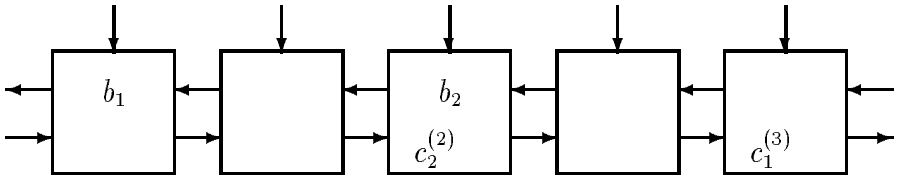


Beat 4

Figure 4.5 (continued): Steps of matrix-vector multiplication



Beat 5



Beat 6

Figure 4.5 (continued): Steps of matrix-vector multiplication

4.2 Instances of the algebraic path problem

4.2.1 The transitive closure problem

Some graph algorithms for models of computation close to a systolic array were designed in the early Seventies (e.g. [Kautz and Levitt, 1972]), that is several years before the invention of the systolic model. The first algorithm for computing the transitive closure on a systolic array was designed by Guibas, H. Kung, and Thompson in 1979 [Guibas *et al.*, 1979]. Later the algorithm has been improved by Atallah and Kosaraju [Atallah and Kosaraju, 1982], Lin and Wu [Lin and Wu, 1985], S. Kung, Lewis, and Lo [S. Kung and Lo, 1985], [S. Kung *et al.*, 1986], [S. Kung *et al.*, 1987], Nunez and Torraba [Nunez and Torraba, 1987], and T. Lang and Moreno [T. Lang and Moreno, 1988]. All these improvements concerned the style and the number of bit operations performed by the algorithm, but the asymptotic running time has remained the same, namely $O(N)$, and the number of processors used by the algorithm has remained N^2. A comparison of different systolic algorithms for the transitive closure problem may be found in [S. Kung *et al.*, 1987].

T. Lang and Moreno suggested a method for computing the transitive closure of large graphs on a small systolic array [T. Lang and Moreno, 1988]. Their method leads to transitive closure algorithms for both linear and orthogonal arrays. Some other methods, designed for the algebraic path problem, also allow us to solve the transitive closure problem on small systolic arrays. E.g. [Nunez and Valero, 1988] described a method for solving the algebraic path problem of arbitrary size on a fixed-size systolic array.

An interesting version of a systolic algorithm for the transitive closure problem was presented by H. Lang in 1988 [H. Lang, 1988]. This algorithm uses an *instruction systolic array*, which is characterized by a systolic flow of instructions (instead of data as in standard systolic arrays) [Kunde *et al.*, 1986], [H. Lang, 1986], [H. Lang, 1987]. The running time and the size of H. Lang's systolic array is the same as in usual systolic algorithms.

Is $O(N)$ the optimal running time? S. Kung, Lewis, and Lo have shown that this running time *is* optimal for systolic arrays of size $O(N^2)$ if a graph is represented by the adjacency matrix [S. Kung *et al.*, 1987]. However, there are other ways to represent a directed graph, most common of which is the *adjacency list* ([Cormen *et al.*, 1990], pages 455–456). It is not known how

efficient systolic algorithms may be on a graph represented by the adjacency list. I did not find any publications on such algorithms.

To find the transitive closure of a graph G_0 with an adjacency matrix $W_0[0..N-1, 0..N-1]$, we use an orthogonal systolic array of $N \times N$ processors (see Figure 4.6). The processor in the i-th row and the j-th column is denoted by P_{ij}. Every processor contains a logical variable, also denoted by P_{ij}, which permanently remains in this processor. Initially for all i and j, $P_{ij} = 0$.

The initial matrix is obtained from the adjasency matrix of a given graph by replacing all its diagonal elements with 1's. We pass two copies of the matrix through the systolic array as shown on Figure 4.6. All values are moved through the systolic array with the same speed. Observe that $W[1,0]$ enters the array one beat later than $W[0,0]$, $W[2,0]$ one beat later than $W[1,0]$, and so on. At each step, each processor P_{ij} performs the following operations:

1. Get $W[i,n]$ from the left neighbour, and $W'[n,j]$ from the upper neighbour. (It is easy to verify that for any n, $W[i,n]$ and $W'[n,j]$ indeed reach P_{ij} at the same time.)

2. Compute

   $$P_{ij} := P_{ij} \vee (W[i,n] \wedge W'[n,j])$$

   Intuitively, we set P_{ij} to 1 if $W[i,n] = W'[n,j] = 1$, and leave it unchanged otherwise.

3. If $n = i$, that is $W[i,j]$ has arrived to P_{ij}, then set

   $$W'[i,j], P_{ij} := W[i,j] \vee P_{ij}$$

   That is after $W[i,j] \vee P_{ij}$ has been computed, it is assigned to both $W'[i,j]$ and P_{ij}. Similarly, if $n = j$, then set

   $$W'[i,j], P_{ij} := W'[i,j] \vee P_{ij}$$

4. Pass $W[i,n]$ to the right neighbour, and $W'[n,j]$ to the lower neighbour.

Below we present the same algorithm (for the processor P_{ij}) in a formal fashion. (The standard syntax of the operator **if**...**fi** is slightly violated in order to improve readability.)

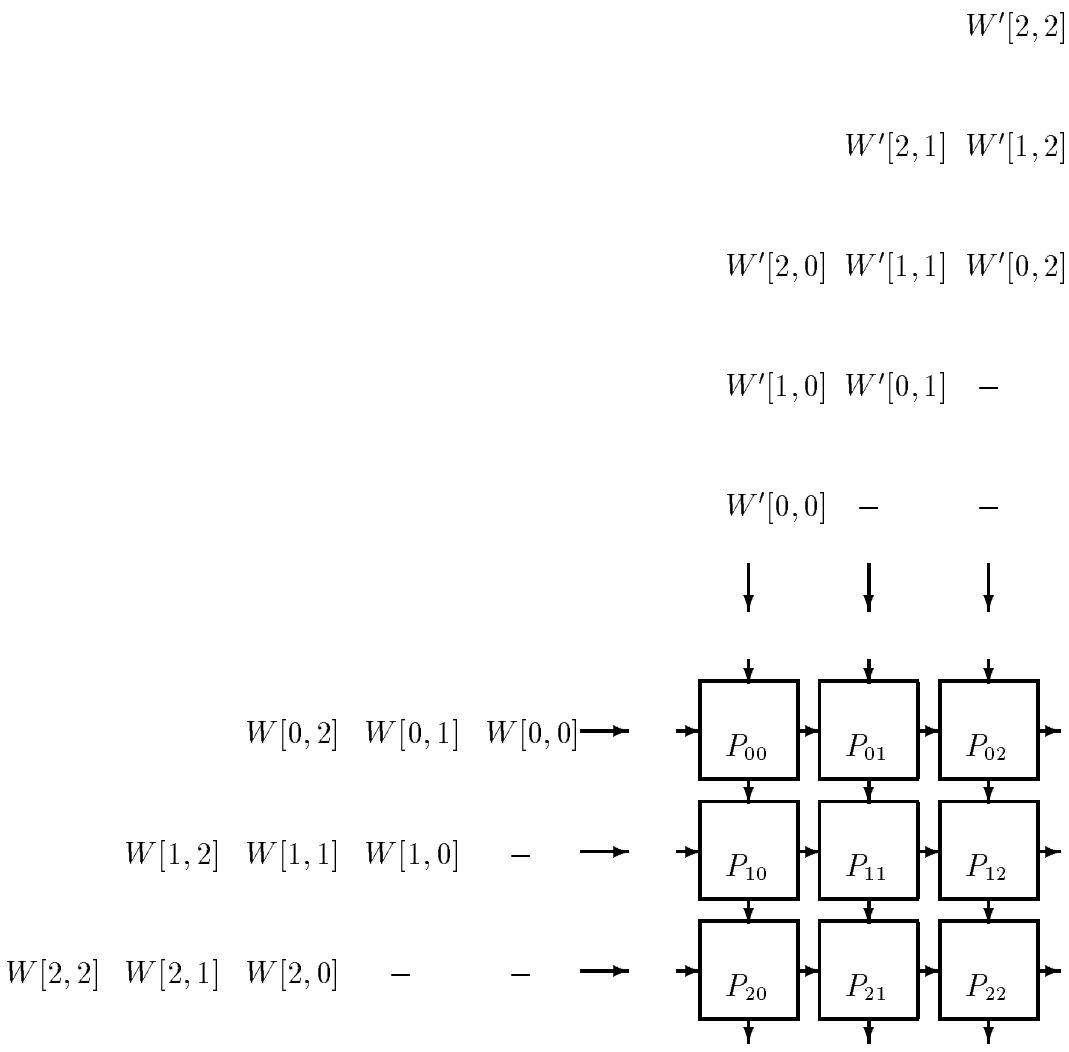


Figure 4.6: Computing the transitive closure

```
|[var W, W', P: Boolean;
  left? W; upper? W'
  ;P := P ∨ (W ∧ W')
  ;if W contains the value of W[i,j] → W := W ∨ P; P := W ∨ P fi
  ;if W' contains the value of W'[i,j] → W' := P ∨ W'; P := P ∨ W' fi
  ;right! W; lower! W'
]|
```

It remains to replace the condition "*W contains the value of* $W[i,j]$" with some formal code. A trivial solution is to keep in each processor its coordinates, and to pass the indices of every element W together with its value. However, this solution requires huge additional memory. A more efficient way to solve this problem is described in [Ullman, 1984].

After the data have passed once through the systolic array, we feed them back into the array in the same order as the first time, and repeat the same operation of passing data through the array. Then we repeat the same pass a third time. In other words, as an element completes the first pass, by reaching the right or bottom edge of the array, it is immediately fed back to the left or top edge, to begin the next pass.

We claim that, after three identical passes described above, P become an adjacency matrix of the transitive closure of the initial graph, that is for all i and j, $P_{ij} = 1$ if and only if there is a path from i to j in the initial graph. It is easy to check that the algorithm is executed in $(5 \cdot N - 2)$ beats. There remains the problem to output the matrix P, which may be done in N additional beats, bringing the total computation time to $(6 \cdot N - 2)$.

We prove the correctness of the algorithm in a series of claims. Again, we denote by $(i \rightarrow^{<n} j)$ a path from i to j all intermediate vertices of which are less than n.

Claim 1 Suppose *beat* 0 of a pass of our algorithm is defined to be when $W[0,0]$ and $W'[0,0]$ arrive at P_{00}. Then $W[i,j]$ arrives at P_{in} at *beat* $(i+j+n)$, and $W[i,j]$ arrives at P_{ni} at *beat* $(i+j+n)$.

A proof of this claim is straightforward, just by counting the number of "steps" in the systolic array that $W[i,j]$ makes to arrive to P_{in}.

Claim 2 If there is a path $(i \rightarrow^{<(i \,\mathbf{min}\, j)} j)$ in G_0, then after the first pass $W[i,j] = W'[i,j] = P_{ij} = 1$.

Proof. We prove the result by induction on the length of the shortest path of the form $(i \rightarrow^{<(i \,\mathbf{min}\, j)} j)$. Our induction hypothesis states that if there is a path from i to j, then P_{ij} is set to 1 before *beat* $(i+j+(i \,\mathbf{min}\, j))$, which is before either $W[i,j]$ or $W'[i,j]$ reaches it.

For paths of lengths 0 and 1 there is nothing to prove, since $W[i,j]$ and $W'[i,j]$ are initially equal to 1, and the algorithm never changes $W[i,j]$ or $W'[i,j]$ from 1 to 0.

Now suppose that there is a multi-edge path $(i \rightarrow^{<(i \,\mathbf{min}\, j)} j)$, and let k be the largest intermediate vertex on this path. Since k is larger than any other intermediate node of the path, the path may be divided into two shorter paths, $(i \rightarrow^{<(i \,\mathbf{min}\, k)} k)$ and $(k \rightarrow^{<(k \,\mathbf{min}\, j)} j)$ (see Figure 4.7a). By the inductional hypothesis, $W[i,k]$ is set to 1 at or before the time it arrives at P_{ik} (which happens at *beat* $(i+2\cdot k)$) and $W'[k,j]$ is set to 1 before it arrives at P_{kj}. Then at *beat* $(i+j+k)$ (which is later, because $k < i,j$), $W[i,k]$ and $W'[k,j]$ both arrive at P_{ij}, and P_{ij} is set to 1. When later $W[i,j]$ and $W'[i,j]$ arrive at P_{ij} (*not* at the same time), they are also set to 1. □

Claim 3 If there is a path $(i \rightarrow^{<(i \,\mathbf{max}\, j)} j)$ in G_0, then after the second pass $W[i,j] = W'[i,j] = P_{ij} = 1$.

Proof. Again, we proceed by induction on the length of the shortest path from i to j of the from $(i \rightarrow^{<(i \,\mathbf{max}\, j)} j)$. The induction hypothesis is that

- If there is a path of the form $(i \rightarrow^{<j} j)$ from i to j, then $W[i,j]$ and P_{ij} are set to 1 by *beat* $(i+2\cdot j)$.
- If there is a path of the form $(i \rightarrow^{<i} j)$ from i to j, then $W[i,j]$ and P_{ij} are set to 1 by *beat* $(2\cdot i+j)$.

If the length of the path is 1, then all three variables are set to 1 during the first path. Suppose there is a multi-edge path of the form $(i \rightarrow^{<j} j)$, and let k be the largest intermediate vertex on the path. Then the path $(i \rightarrow^{<j} j)$ may be decomposed into two paths, $(i \rightarrow^{<(i \,\mathbf{max}\, k)} k)$ and $(k \rightarrow^{<(k \,\mathbf{min}\, j)} j)$ (see Figure 4.7b). By inductional hypothesis $W[i,k]$ is set to one at *beat* $(i+2\cdot k)$, when it arrives at P_{ij}, and by Claim 1, $W'[k,j]$ set to 1 during the first pass. Thus at *beat* $(i+j+k)$, which is later then $(i+2\cdot k)$, $W[i,k]$ and $W'[k,j]$ meet at P_{ij}, and set P_{ij} to 1. The case when i and j are connected by a path of the form $(i \rightarrow^{<i} j)$ is treated similarly. □

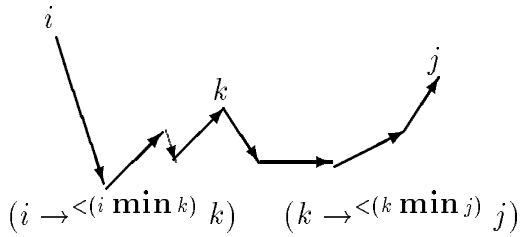
Claim 4 If there is a path from i to j in G_0, then after the third pass $P_{ij} = 1$.

Assume there is a path from i to j, and let k be the largest intermediate node of this path. If all its intermediate nodes are less than $(i \max j)$, then P_{ij} is set to 1 during the second pass. So assume that $k > i, j$. Then the path may be divided into two shorter paths, $(i \rightarrow^{<k} k)$ and $(k \rightarrow^{<k} j)$ (see Figure 4.7c). Then by Claim 3, $W[i,k]$ and $W'[k,j]$ are set to 1 during the second pass, and when they meet at P_{ij} at the third pass, they set P_{ij} to 1. □

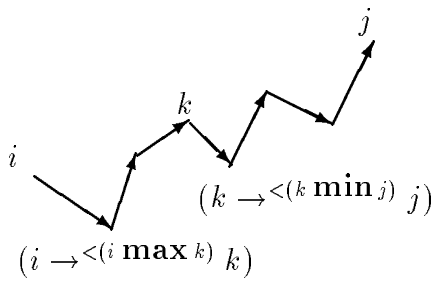
4.2.2 The all-pairs shortest path problem

It is hard to say who first found the solution of the all-pairs shortest path problem on a systolic array, because a shortest path algorithm is a trivial generalization of a transitive closure algorithm, so trivial that it is probably meaningless to speak about the author of this generalization. In their first publication on a systolic shortest path algorithm, Guibas, H. Kung, and Thompson noted that their algorithm is "likely to be also applicable to any other problem with the same data flow. A large class of such problems, called shortest path problems, is discussed in [Aho *et al.*, 1974]" ([Guibas *et al.*, 1979], page 520). However, they did not show how their algorithm may be adapted for the shortest path problem. It seems that after this remark systolic shortest path algorithms were forgotten for the next three years (I did not find any publication on this problem in 1980 or 1981). The problem was considered again in [Atallah and Kosaraju, 1982], and then in [Moldovan, 1983]. Starting in 1985, different modifications of systolic shortest path algorithms have begun to appear in the literature *quantum satis* (in plenty). Examples are [Rote, 1985], [S. Kung and Lo, 1985], [Lewis and S. Kung, 1986], and [Lakhani and Dorairaj, 1987]. [Schwiegelshohn and Thiele, 1986] presents an algorithm that not only computes the lengths of the shortest paths, but also finds the shortest paths themselves, with the size of the systolic array still $O(N^2)$, and the running time still $O(N)$. A clear formal description of a systolic solution of the shortest path problem, together with some results on the lower bound of the space-time complexity and a survey of different shortest path systolic algorithms is presented in [S. Kung *et al.*, 1987].

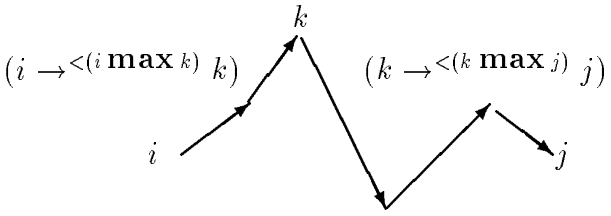
We show how to obtain an all-pairs shortest path algorithm by modifying



(a) $(i \rightarrow^{<(i\ \mathbf{min}\ j)}\ j)$



(b) $(i \rightarrow^{<j}\ j)$



(c) $(i \rightarrow j)$

Figure 4.7: Subpaths of a path $(i \rightarrow j)$

the transitive closure algorithm presented in the previous subsection. To compute the shortest paths in the graph G_0 with the adjacency matrix W_0 (where $W_0[i,j]$ is the length of the edge from i to j, ∞ if there is no edge from i to j, and 0 if $i = j$), we use the same systolic array as those used for the shortest path problem, and again we pass two copies of the matrix W through this array, three times and in the same order. However, we slightly modify the algorithm performed by every processor in the systolic array.

The matrices W, W', and P are now real-valued. Initially, for all i and j, P_{ij} is set to infinity. We replace the operation $\vee$ by **min**, and $\wedge$ by $+$. The following is the description of the shortest-path algorithm for each processor in the array.

1. Get $W[i,n]$ from the left neighbour, and $W'[n,j]$ from the upper neighbour.
2. Compute $P_{ij} := P_{ij}$ **min** $(W[i,n] + W'[n,j])$.
3. If $n = i$, that is $W[i,j]$ has arrived to P_{ij}, than set

   $$W[i,j], P_{ij} := W[i,j] \textbf{ min } P_{ij}$$

   Similarly, if $n = j$, then set

   $$W'[i,j] := W'[i,j] \textbf{ min } P_{ij}$$

4. Pass $W[i,n]$ to the right neighbour, and $W'[n,j]$ to the lower neighbour.

Below we present the same algorithm (for the processor P_{ij}) in a formal fashion.

```
|[var W, W', P: Real;
  left? W; upper? W'
  ;P := P min (W + W')
  ;if W contains the value of W[i,j]
  → W := W min P; P := W min P
  fi
  ;if W' contains the value of W'[i,j]
  → W' := P min W'; P := P min W'
  fi
  ;right! W; lower! W'
]|
```

The proof that this modification computes the shortest paths is completely analogous to the proof in the previous section. The same proof shows that by replacing $\vee$ in the shortest path algorithm by the generalized addition $\oplus$, and $\wedge$ by the generalized multiplication $\otimes$, we obtain a systolic algorithm for computing the closure of a matrix over a Dijkstra semiring. In particular, this algorithm may be used for solving the tunnel problem. Unfortunately, this algorithm has not been adapted for solving the general algebraic path problem, and it probably cannot be adapted. Some reasons for this are given in [Quinton, 1990] (page 128).

4.2.3 Matrix inversion

Some systolic algorithms for operations over matrices, including an algorithm for matrix inversion, were presented in the very first papers on systolic arrays, [H. Kung and Leiserson, 1978c] and [H. Kung, 1979], and later a modification of these algorithms was discussed in [Mead and Conway, 1980]. [Kramer and Leeuwen, 1983] presented a systolic algorithm for computing the inverse of a matrix using the Gauss-Jordan elimination method. Modifications of Kramer and Leeuwen's algorithm were later presented in [Nash and Hansen, 1984] (based on the systolic array described in [Nash *et al.*, 1981]) and [Robert, 1987].

In 1985 Rote has published the first article on the general algebraic path problem [Rote, 1985], where he described a new systolic implementation of the Gauss-Jordan elimination algorithm for the matrix inversion problem and showed how to modify this algorithm for solving the algebraic path problem. Some lower-bound results for the Gauss-Jordan elimination and the optimal algorithms based on these lower bounds are presented in [Louka and Tchuente, 1989] and [Benaini and Robert, 1990b].

We build a systolic algorithm for the Gauss-Jordan method by adapting the sequential algorithm described in Subsection 3.2.1 for parallel computations on a systolic array. For the convenience of the reader, the algorithm from Subsection 3.2.1 is presented in Table 4.1.

We use the systolic array with $N \cdot (N+1)$ processors, originally designed by Gentleman and Kung for matrix triangularization [Gentleman and Kung, 1981]. The array is shown in Figure 4.8. The array is composed of N rows. Each row n contains $(N+1)$ processors, numbered from left to right

for $n := 0$ **to** $N-1$ **do**
begin
{Compute J_n, store $J_n[0..N-1,n]$ in $C_{n+1}[0..N-1,n]$}
$C_{n+1}[n,n] := 1/C_{n+1}[n,n]$; (1)
for $i := 0$ **to** $N-1$ **do**
if $i \neq n$
then $C_{n+1}[i,n] := -C_n[i,n] \cdot C_{n+1}[n,n]$; (2)
{Compute $C_{n+1}[0..N-1, n+1..2 \cdot N-1]$}
for$j := n+1$ **to** $2 \cdot N-1$ **do**
begin
$C_{n+1}[n,j] := C_{n+1}[n,n] \cdot C_n[n,j]$; (3)
for $i := 0$ **to** $N-1$ **do**
if $i \neq n$ (4a)
$C_{n+1}[i,n] := C_n[i,j] + C_{n+1}[i,n] \cdot C_n[n,j]$ (4b)
end
end

Table 4.1: A sequential algorithm for the matrix inversion

$P_{n0}, P_{n1}, .., P_{nN}$. The matrix W_0 followed by the matrix I is fed into the array as shown on Figure 4.8.

The Gauss-Jordan algorithm consists of N steps. We implement each step on one row of the systolic array. For all $n \in [0..N-1]$, the n-th execution of the outer loop is implemented by the n-th row of the array. Thus, the n-th row takes the matrix $C_n[0..N-1, n..2 \cdot N-1]$ and produces the matrix $C_{n+1}[0..N-1, n+1..2 \cdot N-1]$.

The step of the algorithm performed by the n-th row consists of two stages: first, the generation of the matrix J_n, and then the computation of the last $(2 \cdot N - n - 1)$ columns of the product $C_{n+1} = J_n \cdot C_n$. As soon as N elementary matrices J_n are computed and stored, each J_n in the n-th row, the array is initialized. It then functions as a linear operator which transforms the matrix I into W^{-1}. If we enter some other $N \times M$ matrix V after I or instead of I, the algorithm will compute $W^{-1} \cdot V$.

Below we describe the operations performed by three kinds of processors in the array. The Boolean variable *init* in every processor is used to determine whether the current input is the first input into the processor during the algorithm execution. Initially, *init* is set to 1. After the first input is

			1
		0	0
	0	1	0
0	0	0	0
0	1	0	$W_0[3,3]$
0	0	$W_0[2,3]$	$W_0[3,2]$
1	$W_0[1,3]$	$W_0[2,2]$	$W_0[3,1]$
$W_0[0,3]$	$W_0[1,2]$	$W_0[2,1]$	$W_0[3,0]$
$W_0[0,2]$	$W_0[1,1]$	$W_0[2,0]$	–
$W_0[0,1]$	$W_0[1,0]$	–	–
$W_0[0,0]$	–	–	–

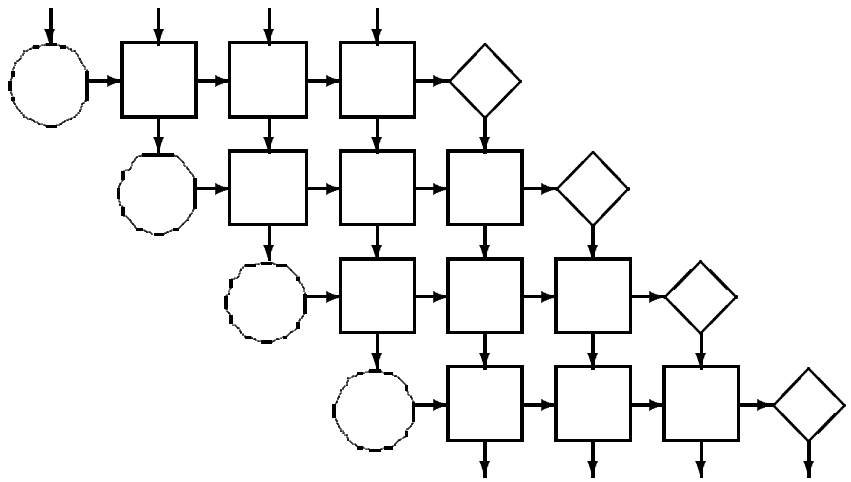
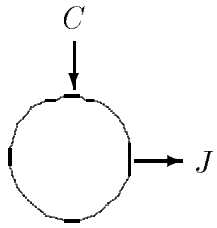


Figure 4.8: Systolic array for matrix inversion

processed, *init* is set to 0 and remains 0 till the end of the execution.

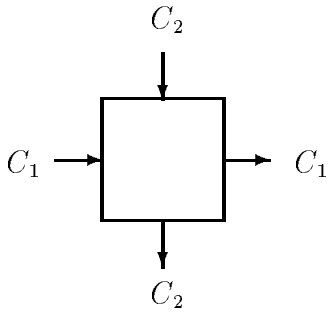
- Round processors.



The round processors compute the inverse of their first valid input, according to line (1) of the algorithm. Then they act simply as delay processors.

```
|[var C, J: real; init: Boolean; {initially init=1}
  upper? C
  ;if init=1
     → J := 1/C
       ;init := 0
   [] init=0
     → J := C
  fi
  ;right! J
]|
```

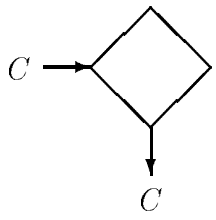
- Square processors.



The square processors first perform line (2) of the algorithm. Then they keep the result of this first computation and use it in performing loop (4) of the algorithm.

```
|[var C1, C2, J: real; init: Boolean; {initially init=1}
  left? C1; upper? C2
  ;if init=1
     → J := −C2 · C1
        ;init := 0
        ;right! C1
   [] init=0
     → C2 := C2 + C1 · J
        ;right! C1; lower! C2
  fi
]|
```

- Diamond processors.



The diamond processors perform line (3) of the algorithm. They operate similarly to the square processors, but do not compute J themselves. Instead, each diamond processor uses the value of J computed by the round processor in the same row.

```
|[var C, J: real; init: Boolean; {initially init=1}
 left? C
 ;if init=1
    → J := C
      ;init := 0
  [] init=0
    → C := C · J
      ;lower! C
 fi
]|
```

Let us consider in detail the operations performed by the n-th row of the algorithm. At *beat* n, P_{n0} computes $C_{n+1}[n,n]$, and acts thereafter as a delay processor. At *beat* $(n+1)$, P_{n1} computes $C_{n+1}[1,n]$, which it stores as J. Then P_{n1} begins to compute the expression of loop (4): at *beat* $(n+2)$ it computes $C_{n+1}[1,n+1]$:

$$C_{n+1}[1,n+1] := C_n[1,n+1] + C[1,n] \cdot C[n,n+1]$$

then at *beat* $(n+3)$ it modifies $C_{n+1}[1,n+2]$, then $C_{n+1}[1,n+3]$, $C_{n+1}[1,n+4]$, and so on. Processors $P_{n2}, P_{n3}, .., P_{n(N-1)}$ work like P_{n1}, beginning respectively at *beats* $(n+2)$, $(n+3)$,.., $(n+N-1)$.

At *beat* $(n+N)$, $C_{n+1}[n,n]$ reaches processor P_{Nn} and is stored in its variable J. From *beat* $(n+N+1)$ to the end of execution, P_{Nn} evaluates the

n-th row of C_{n+1} by executing line (3). Figure 4.9 shows the computations performed by the first row of the array for $N = 4$.

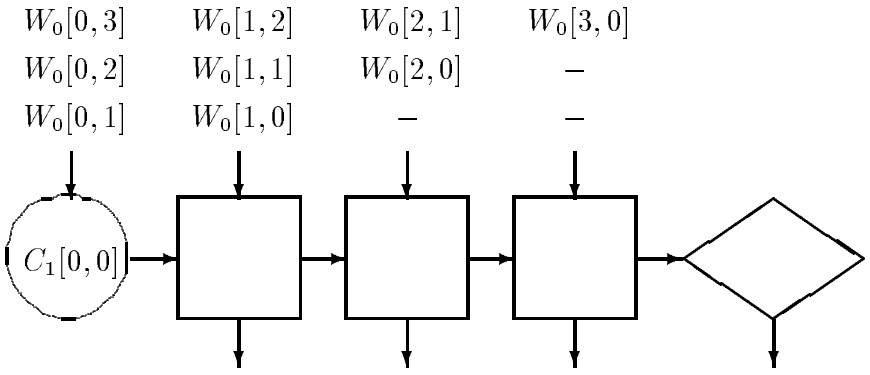
One may verify that the algorithm is executed in $(5 \cdot N - 2)$ beats, that is the running time is the same as the time of the transitive closure algorithm.

4.3 General algebraic path algorithms

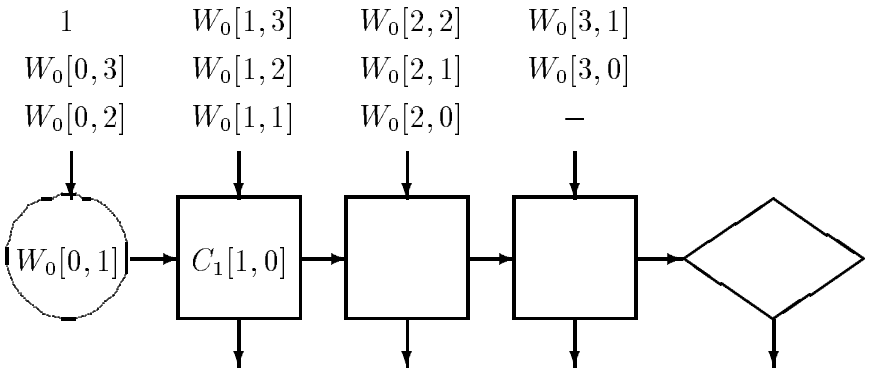
The first systolic algorithm for solving the algebraic path problem was presented in [Rote, 1985]. The algorithm was designed for the graph model of the algebraic path problem, which is less general than Lehmann's matrix model (see Sections 2.3 and 2.4). Also, Rote mentioned a possible way of computing the closure of a large matrix on a small systolic array, but did not investigate this problem. A year later Robert and Trystram showed that Lehmann's general algebraic path problem may be solved by a simple modification of the matrix inversion algorithm described in the previous section [Robert and Trystram, 1986a], [Robert and Trystram, 1986b]. In the same year, Lewis and Kung independently derived a similar algorithm using a square systolic array of $N \times N$ processors [Lewis and S. Kung, 1986].

[Navarro *et al.*, 1986] suggested a method of solving problems close to the algebraic path problem on a linear systolic array. A development of Rote's idea of solving an arbitrary-size algebraic path problem on a fixed-size systolic array was presented in [Nunez and Valero, 1988]. The technique used in this solution is similar to the technique presented in [D. Heller, 1984] and [Moldovan and Fortes, 1986]. (However, it seems that Nunez and Valero developed this technique independently.)

In 1989, Benaini, Robert, and Tourancheau have designed a new, toroidal architecture for a systolic solution of the algebraic path problem [Benaini *et al.*, 1989], [Benaini *et al.*, 1990]. The number of processors in their new array is half that of previously designed arrays, while the speed of computation is almost the same. A year later, Benaini and Robert published an even more efficient algorithm with running time $(5 \cdot N - 2)$ beats, which is the fastest running time ever suggested before, and with only $(N^2/3)$ processors, which is one and a half time less than in [Benaini *et al.*, 1989]. They also have proved that this algorithm is an optimal space-time algorithm for a Gauss-Jordan solution of the algebraic path problem [Benaini and Robert, 1990a], [Benaini and Robert, 1990b].

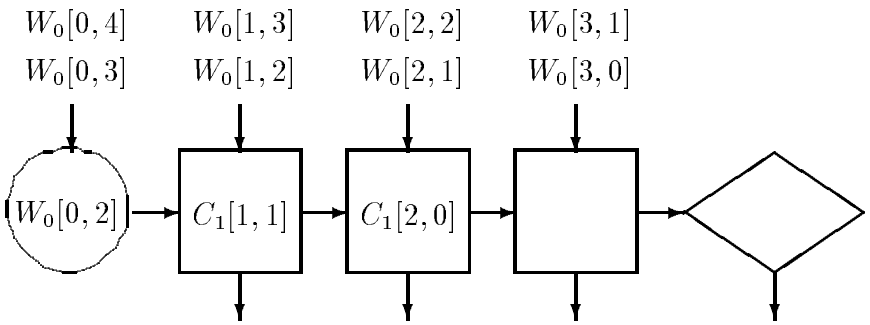


Beat 0

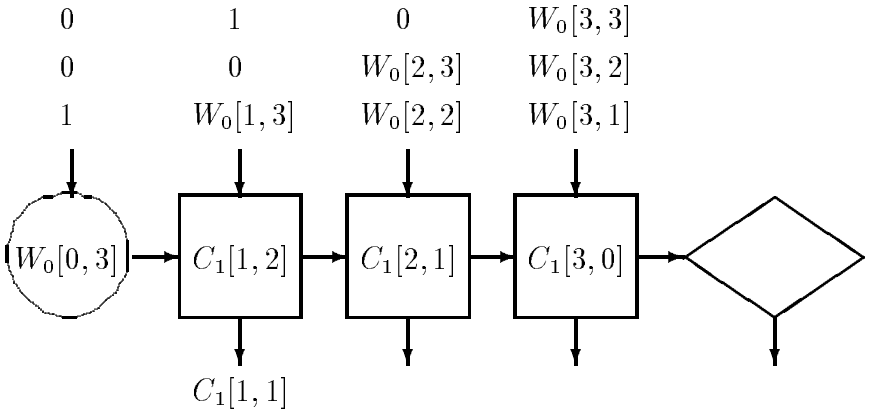


Beat 1

Figure 4.9: Operations of the first row of the array

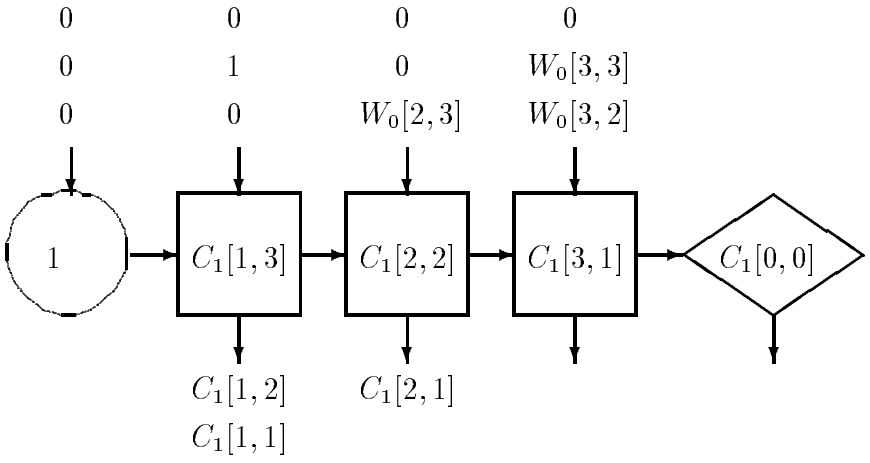


Beat 2

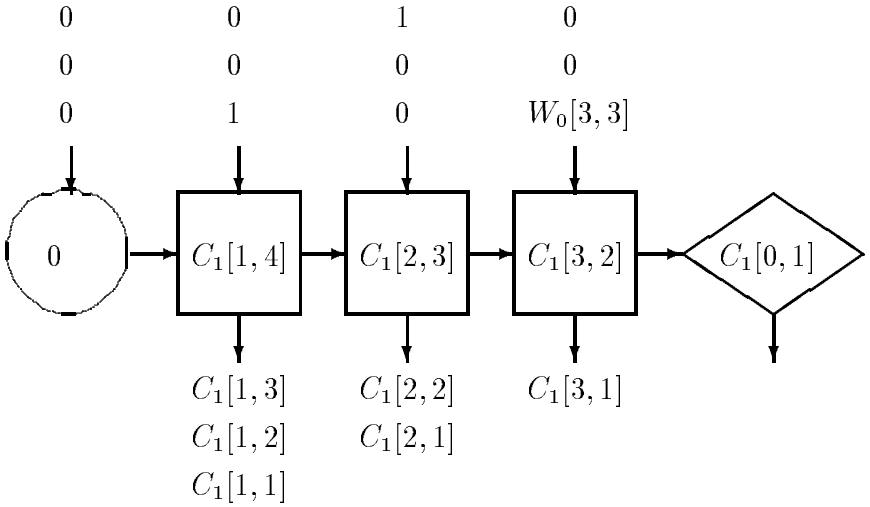


Beat 3

Figure 4.9 (continued): Operations of the first row of the array



Beat 4



Beat 5

Figure 4.9 (continued): Operations of the first row of the array

In this section we present three systolic algebraic path algorithms. We begin by describing the first algebraic path algorithm, designed by Rote. Then we show how to modify the Gauss-Jordan algorithm, discussed in the previous section, to obtain the fastest known algebraic path algorithm. Finally, we show how an arbitrary-size algebraic path problem may be solved on a fixed-size systolic array.

4.3.1 Rote's systolic algorithm

Rote's algorithm is the first systolic algorithm for solving the algebraic path problem [Rote, 1985]. The algorithm uses the Gauss-Jordan elimination technique, modified for systolic computations on a hexagonal array. Rote's solution uses the graph approach to the algebraic path problem, which is slightly less general than Lehmann's matrix approach. Recall that in the graph approach we assume that

for all $a \in S$,

(g)	$a \oplus a = a$	(idempotence of addition)
(h)	$a \otimes \bar{0} = \bar{0} \otimes a = \bar{0}$	(absorption rule)

Rote's method is quite complicated and hard to understand, and, worse, I did not find any publication with a rigorous mathematical proof of its correctness. According to Rote, the proof is presented in his original research report (Rote, 1985), but I was not able to obtain this report. Table 4.2 presents Rote's sequential version of the Gauss-Jordan algorithm, which was used for designing the systolic algorithm. A reader may verify that this algorithm performs the same computation as Lehmann's modification of the Gauss-Jordan algorithm (see [Rote, 1985] for a derivation of this algorithm). Further, by a careful comparison of Rote's sequential algorithm with Rote's systolic array, a careful and patient reader may convince himself that the array indeed correctly computes the transitive closure, but it is a *very* tedious task.

Figure 4.10 shows Rote's systolic array, for the case $N = 3$. The array consists of $(N + 1) \times (N + 1)$ hexagonal processors of seven different kinds, denoted $A, B, .., G$. The processors either perform simple operations corresponding to one of the four types of assignment of Rote's sequential algorithm (Table 4.2), or just pass their input unaltered.

```
for i := 0 to N − 1 do
   for j := 0 to N − 1 do
   begin
      for n := 0 to (i min j)-1 do
         W_{n+1}[i,j] := W_n[i,j] ⊕ W_{n+1}[i,n] ⊗ W_n[n,j];
      if i = j
         then W_{i+1}[i,i] := (W_i[i,i])*;
      if i > j
         then W_{j+1}[i,j] := (W_j[i,j]) ⊗ W_{j+1}[j,j]
   end;

for i := 0 to N − 1 do
   for j := 0 to N − 1 do
   begin
      if i < j
         then W_{i+1}[i,j] := (W_{i+1}[i,i]) ⊗ W_i[i,j];
      for n := (i min j)+1 to (i max j)-1 do
         W_{n+1}[i,j] := W_n[i,j] ⊕ W_{n+1}[i,n] ⊗ W_n[n,j];
      if i < j
         then W_{j+1}[i,j] := (W_j[i,j]) ⊗ W_{j+1}[j,j]
   end;

for i := 0 to N − 1 do
   for j := 0 to N − 1 do
   begin
      if i > j
         then W_{i+1}[i,j] := (W_{i+1}[i,i]) ⊗ W_i[i,j];
      for n := (i max j)+1 to N − 1 do
         W_{n+1}[i,j] := W_n[i,j] ⊕ W_{n+1}[i,n] ⊗ W_n[n,j]
   end
```

Table 4.2: Rote's sequential algorithm for the algebraic path problem

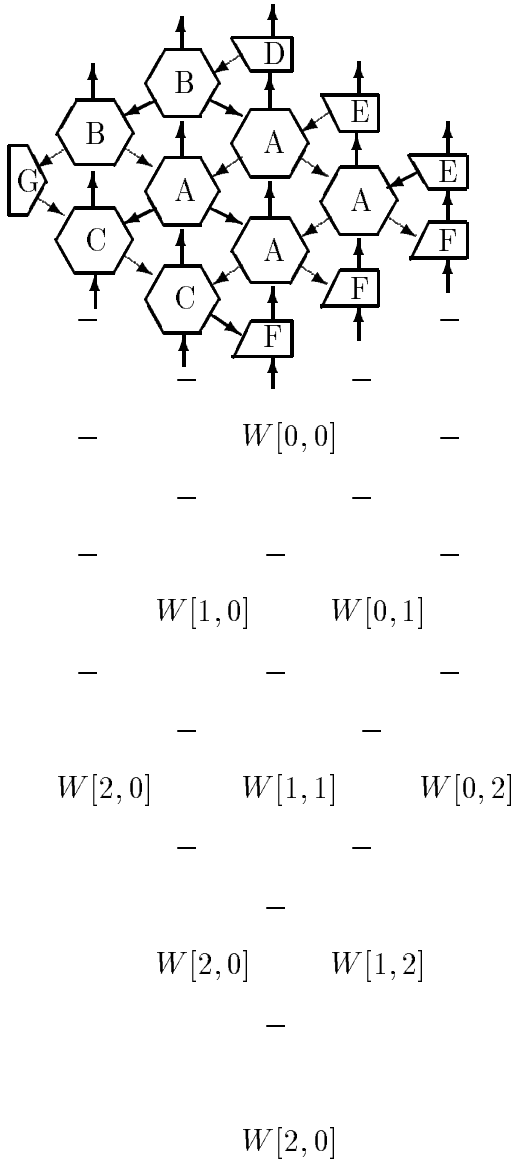
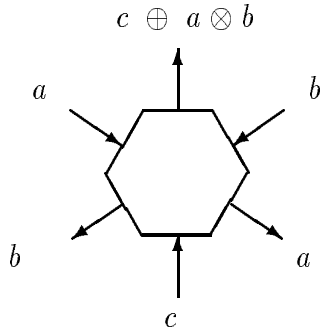


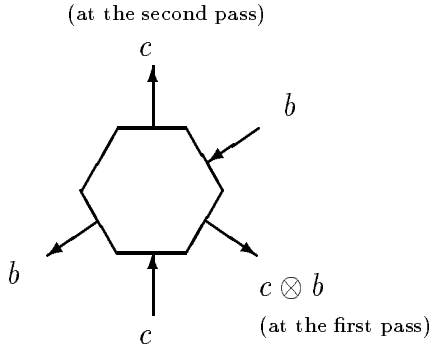
Figure 4.10: Rote's systolic array for the algebraic path problem

- Type A.



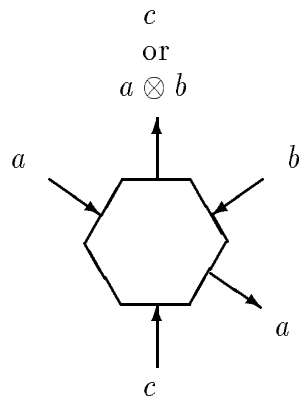
The type A processor performs the inner product calculation $c \oplus a \otimes b$, where a, b, and c are the input values of the processor.

- Type B.



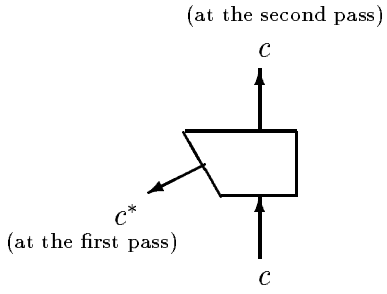
During the first pass of element c through the array, the type B processor calculates the function $c \otimes b$, where c and b are the input values of the processor. During the second pass, the processor just passes c unaltered upward, out of the array.

- Type C.



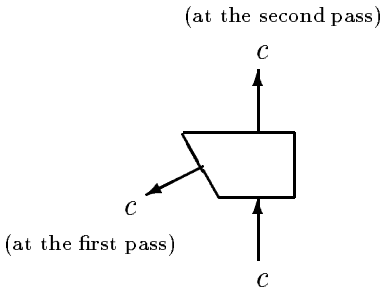
The type C processor either calculates just the identity function, if it receives input from below (that is the input from outside of the array), or it calculates the function $a \otimes b$, if it receives input from the diagonals. The whole algorithm works in such a way that at each beat the processor receives only one of these two inputs.

- Type D.



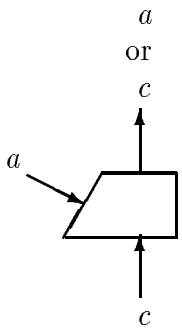
During the first pass of element c through the array, the type D processor calculates the closure operation c^*, where c is the input value. During the second pass, the processor passes c unaltered upward, out of the array.

- Type E.



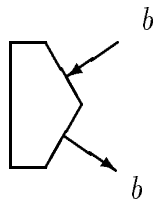
The type E processor does not calculate any expression. It just works as a delay processor, passing on data unaltered. During the first pass, the processor passes its input down and to the left. During the second pass, it passes its input upward.

- Type F.



The type F processor is also a delay processor. It passes on the value received either from below (that is the input from outside of the array) or from the diagonal. The whole algorithm works in such a way that at each beat the processor receives at most one of these two inputs.

- Type G.



  Finally, the type G processor also acts as a delay processor, passing on its input unaltered.

The initial array $W = W_0$ is fed into the systolic array in the order shown in Figure 4.10, and each element $W[i,j]$ passes through the array, changing its value according to the algorithm given in Table 4.2. The paths of three typical elements of the array W are shown on Figure 4.11. The smaller dash denotes the parts of the paths where elements are updated, and the larger dash denotes the parts of the paths where elements are passed unchanged. The path of every element $W[i,j]$ is as follows: first the element enters the array from below and travels upward until it hits the top of the array. The element then reflects either down and to the right, if it hits the top left edge of the array, or down and to the left, if it hits the top right edge of the array. The element continues to travel through the array until it hits a bottom edge, and then reflects and travels straight up again. Upon reaching a top edge for the second time, the element again reflects and travels down and to the left or down and to the right, until it reaches a bottom edge the second time. Then the element reflects from the bottom and travels upward once more.This time when the element reaches the top of the array, it passes straight through and becomes a part of the output, $W_N[i,j]$.

Observe that an element $W[i,j]$ changes its value only when it flows upward. The three upward phases of passing through the array correspond to the three main loops of the sequential algorithm in Table 4.2. The following considerations, in connection with Figure 4.12, are intended to explain intuitively why the systolic array implements correctly the sequential algorithm.

The initial position of the matrix elements is related to their position at some later time by the condition that they have travelled the same distance, because they all travel at the same speed. Thus, it is possible to determine

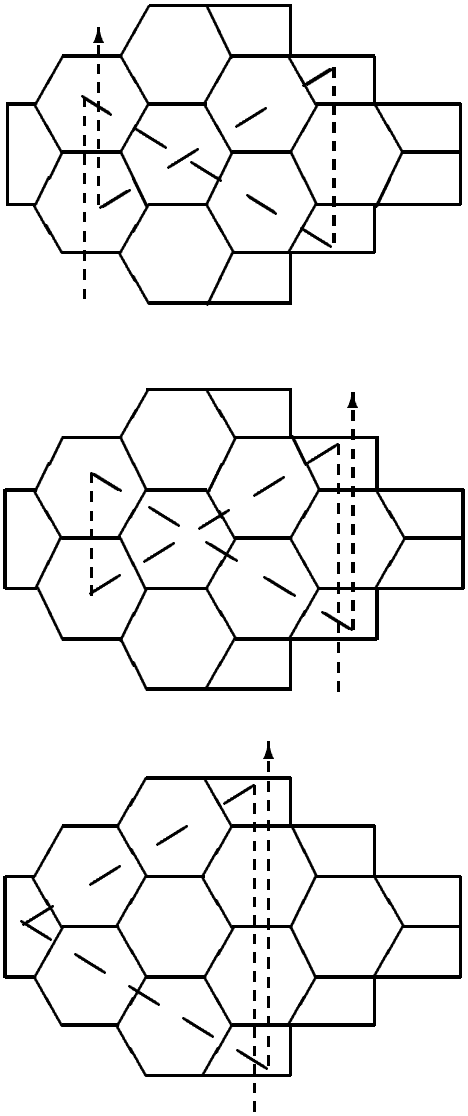


Figure 4.11: Paths of elements in Rote's systolic array

the original position in the matrix of the elements that meet in some processor at some beat.

Figure 4.12 shows some element $W[i,j]$ and four pairs of elements that meet $W[i,j]$. The correctness may be verified by counting the distance along the paths from the initial position of the elements to the processor where they meet. By generalization, one can see that the elements that meet $W[i,j]$ are pairs of $W[i,n]$ and $W[n,j]$. In the left side of the figure, n ranges from 0 to $(i \,\mathbf{min}\, j)$, as long as $W[i,j]$ is in the first upward phase of its path. In the right part of the figure, $W[i,j]$ is in the second upward phase, and n ranges from $(i \,\mathbf{min}\, j)$ to $(i \,\mathbf{max}\, j)$. Similarly, it might be shown that at the third upward stage, $W[i,j]$ meets pairs of elements with n varying from $(i \,\mathbf{max}\, j)$ to $(N-1)$.

From looking at Figure 4.11, it is clear that the path followed by an element is exactly $(3 \cdot N)$ steps longer than if the element travelled straight through the array, and that each element enters and leaves the array in the same vertical line. Therefore the matrix that comes out at the top is of the same shape as the input matrix, but with the delay of $(3 \cdot N)$ steps. The last element, $W[N-1, N-1]$, is fed into the array $(3 \cdot N - 3)$ beats later than the first element, $W[0,0]$, and it takes $(4 \cdot N + 1)$ beats for the last element to complete its pass trough the array. Thus, the total running time of the algorithm is $(7 \cdot N - 2)$ beats.

4.3.2 Fast systolic algebraic-path algorithm

In this subsection we show how to solve the algebraic path problem by executing Lehmann's modification of the Gauss-Jordan algorithm on the systolic array described in Subsection 4.2.3. The modification of a sequential Gauss-Jordan algorithm for computing the closure W_0^* of a matrix W_0 over an arbitrary closed semiring is shown in Table 4.3. The algorithm is obtained from the initial Gauss-Jordan algorithm by replacing addition, multiplication, and division with their generalized equivalents. The algorithm looks different from Lehmann's algorithm described in Subsection 3.2.2, but a careful comparison of these two algorithms shows that they compute exactly the same thing.

To compute the closure of a matrix W_0 on the array described in Subsection 4.2.3, we modify the algorithms performed by the processors in the array as follows.

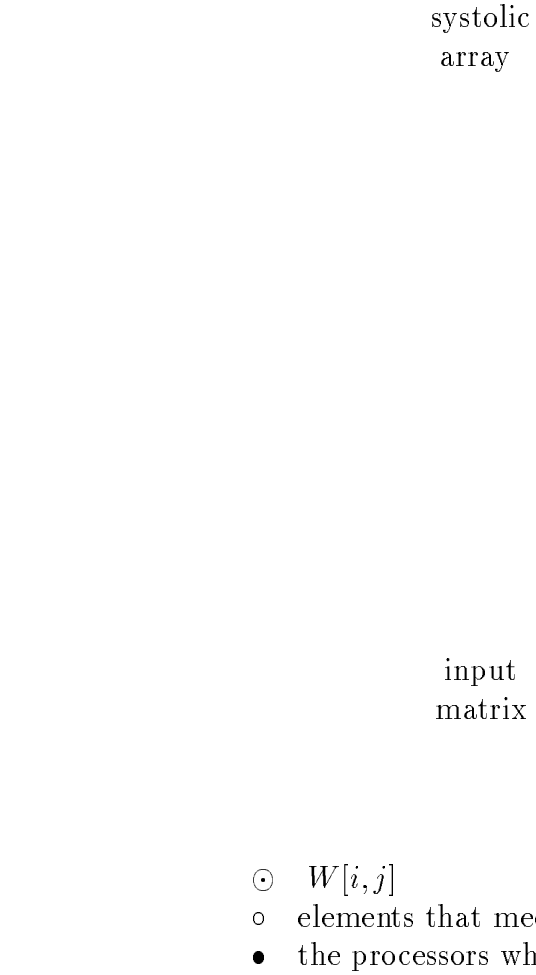


Figure 4.12: The paths of elements that meet in the same processor

for $n := 0$ **to** $N-1$ **do**
begin
 {Compute J_n, store $J_n[0..N-1,n]$ in $C_{n+1}[0..N-1,n]$}
  $C_{n+1}[n,n] := (C_{n+1}[n,n])^*$; (1)
  for $i := 0$ **to** $N-1$ **do**
    if $i \neq n$
      then $C_{n+1}[i,n] := C_n[i,n+1] \otimes C_{n+1}[n,n]$; (2)
 {Compute $C_{n+1}[0..N-1, n..2 \cdot N-1]$}
  for$j := n+1$ **to** $2 \cdot N-1$ **do**
  begin
    $C_{n+1}[n,j] := C_{n+1}[n,n] \otimes C_n[n,j]$; (3)
    for $i := 0$ **to** $N-1$ **do**
      if $i \neq n$ (4a)
        $C_{n+1}[i,n] := C_n[i,j] \ \oplus \ C_{n+1}[i,n] \otimes C_n[n,j]$ (4b)
  end
end

Table 4.3: A sequential Gauss-Jordan algorithm for algebraic path

- Round processors.

```
|[var C, J: Semiring-Values; init: Boolean; {initially init=1}
  upper? C
  ;if init=1
     → J := C*
        init := 0
   [] init=0
     → J := C
  fi
  ;right! J
]|
```

- Square processors.

```
|[var C1, C2, J: Semiring-Values; init: Boolean; {initially init=1}
 left? C1; upper? C2
 ;if init=1
    → J := C2 ⊗ C1
      init := 0
      ;right! C1
  [] init=0
    → C2 := C2 ⊕ C1 ⊗ J
      ;right! C1; lower! C2
 fi
]|
```

- Diamond processors.

```
|[var C, J: Semiring-Values; init: Boolean; {initially init=1}
 left? C ;if init=1
    → J := C
      init := 0
  [] init=0
    → C := C ⊗ J
      ;lower! C
 fi
]|
```

The correctness of this algorithm immediately follows from the correctness of Lehmann's algebraic path algorithm, shown in Table 4.3, and the discussion in Subsection 4.2.3. The running time of the algorithm is $(5 \cdot N - 2)$ beats, that is the algorithm is almost one and a half times as fast as Rote's algorithm. At present, this is the fastest known systolic algorithm for solving the algebraic path problem. However, it is *not* space-time optimal. The algorithm described in [Benaini *et al.*, 1989] contains three times less processors than the algorithm described in this section, and computes the closure of a matrix in the same number of beats.

4.3.3 Solving the algebraic path problem on a fixed size systolic array

In this section we describe the method for solving an arbitrary-size algebraic path problem on a fixed-size systolic array presented in [Nunez and Valero, 1988]. We develop a sequential algorithm, which is allowed to use a fixed size, $M \times M$, systolic array for executing "elementary" operations on matrices of size $M \times M$. Recall that usual sequential algorithms find the closure of a given $N \times N$ matrix W_0 by computing the sequence of matrices $W_0, W_1, .., W_N$, where $W_N = W_0^*$. We use a similar technique with a fixed-size systolic array, but we move along this sequence by steps of size M. That is we compute $W_0, W_M, W_{2 \cdot M}, .., W_N$. For simplicity we assume that N is a multiple of M. Let us divide the matrix W into four matrices, A, B, C, and D, as shown in Figure 4.13. Here A is an $M \times M$ diagonal matrix between the $(n \cdot M)$-th and $((n+1) \cdot M - 1)$-st rows and columns, that is

$$A = W[n \cdot M \ \ldots \ (n+1) \cdot M - 1, \ n \cdot M \ \ldots \ (n+1) \cdot M - 1]$$

B and C are respectively $M \times (N - M)$ and $(N - M) \times M$ matrices, and D is a $(N - M) \times (N - M)$ matrix. (Matrices B, C, and D consist of several blocks. To obtain the matrices, we put the blocks together.) The following formula shows how to compute $W_{(n+1) \cdot M}$ via $W_{n \cdot M}$, by computing each of its four parts. The formulas work only for the graph model of the algebraic path problem. Similar formulas for Lehmann's matrix model are more complicated, but they still may be directly derived from the definition of the closure of a matrix (see Section 2.4).

$$A_{(n+1) \cdot M} = (A_{n \cdot M})^* \quad (1)$$
$$B_{(n+1) \cdot M} = (A_{n \cdot M})^* \otimes B_{n \cdot M} \quad (2)$$
$$C_{(n+1) \cdot M} = C_{n \cdot M} \otimes (A_{n \cdot M})^* \quad (3)$$
$$D_{(n+1) \cdot M} = D_{(n+1) \cdot M} \ \oplus \ C_{n \cdot M} \otimes (A_{n \cdot M})^* \otimes B_{n \cdot M} \quad (4)$$

The proof of the correctness of these formulas is obtained by a straightforward use of the definition of the closure operation. (However, it is quite tedious.) Below we present a semi-intuitive justification of the formulas, using the weighted graph G_0 defined by the matrix W_0. We wish to show that the element $W_{n \cdot M}[i, j]$, for all $i, j \in [0..N-1]$ and for all $n \in [0..N/M]$, equals the length of the shortest path of the form $(i \rightarrow^{<n \cdot M} j)$ from i to j,

D	C	D
B	A	B
D	C	D

Figure 4.13: Dividing W into matrices A, B, C, and D

that is $W_{n\cdot M}[i,j]$ is the length of the shortest path from i to j all intermediate vertices of which are less than $(n \cdot M)$. As usual, we proceed by induction.

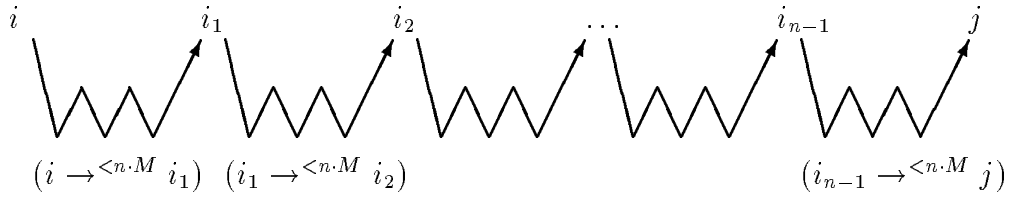
The claim trivially holds for W_0. So, we assume that the claim holds for $W_{n\cdot M}$ and wish to prove that it holds for $W_{(n+1)\cdot M}$. A path $(i \rightarrow^{<(n+1)\cdot M} j)$ in G_0 is either a path of the form $(i \rightarrow^{<n\cdot M} j)$, or it consists of a series of n paths

$$(i \rightarrow^{<n\cdot M} i_1),\ (i_1 \rightarrow^{<n\cdot M} i_2),\ldots,\ (i_{n-1} \rightarrow^{<n\cdot M} j)$$

whose endpoints $i_1, i_2, .., i_{n-1}$ are in the set

$$[n \cdot M\ \ldots\ (n+1)\cdot M - 1]$$

Pictorially, this may be shown as follows:



Formula (1) follows trivially from considering the case

$$i, j \in [n \cdot M\ \ldots\ (n+1)\cdot M - 1]$$

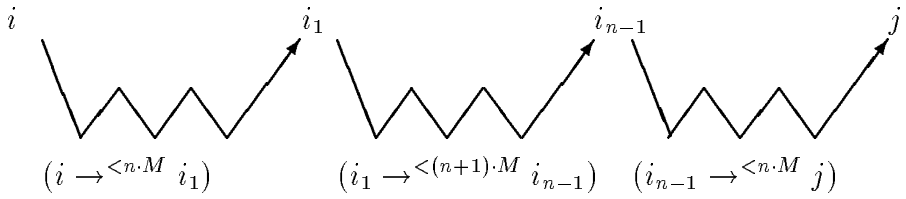
To prove Formula (4), we consider the case when

$$i, j \notin [n \cdot M\ \ldots\ (n+1)\cdot M - 1]$$

We divide this path into three parts:

$$(i \rightarrow^{<n\cdot M} i_1),\ (i_1 \rightarrow^{<(n+1)\cdot M} i_{n-1}),\ \text{and}\ (i_{n-1} \rightarrow^{<n\cdot M} j)$$

which pictorially looks as follows



By formula (1), the shortest possible length of the second part of the path equals $W_{(n+1)\cdot M}[i_1, i_{n-1}]$. Thus, the shortest path from i to j equals the minimum of the expression

$$W_{n\cdot M}[i, i_1] + W_{(n+1)\cdot M}[i_1, i_{n-1}] + W_{n\cdot M}[i_{n-1}, j]$$
$$\text{where } i_1, i_{n-1} \in [n \cdot M \ \ldots \ (n+1)\cdot M - 1]$$

Now we replace addition by a generalized multiplication $\otimes$, and minimum by a generalized addition $\oplus$, and get the formula

$$W_{(n+1)\cdot M} = \bigoplus_{i_1, i_n \in [n\cdot M \,\ldots\, (n+1)\cdot M - 1]} W_{n\cdot M}[i, i_1] \otimes W_{(n+1)\cdot M}[i_1, i_m] \otimes W_{n\cdot M}[i_m, j]$$

which is equivalent to (4). Thus, we have proved Formula (4). Formulas (2) and (3) are proved similarly.

To design a sequential algorithm that uses an $M \times M$ systolic array, we divide the matrix W into $(N/M)^2$ blocks, each of size $M \times M$, as shown in Figure 4.14. We denote a matrix consisting of blocks $i, i+1, .., j$ of the n-th row by $W(n, i..j)$, the matrix containing all rows except the i-th and all columns except the j-th by $W(\neg i, \neg j)$, and other possible cases similarly. Below we present an algorithm for computing the closure of a matrix W_0.

$W(0,0)$	$W(0,1)$	...	$W(0,N/M-1)$
$W(1,0)$	$W(1,1)$	...	$W(1,N/M-1)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$W(N/M-1,0)$	$W(N/M-1,1)$	...	...

Figure 4.14: Dividing W into $(N/M)^2$ blocks of equal size

$|[$**con** M, N: int $\{M, N > 0$ and M divides $N\}$;
$W[0..N-1, 0..N-1]$: **array of** Real;
$\{W$ is divided into blocks $W(i,j)$, $0 \leq i, j < N/M$,
such that for all i and j,
$W(i,j) = W[i \cdot M \ \ldots \ (i+1) \cdot M - 1, j \cdot M \ \ldots \ (j+1) \cdot M - 1]\}$
$\{W = W_0\}$
$n := 0$
;**do** $n < N/M$
$\rightarrow \{W = W_{n \cdot M}\}$
$W(n,n) := (W(n,n))^*$ $\{(1)\}$
$;W(n,\neg n) := W(n,n) \otimes W(n,\neg n)$ $\{(2)\}$
$;W(\neg n,\neg n) := W(\neg n,\neg n) \oplus W(n,\neg n)$
$\otimes W(n,0..N/M-1)$ $\{(3)\}$
$\{W = W_{(n+1) \cdot M}\}$
$;n := n + 1$
od
$\{W = W_0^*\}$
$]|$

The algorithm uses Formulas (1), (2), and (4). Similarly, we may write an algorithm based on Formulas (1), (3), and (4). As we already mentioned, the algorithm works only for the graph model of the algebraic path problem. To design an algorithm for the more general matrix approach, one has to derive

more complex formulas, similar to Formulas (1)–(4), based on Lehmann's definition of the matrix closure.

This algorithm still needs some adjastment, because we cannot multiply a matrix of the size $(N - M) \times M$ by a matrix of the size $M \times N$ on an $M \times M$ systolic array. Let us define two operations over matrices X, Y, and Z with sizes respectively $M \times M$, $M \times K$, and $M \times K$, where K is an arbitrary natural number:

$$f[X;Y] = X^* \otimes Y$$
$$g[X;Y;Z] = X \otimes Y \ \oplus \ Z$$

These operations may be readily implemented on an $M \times M$ array with running time $O(N)$ (see for example [Quinton, 1990]). We introduce some more notation: we denote by (X, I, Y) the matrix obtained by concatenating together (horizontally) matrices X, I, and Y, where I is the $M \times M$ identity matrix. Now we are ready to present a sequential algorithm for solving the algebraic path problem that uses an $M \times M$ systolic array. The algorithm is obtained from the previous one by several simple modifications.

```
|[con M, N: int {M, N > 0 and M divides N};
  W[0..N − 1, 0..N − 1]: array of Real;
  {W is divided into blocks W(i, j), 0 ≤ i, j < N/M,
     such that for all i and j,
     W(i, j) = W[i · M ... (i + 1) · M − 1, j · M ... (j + 1) · M − 1]}
  {W = W_0}
     n := 0
     ;do n < N/M
     → {W = W_{n·M}}
        W(n, 0..N/M − 1)
           := f[W(n, n) ; (W(n, 0..n − 1), I, W(n, n + 1..N/M − 1))]
        i := 0;
        ;do i < N/M
        → if i ≠ n → W(i, 0..N/M − 1)
                          := g[W(i, n) ; W(n, 0..N/M − 1) ; W(i, ¬n)]
          [] i = n → skip
          fi
          ;i := i + 1
        od
        {W = W_{(n+1)·M}}
        ;n := n + 1
     od
  {W = W_0^*}
]|
```

One may verify that the algorithm takes $O(N^3/M^2)$ time. A more precise evaluation of running time is presented in [Nunez and Valero, 1988]. Nunez and Valero described systolic implementations of the operations f and g with which the algorithm computes the closure of a matrix in $(N^3/M^2+3\times M-2)$ beats.

Chapter 5

Conclusion

In this report, we have described the algebraic path problem and have shown how this problem may be used to perform a lot of important operations on graphs and matrices. Also, we reviewed the history of the problem and hopefully helped the reader to grasp a picture of the historical development of many different seemingly unrelated problems into a single general theory. The history of the algebraic path problem is typical for many other general problems in computer science. The moral of the story is summarized in Moore's statement quoted in the introduction of this report. In other words,

> If you see similarities between several different problems, always look for a unified theory.

Another historical observation, also typical for computer science problems (and for problems in other sciences too) is that all solutions of the algebraic path problem have been obtained by modifying solutions of less general problems. This approach simplifies the task of researchers and often helps to develop a formal, well-structured theory by generalizing an old theory in small clear steps and using a lot of old, well-formalized and polished results in a new theory. However, this approach may have negative effects too, because it sometimes prevents scientists from creative thinking and from gaining a new unexpected insight for the general theory.

We have described several sequential and systolic algorithms for solving the algebraic path problem and four special cases of the general problem. We have seen that the systolic model of computations is convenient for developing algorithms for this problem, and that an increase in the number

of processors allows a proportional decrease in the running time, that is an $M \times M$ systolic array allows us to solve the problem in $O(N^3/M^2)$ running time. The boundary cases are a sequential algorithm with running time $O(N^3)$, and an $N \times N$ systolic array with running time $O(N)$. The running time $O(N^3/M^2)$ is a big success, and shows that systolic arrays well fit matrix computations. For many other problems the speed increases slower than the number of processors in the systolic array. For example, the best sorting algorithm works in $O(\sqrt{N})$ time on an N-processor systolic array [Ullman, 1984] versus an $O(N \cdot \log N)$ sequential algorithm, that is the space-time complexity increases from $O(N \cdot \log N)$ to $O(N \cdot \sqrt{N})$.

Ironically, the lower bound for the space-time complexity of systolic algebraic path algorithms has already been found and optimal algorithms have been designed, while the optimal running time of sequential solutions is still unknown.

A lot of work has been done on using less restrictive models of parallel computations to perform matrix computations [Stone, 1973], [D. Heller, 1978] and to solve graph problems [Ecksten and Alton, 1977], [Reghbati, 1978], [Quinn and Yoo, 1984], [Savage and Ja'Ja', 1981], especially on matrix multiplication [Dekel *et al.*, 1981] and inversion [Pease, 1967], [Csanky, 1975], shortest path [Deo *et al.*, 1980], [Chen, 1982], [Frieze and Rudolph, 1984], [Lakhani, 1984], [Paige and Kruskal, 1985], [Jenq and Sahni, 1987], and transitive closure [Hirschberg, 1976], [Greenberg, 1982]. Matrix algorithms on non-systolic parallel machines are *not* more efficient than corresponding systolic algorithms (except for sparse matrix computations, which are more efficient on some parallel machines). Some non-systolic parallel algorithms for graph problems are more efficient than their systolic equivalents. However, if we take into account inefficiency of hardware implementation of less restrictive parallel models, systolic algorithms for both graphs and matrices are probably fastest.

Bibliography

[Aho *et al.*, 1972] A. V. Aho, M. R. Garey, and J. D. Ullman. The transitive reduction of a directed graph. *SIAM Journal of Computing*, 2, 1972, pages 131-137.

[Aho *et al.*, 1974] A. V. Aho, J. E. Hopcroft, J. D. Ullman. *The design and analysis of computer algorithms*, Addison-Wesley, Reading, MA, 1974.

[Aho *et al.*, 1983] A. V. Aho, J. E. Hopcroft, J. D. Ullman. *Data structures and algorithms*, Addison-Wisley, Reading, MA, 1983.

[Apt and Olderog, 1991] K. R. Apt and E. R. Olderog. *Verification of sequential and concurrent programs.* Springer-Verlag, NY, 1991.

[Arlazarov *et al.*, 1970] V. L. Arlazarov, E. A. Dinic, M. A. Kronod, and I. A. Faradzev. On economical construction of the transitive closure of an oriented graph. *Doklad Akademii Nauk SSSR*, 11, 1970, pages 1209–1210.

[Atallah and Kosaraju, 1982] M. J. Atallah and S. R. Kosaraju. Graph problems on a mesh-connected processor array. *Proceedings of the Fourteenth Annual ACM Symposium on the Theory of Computing*, 1982, pages 345–353.

[Backhouse and Carre, 1975] R. C. Backhouse and B. A. Carre. Regular algebra applied to path-finding problems. *Journal of the Institute of Mathematics and Its Applications*, 7, 1975, pages 273–294.

[Bellman, 1958] R. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16, 1958, pages 87–90.

[Bellman and Kalaba, 1960] R. Bellman and R. Kalaba. On kth best policies. *SIAM Journal of Computing*, 8, 1960, pages 582–588.

[Benaini *et al.*, 1989] A. Benaini, Y. Robert, and B. Tourancheau. A new systolic architecture for the algebraic path problem. *Systolic array processors*, eds.: J. McCanny *et al.*, Prentice Hall, Englewood Cliffs, NJ, 1989, pages 73–82.

[Benaini *et al.*, 1990] A. Benaini, P. Quinton, Y. Robert, Y. Saouter, and B. Tourancheau. A new systolic architecture for the algebraic path problem. *Science of Computer Programming*, 15, 1990, pages 135–158.

[Benaini and Robert, 1990a] A. Benaini and Y. Robert. Space-time minimal systolic arrays for Gaussian elimination and the algebraic path problem. *International Conference on Application Specific Array Processors*, 1990, pages 746–757.

[Benaini and Robert, 1990b] A. Benaini and Y. Robert. Space-time minimal systolic arrays for Gaussian elimination and the algebraic path problem. *Parallel Computing*, 15, 1990, pages 211–225.

[Bentley and Cooke, 1965] D. L. Bentley and K. L. Cooke. Convergence of successive approximations in the shortest rote problem. *Journal of Mathematical Analysis and Applications*, 10, 1965, pages 269–274.

[Bertolazzi *et al.*, 1988] P. Bertolazzi, C. Guerra, and S. Salza. A systematic approach to the design of modular systolic arrays. *IEEE International Conference on Systolic Arrays*, 1988, pages 453–472.

[Bloniarz *et al.*, 1976] P. A. Bloniarz, M. J. Fischer, and A. R. Meyer. A note on the average time to compute transitive closure. *Automata, Languages and Programming*, eds.: Michaelson and Milner, Edinburg University Press, Edinburg, 1976, pages 425–434.

[Bulirsch and Stoer, 1980] R. Bulirsch and J. Stoer. *Introduction to numerical analysis.* Springer-Verlag, NY, 1980.

[Carre, 1971] B. A. Carre. An algebra for network routing problems. *Journal of the Institute of Mathematics and Its Applications*, 7, 1971, pages 273–294.

[Carre, 1979] B. A. Carre. *Graphs and networks.* The Clarendon Press, Oxford University Press, 1979.

[Chen, 1982] C. C. Chen. A distributed algorithm for shortest paths. *IEEE Transactions on Computers*, C-31, 1982, pages 892–899.

[Clarke *et al.*, 1963] S. Clarce, A. Krikorian, and J. Rausen. Computing the N best loopless paths in a network. *SIAM Journal of Computing*, 11, 1963, pages 83–89.

[Conway, 1971] J. H. Conway. *Regular algebra and finite machines*, Chapman and Hall, London, 1971.

[Cooke and Halsey, 1966] K. L. Cooke and E. Halsey. The shortest node through a network with time-dependent internodal transit time. *Journal of Mathematical Analysis and Applications*, 14, 1966, pages 493–498.

[Cormen *et al.*, 1990] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest. *Introduction to algorithms.* MIT Press, 1990.

[Csanky, 1975] L. Csanky. Fast parallel matrix inversion algorithms. *Proceedings of the 16th Annual IEEE Symposium on Foundations of Computer Science*, 1975, pages 11–12.

[Culik and Fris, 1984] K. Culik and I. Fris. *Topological transformations as a tool in the design of systolic networks.* Dept. of Computer Science, University of Waterloo, Research Report CS-84-11, 1984.

[Dantzig, 1957] G. B. Dantzig. Discrete-variable extremum problems. *The Journal of the Operations Research Society of America*, 5, 1957, pages 266–276.

[Dantzig, 1960] G. B. Dantzig. On the shortest route through a network. *Management Science*, 6, 1960, pages 187–190.

[Dantzig, 1966] G. B. Dantzig. All shortest routes in a graph. *Proceedings of the International Symposium on Theory of Graphs*, 1966, eds.: Gordon and Breach, NY, 1967, pages 91–92.

[Dantzig *et al.*, 1954] G. B. Dantzig, D. R. Fulkerson, and S. Johnson. Solution of a large scale travelling salesman problem. *The Journal of the Operations Research Society of America*, 2, 1954, pages 393–410.

[Dantzig *et al.*, 1966] G. B. Dantzig, W. O. Blattner, and M. R. Rao. All shortest routes from a fixed origin in a graph. *Proceedings of the International Symposium on Theory of Graphs*, 1966, eds.: Gordon and Breach, NY, 1967, pages 85–90.

[Dekel *et al.*, 1981] E. Dekel, D. Nassimi, and S. Sahni. Parallel matrix and graph algorithms. *SIAM Journal of Computing*, 1981, pages 657–675.

[Denardo and Fox] E. V. Denardo and B. L. Fox. Shortest-route methods: reaching, pruning, and buckets. *The Journal of the Operations Research Society of America*, 27, 1979, pages 161–186.

[Deo *et al.*, 1980] N. Deo, C. Y. Pang, and R. E. Lord. Two parallel algorithms for shortest path problems. *Proceedings of the International Conference on Parallel Processing*, 1980, pages 244–253.

[Dey and Srimani, 1989] S. Dey and P. K. Srimani. Fast parallel algorithm for all-pairs shortest path problem and its VLSI implementation. *IEE Proceedings*, Part E: Computers and Digital Techniques, 1989, 136(2), page 85.

[Dial *et al.*, 1979] R. B. Dial, F. Glover, D. Karney, and D. Kligman. A computational analysis of alternative algorithms and labelling techniques for finding shortest path trees. *Networks*, 9, 1979, pages 215–248.

[Dijkstra, 1959] E. W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1, 1959, page 269–271.

[Dijkstra, 1976] E. W. Dijkstra. *A discipline of programming.* Prentice-Hall, Englewood Cliffs, NJ, 1976.

[Dreyfus, 1969] S. E. Dreyfus. An appraisal of some shortest-path algorithms. *The Journal of the Operations Research Society of America*, 17, 1969, pages 266–276.

[Ebergen and Hoogerwoord, 1990] J. C. Ebergen and R. R. Hoogerwoord. A derivation of a serial-parallel multiplier. *Science of Computer Programming*, 15, 1990, pages 201–215.

[Ecksten and Alton, 1977] D. M. Ecksten and D. A. Alton. Parallel graph processing using depth-first search. *Proceedings of the Conference on Theoretical Computer Science*, 1977, pages 21–29.

[Enderton, 1977] H. B. Enderton. *Elements of set theory.* Academic Press, California, 1977.

[Fischer and Meyer, 1971] M. J. Fischer and A. R. Meyer. Boolean matrix multiplication and transitive closure. *Conference Record, Twelfth Annual Symposium on Switching and Automata Theory*, East Lansing, Mich., 1971, pages 129–131.

[Floyd, 1962] R. W. Floyd. Algorithm 97: Shortest path. *Communications ACM*, 5, 1962, page 345.

[Flynn, 1966] M. J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54, 1966, pages 1901–1909.

[Flynn, 1972] M. J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21, 1972, pages 948–960.

[Ford and Fulkerson, 1956] L. R. Ford, Jr. and D. R. Fulkerson. Maximal flow through a network. *Canadian Journal of Mathematics*, 8, 1956, pages 399–404.

[Ford and Fulkerson, 1958] L. R. Ford, Jr. and D. R. Fulkerson. Constructing maximal dynamic flow from static flows. *The Journal of the Operations Research Society of America*, 6, 1958, pages 419–433.

[Ford and Fulkerson, 1962] L. R. Ford, Jr. and D. R. Fulkerson. *Flows in networks.* Princeton University Press, Princeton, NJ, 1962.

[Forsythe and Moller, 1967] G. E. Forsythe and C. B. Moller. *Computer solution of linear algebraic systems.* Prentice Hall, Englewood Cliffs, NJ, 1967.

[Fortes *et al.*, 1985] J. A. B. Fortes, K. S. Fu, and B. W. Wah. Systematic approaches to the design of algorithmic specified systolic arrays. *IEEE ICASSP Proceedings*, 1985, pages 300–303.

[Fortune and Wylie, 1978] S. Fortune and J. Wyle. Parallelism in random access machines. *Proceedings of the Tenth Annual ACM Symposium on Theory of Computing*, 1978, pages 114–118.

[Fredman, 1975] M. L. Fredman. On the decision tree complexity of the shortest path problems. *Conference Records of the Sixteenth Annual IEEE Symposium on Foundations of Computer Science*, 1975, pages 98–99.

[Fredman, 1976] M. L. Fredman. New bounds on the complexity of the shortest path problem. *SIAM Journal of Computing*, 5(1), 1976, pages 83–89.

[Frieze and Rudolph, 1984] A. Frieze and L. Rudolph. A parallel algorithm for all pairs shortest paths in a random graph. *Proceedings of the 22nd Annual Allerton Conference on Communication, Control, and Computing*, 1984, pages 663–670.

[Furman, 1970] M. E. Furman. Application of a method of fast multiplication of matrices in the problem of finding the transitive closure of a graph. *Doklad Akademii Nauk SSSR*, 11, 1970, no. 5, page 1252.

[Gentleman and Kung, 1981] W. M. Gentleman and H. T. Kung. Matrix triangularization by systolic arrays. *Proceedings of the Society of Photo-Optical Instrumentation Engineers*, Real-Time Signal Processing, 298, 1981, pages 19–26.

[Gondran and Minoux, 1984] M. Gondran and M. Minoux. *Graphs and algorithms.* Wiley, NY, 1984.

[Goralcikowa and Koubek, 1979] A. Goralcikowa, V. Koubek. A reduct and closure for graphs. Mathematical Foundation of Computer Science 1979, Springer Lecture Notes in Computer Science 74, pages 301–307.

[Gosch, 1979] J. Gosch. Computer processes multiple instruction sets, multiple data streams. *Electronics*, Oct. 1979, pages 78–79.

[Greenberg and Fischer, 1982] A. G. Greenberg and M. J. Fischer. On computing weak transitive closure in $O(\log N)$ expected random parallel time. *Proceedings of the International Conference on Parallel Processing*, 1982, pages 199–204.

[Gries, 1981] D. Gries. *The science of programming.* Springer-Verlag, NY, 1981.

[Gries *et al.*, 1989] D. Gries, A. J. Martin, J. L. A. van de Shepscheut, and J. T. Udding. An algorithm for transitive reduction of an acyclic graph. *Science of Computer Programming*, 12, 1989, pages 151–155.

[Guibas *et al.*, 1979] L. J. Guibas, H. T. Kung, and C. D. Thompson. Direct VLSI implementation of combinatorial algorithms. *Proceedings of Caltech Conference on VLSI: Architecture, Design and Fabrication*, 1979, pages 509–525.

[Harary, 1971] F. Harary. *Graph Theory.* Addison Wesley, Reading, MA, 1971.

[D. Heller, 1984] D. Heller. Partitioning big matrices for small systolic arrays. *VLSI and Modern Signal Processing*, eds.: S. Y. Kung, H. J. Whitehouse, and T. Kailath, Prentice Hall, NJ, 1984, pages 185–199.

[D. Heller, 1978] D. Heller. A survey on parallel algorithms in numerical linear algebra. *SIAM Review*, 20, 1978, pages 740–777.

[I. Heller, 1955] I. Heller. On the travelling salesman's problem. *Proceedings of the Second Symposium in Linear Programming*, 2, 1955, pages 643–665.

[Hirschberg, 1976] D. S. Hirschberg. Parallel algorithms for the transitive closure and the connected component problems. *Proceedings of the Eighth Annual ACM Symposium on Theory of Computing*, 1976, pages 55–57.

[Hoare and Jones, 1989] C. A. R. Hoare and C. B. Jones. *Essays in computer science*, Prentice Hall, 1989.

[Hoffman and Pavley, 1959] W. Hoffman and R. Pavley. A method for the solution of the Nth best path problem. *Journal ACM*, 6, 1959, pages 506–514.

[Hoffman and Winograd, 1972] A. J. Hoffman and S. Winograd. Finding all shortest distances in a directed network. *IBM Journal of Research and Development*, 16(4), 1972, pages 412–414.

[Hu, 1967] T. C. Hu. Revised matrix algorithms for shortest paths. *SIAM Journal on Applied Mathematics*, 15, 1967, pages 207–218.

[Hu, 1968] T. C. Hu. A decomposition algorithm for the shortest paths in a network. *The Journal of the Operations Research Society of America*, 16, 1968, pages 91–102.

[Hu and Torres, 1969] T. C. Hu and W. T. Torres. Shortcut in the decomposition algorithm for shortest paths in a network. *IBM Journal of Research and Development*, 13(4), 1969, pages 387–390.

[Ibarra *et al.*, 1986] O. H. Ibarra, S. M. Kim, and M. A. Palis. Designing systolic algorithms using sequential machines. *IEEE Transactions on Computers*, C-35(6), 1986, pages 531–542.

[Ibarra and Sohn, 1989] O. H. Ibarra and S. M. Sohn. On mapping systolic algorithms onto a hypercube. *Proceedings of the International Conference on Parallel Processing*, volume 1, 1989, pages 121–124.

[Jenq and Sahni, 1987] J. F. Jenq and S. Sahni. All pairs shortest paths on a hypercube multiprocessor. *Proceeding of the International Conference on Parallel Processing*, 1987, pages 713–716.

[D. Johnson, 1973] D. B. Johnson. A note on Dijkstra's shortest path algorithm. *Journal ACM*, 20(3), 1973, pages 385–388.

[D. Johnson, 1977] D. B. Johnson. Efficient algorithms for shortest paths in sparse networks. *Journal ACM*, 24(1), 1977, pages 1–13.

[E. Johnson, 1972] E. L. Johnson. On shortest path and sorting. *Proceedings of the ACM Twenty-Fifth Annual Conference*, 1972, pages 510–517.

[Kalaba, 1958] R. Kalaba. *Proceedings of the Tenth Symposium in Applied Mathematics*, 1958, pages 261–280.

[Kaldewaij, 1990] A. Kaldewaij. *Programming: the derivation of algorithms.* Prentice Hall, Englewood Cliffs, NJ, 1990.

[Kaldewaij and Rem, 1990] A. Kaldewaij and M. Rem. The derivation of systolic computations. *Science of Computer Programming*, 14, 1990, pages 229–242.

[Kaldewaij and Udding, 1992] A. Kaldewaij and J. T. Udding. Rank order filters and priority queues. In preparation.

[Kam and Ullman, 1976] J. B. Kam and J. D. Ullman. Global data flow analysis and iterative algorithms. *Journal ACM*, 23(1), 1976, pages 158–171.

[Kautz and Levitt, 1972] W. H. Kautz and K. N. Levitt. Cellular arrays for the solution of graph problems. *Communications ACM*, 15(9), 1972, pages 789–801.

[Kirkpatrick, 1974] D. G. Kirkpatrick. Determining graph properties from matrix representations. *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, 1974, pages 84–90.

[Kleene, 1956] S. C. Kleene. Representation of events in nerve nets and finite automata. *Annals Of Mathematics Studies 34, Automata Studies*, eds: J. McCarthy and C. E. Shannon, 1956, pages 3–41.

[Knuth, 1968] D. E. Knuth. *The art of computer programming. Volume 1: Fundamental algorithms.* Addison-Wesley, Reading, Mass., 1968, pages 258–265.

[Knuth, 1969] D. E. Knuth. *The art of computer programming. Volume 2: Seminumerical Algorithms.* Addison-Wesley, Reading, Mass., 1969, pages 258–265.

[Knuth, 1973] D. E. Knuth. *The art of computer programming. Volume 3: Sorting and Searching.* Addison-Wesley, Reading, Mass., 1973, pages 258–265.

[Kramer and Leeuwen, 1983] M. R. Kramer and J. van Leeuwen. Systolic computation and VLSI. *Foundation of Computer Science IV, Part 1, Algorithms and Complexity*, eds.: J. W. DeBakker and J. van Leeuwen, 1983, pages 75–103.

[Kuck, 1977] D. J. Kuck. A survey of parallel machine organization and programming. *ACM Computing Surveys*, volume 9, 1977, pages 29–59.

[Kumar and Tsai, 1988] V. K. P. Kumar and Y. C. Tsai. Mapping two-dimensional systolic arrays to one-dimensional arrays and applications.

Proceedings of the International Conference on Parallel Processing, 1988, pages 39–46.

[Kunde *et al.*, 1986] M. Kunde, H. W. Lang, M. Schimmler, H. Schmeck, and H. Schroder. The instruction systolic array and its relation to other models of parallel computers. *Proceedings of the International Conference on Parallel Computing*, 1986, pages 491–498.

[H. Kung, 1979] H. T. Kung. Let's design algorithms for VLSI systems. *Proceedings of Caltech Conference on VLSI*, 1979, pages 65–90.

[H. Kung, 1980] H. T. Kung. The structures of parallel algorithms. *Advances in Computers*, 19, 1980, pages 65–103.

[H. Kung, 1982] H. T. Kung. Why systolic architecture. *IEEE Computer Magazine*, 15, 1982, pages 37–46.

[H. Kung and Leiserson, 1978a] H. T. Kung and C. E. Leiserson. *Systolic arrays for VLSI.* Carnegie-Mellon University, 1978. Tech. Report CMU-CS-79-103.

[H. Kung and Leiserson, 1978b] H. T. Kung and C. E. Leiserson. Systolic arrays (for VLSI). *Sparse Matrix Proceeding (Symposium on Sparse Matrix Computation)*, 1978, pages 256–282.

[H. Kung and Leiserson, 1978c] H. T. Kung and C. E. Leiserson. Algorithms for VLSI processor arrays. *Sparse Matrix Proceeding (Symposium on Sparse Matrix Computation)*, 1978.

[S. Kung, 1984] S. Y. Kung. On supercomputing with systolic/wavefront array processors. *Proceedings of the IEEE*, 72, 1984, pages 867–884.

[S. Kung, 1987] S. Y. Kung. VLSI array processors. *The First International Workshop on Systolic Arrays*, 1986. In *Systolic Arrays*, eds.: W. Moore, A. McCabe, and R. Urquhart, Adam Hilger, Bristol and Boston, 1987, pages 7–24.

[S. Kung, 1988] S. Y. Kung. *VLSI array processors.* Prentice Hall, Englewood Cliffs, NJ, 1988.

[S. Kung *et al.*, 1984] S. Y. Kung, S. C. Lo, and J. Annevelink. Temporal localization and systolization of signal flow graph (SFG) computing networks. *Proceedings of the Society of Photo-Optical Instrument Engineers*, Real-Time Signal Processing, VII, 1984.

[S. Kung *et al.*, 1986] S. Y. Kung, P. S. Lewis, and S. C. Lo. On optimally mapping algorithms to systolic arrays with application to the transitive closure problem. *Proceeding of IEEE International Symposium on Circuits and Systems*, 1986, pages 1316–1322.

[S. Kung *et al.*, 1987] S. Y. Kung, P. S. Lewis, and S. N. Jean. Canonic mapping from algorithms to arrays — a graph based methodology. *Proceedings of the Twentieth Hawaii International Conference on System Sciences*, 1987, volume 1, pages 124–133.

[S. Kung *et al.*, 1987] S. Y. Kung, P. S. Lewis, and S. C. Lo. Optimal systolic design for the transitive closure and the shortest path problem. *IEEE Transactions on Computers*, C-36, 1987, pages 603–614.

[S. Kung and Lo, 1985] S. Y. Kung and S. C. Lo. A spiral systolic architecture/algorithm for transitive closure problems. *IEEE International Conference on Computer Design*, 1985, pages 622–626.

[Lakhani, 1984] G. D. Lakhani. An improved distribution algorithm for shortest path problem. *IEEE Transactions on Computers*, C-33, 1984, pages 855–857.

[Lakhani and Dorairaj, 1987] G. Lakhani and R. Dorairaj. A VLSI implementation of all-pairs shortest path problem. *Proceedings of the International Conference on Parallel Processing*, 1987, pages 207–209.

[H. Lang, 1986] H. W. Lang. The instruction systolic array — a parallel architecture for VLSI. *Integration, the VLSI Journal*, 4, 1986, pages 65–74.

[H. Lang, 1987] H. W. Lang. ISA and SISA: two variants of a general purpose systolic array architecture. *Proceedings of the Second International Conference on Supercomputing*, volume 1, 1987, pages 460–465.

[H. Lang, 1988] H. W. Lang. Transitive closure on an instruction systolic array. *IEEE International Conference on Systolic Arrays*, 1988, pages 295–304.

[T. Lang and Moreno, 1988] T. Lang and J. H. Moreno. Graph-based partitioning of matrix algorithms for systolic arrays: application to transitive closure. *Proceedings of the International Conference on Parallel Processing*, 1988, pages 28–31.

[Lee, 1961] C. Y. Lee. An algorithm for path connections and its applications. *IRE Transactions on Electronic Computers*, EC-10(3), 1961, pages 346–365.

[Lehmann, 1977] D. J. Lehmann. Algebraic structures for transitive closure. Theoretical Computer Science, 4, 1977, pages 59–76.

[Leighton, 1983] F. T. Leighton. *Complexity Issues in VLSI.* MIT Press, Cambridge, MA, 1983.

[Leiserson, 1979] C. E. Leiserson. Systolic priority queues. *Proceedings of Caltech Conference on VLSI*, 1979, pages 199–214.

[Leiserson, 1981] C. E. Leiserson. Area-efficient VLSI computation. PhD Thesis, Carnegie-Mellon University, Computer Science Department, Pittsburgh, PA, 1981. Tech. Report CMU-CS-82-108.

[Lewis and S. Kung, 1986] P. S. Lewis and S. Y. Kung. Dependence graph based design of systolic arrays for the algebraic path problem. *Proceedings of the Twelfth Asilomar Conference on Signals, Systems, and Computers*, 1986, pages 13–18.

[Li and Wah, 1985] G. J. Li and B. W. Wah. The design of optimal systolic arrays. *IEEE Transactions on Computers*, C-34, 1985, pages 66–77.

[Lin and Wu, 1985] F. C. Lin and Wu. Systolic arrays for transitive closure algorithms. *Proceedings of International Symposium of VLSI System Designs*, 1985.

[Liu *et al.*, 1987] A. C. Liu, Y. M. Zou, and D. M. Liou. Divide-and-conquer for a shortest path problem. *Proceedings of the International Conference on Parallel Processing*, 1987, pages 210–212.

[Louka and Tchuente, 1989] B. Louka and M. Tchuente. An optimal solution for Gauss-Jordan elimination on systolic arrays. In *Systolic Array Processors*, eds.: J. McCanny *et al.*, Prentice Hall, Englewood Cliffs, NJ, 1989, pages 264–274.

[Loyens and van de Vorst, 1990] L. D. J. C. Loyens and J. G. G. van de Vorst. Two small parallel programming exercises. *Science of Computer Programming*, 15, 1990, pages 156–169.

[Mahr, 1984] B. Mahr. Iteration and summability in semirings. *Annual Discrete Mathematics*, 1984, pages 229–256.

[Martin, 1989] A. J. Martin. Formal program transformations for VLSI circuit synthesis. *Formal Development of Programs and Proofs*, ed.: E. W. Dijkstra, Addison-Wesley, Reading, MA, 1989, pages 59–80.

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

[Minty, 1957] G. J. Minty. A comment on the shortest-route problem. *The Journal of the Operation Research Society of America*, 5, 1957, page 724.

[Miranker and Winkler, 1984] W. L. Miranker and A. Winkler. Space-time representation of computational structures. *Computing*, 32, 1984, pages 93–114.

[Moldovan, 1983] D. I. Moldovan. On the design of algorithms for VLSI systolic arrays. *Proceedings of the IEEE*, 71(1), 1983, pages 113–120.

[Moldovan and Fortes, 1986] D. Moldovan and J. Fortes. Partitioning and mapping algorithms into fixed size systolic arrays. *IEEE Transactions of Computers*, C-35, 1986, pages 374–382.

[Moore, 1957] E. F. Moore. The shortest path in a maze. *Proceedings of an International Symposium on the Theory of Switching*, 1957, pages 285–292.

[Munro, 1971] J. I. Munro. Efficient determination of the transitive closure of a directed graph. *Information Processing Letters*, 1, 1971, pages 56–58.

[Moravek, 1970] J. Moravek. A note upon minimal path problem. *Journal of Mathematical Analysis and Applications*, 30, 1970, pages 702–717.

[Nash and Hansen, 1984] J. G. Nash and S. Hansen. Modified Fadeev algorithm for matrix multiplication. *Proceedings of the Society of Photo-Optical Instrument Engineers*, Real-Time Signal Processing, VII, 1984, pages 39–46.

[Nash *et al.*, 1981] J. G. Nash, S. Hansen, and G. R. Nudd. VLSI processor arrays for matrix multiplication. *VLSI Systems and Computations*, eds.: H. T. Kung, B. Sproull, and G. Steel, Computer Science Press, 1981, pages 367–378.

[Navarro *et al.*, 1986] J. J. Navarro, J. M. Laberia, and M. Valero. Solving matrix problems with no size-restriction using a one-dimensional systolic array processor. *Proceedings of the International Conference on Parallel Processing*, 1986, pages 676–683.

[Navarro *et al.*, 1987] J. J. Navarro, J. M. Laberia, and M. Valero. Partitioning: an essential step in mapping algorithms into systolic array processors. *IEEE Computer Magazine*, July 1987.

[Nemhauser, 1972] G. L. Nemhauser. A generalized permanent label setting algorithm for the shortest path between specified nodes. *Journal of Mathematical Analysis and Applications*, 38(2), 1972, pages 328–334.

[von Neumann, 1945] . von Neumann. First draft of report on the EDVAC. School of Electrical Engineering, University of Pennsilvania, 1945.

[Nicholson, 1966] T. A. J. Nicholson. Finding the shortest route between two points in a network. *Computer Journal*, 9, 1966, pages 275–280.

[Nunez and Valero, 1988] F. J. Nunez and M. Valero. A block algorithm for the algebraic path problem and its execution on a systolic array. *IEEE International Conference on Systolic Arrays*, 1988, pages 265–274.

[Nunez and Torralba, 1987] F. J. Nunez and N. Torralba. Transitive closure partitioning and its mapping to a systolic array. *Proceedings of the International Conference on Parallel Processing*, 1987, pages 564–566.

[Ore, 1962] O. Ore. *Theory of Graphs*. Providence, RI, American Mathematical Society, 1962.

[Paige and Kruskal, 1985] R. C. Paige and C. P. Kruskal. Parallel algorithms for shortest path problems. *Proceedings of the International Conference on Parallel Processing*, 1985, pages 14–20.

[Pape, 1974] U. Pape. Implementation and efficiency of Moore-algorithm for the shortest route problem. *Mathematical Programming*, 7, 1974, pages 212–222.

[Pease, 1967] M. C. Pease. Matrix inversion using parallel processing. *Journal ACM*, 14(4), 1967, pages 757–764.

[Pierce, 1975] A. R. Pierce. Bibliography on algorithms for shortest path, shortest spanning tree and related circuit routing problems (1956–1974). *Networks*, 5(2), 1975, pages 129–149.

[Pollack, 1961a] M. Pollack. Solutions of the kth best route through a network — a review. *Journal of Mathematical Analysis and Applications*, 3, 1961, pages 547–559.

[Pollack, 1961b] M. Pollack. The kth best rote through a network. *The Journal of the Operations Research Society of America*, 6, 1961, pages 578–580.

[Pollack and Wiebenson, 1960] M. Pollack and W. Wiebenson. Solutions of the shortest-route problem — a review. *The Journal of the Operations Research Society of America*, 8, 1960, pages 224–230.

[Press *et al.*, 1989] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical recipes: the art of scientific computing.* Cambridge University Press, 1989.

[Purdom, 1970] P. Purdom Jr. A transitive closure algorithm. *BIT*, 10, 1970, pages 76–94.

[Quinn and Yoo, 1984] M. J. Quinn and Y. B. Yoo. Data structures for the efficient solution of graph theoretic problems on tightly coupled MIMD computers. *Proceedings of the International Conference on Parallel Processing*, 1984, pages 431–438.

[Quinton, 1984] P. Quinton. Automatic synthesis of systolic arrays from uniform recurrent equations. *Proceedings of the Eleventh Symposium on Computer Architecture*, 1984, pages 208–214.

[Quinton, 1987] P. Quinton. The systematic design of systolic arrays. *Automata Networks in Computer Science*, eds.: F. Fogelman-Soulie, Y. Robert, and M. Tchuente, Manchester University Press, 1987, pages 229–260.

[Quinton, 1990] P. Quinton and Yves Robert. *Systolic algorithms & architectures.* Prentice Hall, 1991.

[Rapaport and Abramson, 1959] H. Rapaport and P. Abramov. An analog computer for finding an optimum route through a communications network. *IRE Transactions on Communications Systems*, CS-7, 1959, pages 37–42.

[Reghbati, 1978] E. Reghbati and D. G. Corneil. Parallel computations in graph theory. *SIAM Journal of Computing*, 7, 1978, pages 230–236.

[Robert, 1987] Y. Robert. Systolic algorithms and architectures. *Automata Networks in Computer Science*, eds.: F. Fogelman-Soulie, Y. Robert, and M. Tchuente, Manchester University Press, 1987, pages 187–228.

[Robert and Trystram, 1986a] Y. Robert and D. Trystram. Systolic solution of the algebraic path problem. *International Workshop on Systolic Arrays*, 1986. In *Systolic Arrays*, eds.: W. Moore, A. McCabe, and R. Urquhart, Adam Hilger, Bristol and Boston, 1987, pages 171–180.

[Robert and Trystram, 1986b] Y. Robert and D. Trystram. Parallel implementation of the algebraic path problem. *Proceedings of the International Conference CONPAR*, 1986. In *Lecture Notes in Computer Science*, 237, Springer-Verlag, 1986, pages 149–156.

[Rote, 1985] G. Rote. A systolic array algorithm for the algebraic path problem (shortest path; matrix inversion). *Computing*, 34, 1985, pages 191–219.

[Saksena and Kumar, 1968] J. P. Saksena and S. Kumar. The routing problem with 'K' specified nodes. *The Journal of the Operations Research Society of America*, 16, 1968, pages 909–913.

[Savage and Ja'Ja', 1981] C. Savage and J. Ja'Ja'. Fast efficient algorithms for some graph problems. *SIAM Journal of Computing*, 4, 1981, pages 682–691.

[Schnorr, 1978] C. P. Schnorr. An algorithm for transitive closure with linear expected time. *SIAM Journal of Computing*, 7(2), 1978, pages 127–133.

[Schwiegelshohn and Thiele, 1986] U. Schwiegelshohn and L. Thiele. On the systolic detection of shortest routes. *Proceedings of the International Conference on Parallel Processing*, 1986, pages 762–764.

[Shimbel, 1954] A. Shimbel. Structure in Communication Nets. *Proceedings of the Symposium on Information Networks*, Polytechnic Institute of Brooklyn, 1954, pages 12–14.

[Simon, 1985] K. Simon. An improved algorithm for transitive closure on acyclic digraphs. Tech. Report A85/13, Universitat des Saarlandes, 1985.

[A. Smith, 1971] A. R. Smith. Two-dimensional formal languages and pattern recognition by cellular automata. *The Twelfth IEEE Symposium on Switching and Automata Theory*, 1971, pages 144–152.

[B. Smith, 1978] B. J. Smith. A pipelined shared resource MIMD computer. *Proceedings of the International Conference on Parallel Processing*, 1978, pages 199–204.

[Snepscheut, 1986] J. L. A. van de Snepscheut. A derivation of a distributed implementation of Warshall's algorithm. *Science of Computer Programming*, 7, 1986, pages 55–60.

[Spira, 1973] P. M. Spira. A new algorithm for finding all shortest paths in a graph of positive arcs in average time $O(n^2 \log^2 n)$. *SIAM Journal of Computing*, 2, 1973, pages 28–32.

[Spira and Pan, 1973] P. M. Spira and A. Pan. On finding and updating shortest paths and spanning trees. *Conference Record, Fourteenth Annual Symposium on Switching and Automata Theory*, 1971, pages 82–84.

[Stone, 1973] H. S. Stone. An efficient parallel algorithm for a tridiagonal linear system. *Journal ACM*, 20, 1973, pages 27–38.

[Strassen, 1969] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13, 1969, page 354–356.

[Tarjan, 1975] R. E. Tarjan. *Solving path problems on directed graphs.* Technical Report, Computer Science Department, Stanford University, 1975.

[Tarjan, 1976] R. E. Tarjan. Graph theory and Gaussian elimination. In *Sparse Matrix Computations*, eds.: J. R. Bunch and D. J. Rose, Academic Press, New York, 1976, pages 3–22.

[Tarjan, 1981a] R. E. Tarjan. A unified approach to path problems. *Journal ACM*, 28, 1981, pages 577–593.

[Tarjan, 1981b] R. E. Tarjan. Fast algorithms for solving path problems. *Journal ACM*, 28, 1981, pages 594–614.

[Thompson, 1979] C. D. Thomson. Area-time complexity for VLSI. *The Eleventh Annual ACM Symposium on Theory of Computing*, 1979, pages 81–88.

[Thorelli, 1966] L. E. Thorelli. An algorithm for computing all paths in a graph. *BIT*, 6, 1966, pages 347–349.

[Ullman, 1984] J. D. Ullman. *Computational aspects of VLSI.* Computer Science Press, Rockville, MD, 1984.

[Umeo, 1985] H. Umeo. A class of SIMD machines simulated by systolic arrays. *Journal of Parallel and Distributed Computing*, 2, 1985, pages 391–403.

[Varga, 1962] R. S. Varga. *Matrix iterative analysis.* Prentice-Hall, Englewood Cliffs, NJ, 1962.

[Vliet, 1978] D. van Vliet. Improved shortest path algorithm for transportation networks. *Transportation Res.*, 12, 1978, pages 7–20.

[Wagner, 1976] R. A. Wagner. A shortest path algorithm of edge-sparse graphs. *Journal ACM*, 23, 1976, pages 50–57.

[Warshall, 1962] S. Warshall. A theorem on Boolean matrices. *Journal ACM*, 9, 1962, pages 11–12.

[Whiting and Hiller, 1960] P. D. Whiting and J. A. Hiller. A method for finding the shortest route through a road network. *Operational Research Quarterly*, 11, 1960, pages 37–40.

[Wilkinson, 1956] R. I. Wilkinson. Abridged bibliography of articles on toll alternate routing. *Bell System Technical Journal*, 35, 1956, page 507.

[Yoeli, 1961] M. Yoeli. A note on a generalization of Boolean matrix theory. *American Mathematical Monthly*, 68, 1961, pages 552–557.

[Zimmermann, 1981] U. Zimmermann. Linear and combinatorial optimization in ordered algebraic structures. *Annals of Discrete Mathematics*, 10, 1981, pages 1–380.

Also, there are two papers that I refer to in this report, but was unable to obtain them and therefore did not read. They are listed below.

- (Rote, 1984) G. Rote. *A systolic array for the algebraic path problem (which includes the inverse of a matrix and shortest distances is a graph).* Rechenzentrum Graz, Bericht RZG-101, 1984.
- (Roy, 1957) B. Roy. Transitivite et connexite. Comptes Rendus, Acad. Sci, Paris, 249, 1959, pages 216–218.

Index