Chapter 5. Path Problems in Graphs and Matrix Multiplication

In this chapter we concentrate on path problems in graphs. The problems of computing shortest or longest paths or computing the k least cost paths between all pairs of points in a graph are typical examples. The best known algorithms for these problems differ only slightly. In fact, they are all special cases of an algorithm for solving general path problems on graphs. General path problems over closed semi-rings and Kleene's algorithm for solving them are dealt with in Section 5.1, special cases are then treated in Section 5.2. The algebraic point of view allows us to formulate the connection between general path problems and matrix multiplication in an elegant way: matrix multiplication in a semi-ring and solution of a general path problem have the same order of complexity. In Section 5.4 we consider fast algorithms for multiplication of matrices over a ring. This is then applied to boolean matrices. Section 5.7 contains a lower bound on the complexity of the boolean matrix product.

5.1. General Path Problems

Let us recall some notation. Let $G = (V, E)$ with $V = (v_1, \ldots, v_n)$ be a digraph. A path p from v_i to v_j is a sequence $w_0, w_1, \ldots, w_k$ of nodes with $v_i = w_0, v_j = w_k$ and $(w_l, w_{l+1}) \in E$ for $0 \leq l \leq k-1$. The length of this path is k. Note that there is always the path of length 0 from a node to itself. We use P_{ij} to denote the set of all paths from v_i to v_j:

$$P_{ij} = \{p;\ p \text{ is a path from } v_i \text{ to } v_j\}.$$

In a sense Kleene's algorithm computes set P_{ij} for all i and j. We describe this algorithm in a very general setting first and apply it to some special cases in Section 5.2. The general framework is provided by closed semi-rings.

Definition: a) A set S with distinguished elements 0 and 1 and binary operations $\oplus$ and $\odot$ is a **semi-ring** if

(1) $(S, \oplus, 0)$ is a commutative monoid, i.e., for all $a, b, c \in S$

$$(a \oplus b) \oplus c = a \oplus (b \oplus c)$$
$$a \oplus b = b \oplus a$$
$$a \oplus 0 = a.$$

(2) $(S, \odot, 1)$ is a monoid, i.e., for all $a, b, c \in S$

$$(a \odot b) \odot c = a \odot (b \odot c)$$
$$a \odot 1 = 1 \odot a = a.$$

(3) Multiplication distributes over addition and 0 is a null-element with respect to multiplication, i.e., for all $a, b, c \in S$

$$(a \oplus b) \odot c = (a \odot c) \oplus (b \odot c)$$
$$c \odot (a \oplus b) = (c \odot a) \oplus (c \odot b)$$
$$0 \odot a = a \odot 0 = 0.$$

b) A semi-ring S is a **closed semi-ring** if in addition infinite sums exist, i.e., with every family $\{a_i;\ i \in I\}$ of elements of S with countable (finite or infinite) index set I there is associated an element $\bigoplus_{i \in I} a_i$, its sum. Infinite sums satisfy the following laws:

(4) For finite non-empty index set $i = \{i_1, i_2, \ldots, i_k\}$

$$\bigoplus_{i \in I} a_i = a_{i_1} \oplus a_{i_2} \oplus \cdots \oplus a_{i_k}$$

and for empty index set $I = \emptyset$

$$\bigoplus_{i \in \emptyset} a_i = 0.$$

(5) The result of a summation does not depend on the ordering of the factors, i.e., for every index set I and every partition $\{I_j;\ j \in J\}$ of I with

$$\bigcup_{j \in J} I_j = I \text{ and } I_i \cap I_k = \emptyset \text{ for } i \neq k$$

we have

$$\bigoplus_{i \in I} a_i = \bigoplus_{j \in J} \Big(\bigoplus_{i \in I_j} a_i \Big).$$

(6) Multiplication distributes over infinite sums, i.e.,

$$\Big(\bigoplus_{i \in I} a_i \Big) \odot \Big(\bigoplus_{j \in J} b_j \Big) = \bigoplus_{i \in I} \Big(\bigoplus_{j \in J} a_i \odot b_j \Big). \qquad \blacksquare$$

In Exercise 3 we draw some conclusions from these axioms.

Examples: 1) The boolean semi-ring $B = (\{0,1\}, \vee, \wedge, 0, 1)$. The basic operations are boolean OR (= addition)

$$\bigvee_{i \in I} a_i = \begin{cases} 1 & \text{if there is } i \in I \text{ with } a_i = 1; \\ 0 & \text{otherwise} \end{cases}$$

with neutral element 0 and boolean AND (= multiplication)

$$x \wedge y = \min(x, y)$$

with neutral element 1. The boolean semi-ring is the simplest closed semi-ring. We use it to determine the existence of paths between points.

2) The $(\min,+)$-semi-ring $(\mathbb{R}\cup\{\infty\}\cup\{-\infty\},\min,+,\infty,0)$ of reals with additional elements ∞ and $-\infty$. The operations are infimum, denoted min, with neutral element ∞ and addition with neutral element 0. We define $(-\infty)+\infty=\infty$. Here min corresponds to addition and + corresponds to multiplication. We use the $(\min,+)$-semi-ring to compute least cost paths. ∎

For further examples of closed semi-rings we refer to the exercises. Let $G=(V,E)$ be a digraph with $V=\{v_1,\ldots,v_n\}$, let S be a closed semi-ring and let $c:E\to S$ be a labelling of the edges of G by elements of S. We extend c to paths and sets of paths. If $p=w_0,w_1,\ldots,w_k$ is a path, then $c(p)=c(w_0,w_1)\odot c(w_1,w_2)\odot\cdots\odot c(w_{k-1},w_k)$; if $k=0$ then $c(p)=1$. If P is a set of paths then $c(P)=\bigoplus_{p\in P}c(p)$.

Definition: The **general path problem** is to compute $a_{ij}=\bigoplus_{p\in P_{ij}}c(p)$ for all i,j, $1\le i,j\le n$. ∎

We need one further operation in closed semi-rings. For $a\in S$ we define the **closure** a^* of a by $a^*=1\oplus a\oplus a^2\oplus\cdots=\bigoplus_{i\ge 0}a^i$. Here $a^0=1$ and $a^{i+1}=a\odot a^i$. Note that $0^*=1$.

Examples: In the boolean semi-ring we have $a^*=1$ for all a in the $(\min,+)$-semi-ring of reals we have $a^*=0$ for $a\ge 0$ and $a^*=-\infty$ for $a<0$. In both cases, the closure is trivial to compute. ∎

Definition: $P_{ij}^{(k)}$ is the set of paths $p=w_0,w_1,\ldots,w_l$ with

(1) $w_0=v_i$, $w_l=v_j$, i.e., p goes from v_i to v_j;

(2) $w_h=v_{g_h}$ for some $g_h\le k$ and all $1\le h<l$, i.e., all intermediate points on the path are in $\{v_1,\ldots,v_k\}$;

(3) if $i=j>k$ then $l>0$, i.e., the trivial path of length 0 is included in $P_{ii}^{(k)}$ only if $i\le k$. ∎

With $p=v_i,v_1,v_3,v_j$ we have $p\in P_{ij}^{(3)}$ and $p\notin P_{ij}^{(2)}$. Also path $p=v_i\in P_{ii}^{(i)}$ and $p\notin P_{ii}^{(i-1)}$.

Kleene's algorithm is an application of dynamic programming. It computes iteratively, $k=0,1,2,\ldots,n$, matrices $a_{ij}^{(k)}$, $1\le i,j\le n$, where

$$a_{ij}^{(k)}=\bigoplus_{p\in P_{ij}^{(k)}}c(p).$$

Next we derive recursion formulae for computing $a_{ij}^{(k)}$. A path in $P_{ij}^{(0)}$ can have no intermediate point (by part (2) of the definition of $P_{ij}^{(k)}$) and it must have length at least 1 (this is obvious for $i\ne j$ and follows from part (3) of the definition of $P_{ij}^{(k)}$

for $i = j$). Thus all paths in $P_{ij}^{(0)}$ must have length exactly one, i.e., consist of a single edge. Hence

$$a_{ij}^{(0)} = \textbf{if } (v_i, v_j) \in E \textbf{ then } c(v_i, v_j) \textbf{ else } 0 \textbf{ fi}.$$

Next suppose $k > 0$. A path $p \in P_{ij}^{(k)}$ either passes through node k or does not. Also if $i = j = k$ then the path of length 0 belongs to $P_{ij}^{(k)}$ and does not belong to $P_{ij}^{(k-1)}$. Thus by property (5) of a closed semi-ring

$$\begin{aligned}\bigoplus_{p \in P_{ij}^{(k)}} c(p) &\overset{(5)}{=} \textbf{if } (i = j = k) \textbf{ then } 1 \textbf{ else } 0 \textbf{ fi} \\ &\quad \oplus \bigoplus_{p \in P_{ij}^{(k-1)}} c(p) \oplus \bigoplus_{\substack{p \in P_{ij}^{(k)} - P_{ij}^{(k-1)} \\ p \text{ has length at least } 2}} c(p) \\ &= \textbf{if } (i = j = k) \textbf{ then } 1 \textbf{ else } 0 \textbf{ fi} \\ &\quad \oplus a_{ij}^{(k-1)} \oplus \bigoplus_{\substack{p \in P_{ij}^{(k)} - P_{ij}^{(k-1)} \\ p \text{ has length at least } 2}} c(p).\end{aligned}$$

A path $p \in P_{ij}^{(k)} - P_{ij}^{(k-1)}$ of length at least 2 has the form $v_i \ldots v_k \ldots v_j$. It can be divided into three parts, into an initial segment p' of length at least 1 which leads from v_i to v_k without going through v_k on the way, i.e., $p' \in P_{ik}^{(k-1)}$, into a terminal segment p''' of length at least 1 which leads from v_k to v_j without going through v_k on the way, i.e., $p''' \in P_{kj}^{(k-1)}$ and into an intermediate segment p'' of length at least 0 which leads from v_k to v_k going l-times for some number $l \geq 0$ through v_k, i.e., $p'' \in P_{kk}^{(k)}$. Conversely, if $p' \in P_{ik}^{(k-1)}$, $p'' \in P_{kk}^{(k)}$ and $p''' \in P_{kj}^{(k-1)}$ then path $p'p''p'''$ obtained by concatenating p', p'' and p''' is in $P_{ij}^{(k)} - P_{ij}^{(k-1)}$. Here it is important to observe that $P_{ik}^{(k-1)}$ and $P_{kj}^{(k-1)}$ contain only paths of length at least 1. This is obvious for $i \neq k$ ($k \neq j$) and follows from part (3) of the definition of $P_{ij}^{(i)}$ ($P_{ij}^{(j)}$ for $i = k$ ($k = j$). Thus

$$\begin{aligned}\bigoplus_{\substack{p \in P_{ij}^{(k)} - P_{ij}^{(k-1)} \\ p \text{ has length at least } 2}} c(p) &\overset{(5)}{=} \bigoplus_{p' \in P_{ik}^{(k-1)}} \bigoplus_{p'' \in P_{kk}^{(k)}} \bigoplus_{p''' \in P_{kj}^{(k-1)}} c(p') \odot c(p'') \odot c(p''') \\ &\overset{(6)}{=} \left(\bigoplus_{p' \in P_{ik}^{(k-1)}} c(p') \right) \odot \left(\bigoplus_{p'' \in P_{kk}^{(k)}} c(p'') \right) \odot \left(\bigoplus_{p''' \in P_{kj}^{(k-1)}} c(p''') \right) \\ &= a_{ik}^{(k-1)} \odot \left(\bigoplus_{p'' \in P_{kk}^{(k)}} c(p'') \right) \odot a_{kj}^{(k-1)}.\end{aligned}$$

A path $p'' \in P_{kk}^{(k)}$ either has length 0 or it is a proper path of length > 0 which goes l-times through v_k for some number l ($l \geq 0$). In the latter case it consists of $l+1$ subpaths in $P_{kk}^{(k-1)}$. Hence

$$\bigoplus_{p'' \in P_{kk}^{(k)}} c(p'') \overset{(5)}{=} c(\text{path of length } 0) \oplus \bigoplus_{l \geq 0} \bigoplus_{\substack{p'' \in P_{kk}^{(k)} \\ l \text{ intermediate points of } p'' \\ \text{are equal to } v_k}} c(p'')$$

$$\overset{(6)}{=} 1 \oplus \bigoplus_{l \geq 0} \left(\bigoplus_{p'''' \in P_{kk}^{(k-1)}} c(p'''') \right)^{l+1}$$

$$= 1 \oplus \bigoplus_{l \geq 0} \left(a_{kk}^{(k-1)} \right)^{l+1}$$

$$= \left(a_{kk}^{(k-1)} \right)^{*}.$$

The set of recursion equations derived above immediately leads to the algorithm of Program 1.

---

(1) **for** $i, j \in \{1, \ldots, n\}$
(2) **do** $a_{ij}^{(0)} \leftarrow$ **if** $(v_i, v_j) \in E$ **then** $c(v_i, v_j)$ **else** 0 **fi od**;
(3) **for** k **from** 1 **to** n
(4) **do for** $i, j \in \{1, \ldots, n\}$
(5) **do** $a_{ij}^{(k)} \leftarrow a_{ij}^{(k-1)} \oplus \left(a_{ik}^{(k-1)} \odot \left(a_{kk}^{(k-1)}\right)^{*} \odot a_{kj}^{(k-1)}\right)$;
(6) **if** $(i = j = k)$ **then** $a_{ii}^{(i)} \leftarrow a_{ii}^{(i)} \oplus 1$ **fi**
(7) **od**
(8) **od**.

Program 1

Since $P_{ij}^{(n)} = P_{ij}$ we have $a_{ij} = a_{ij}^{(n)}$ and the general path problem is solved.

Theorem 1. *Kleene's algorithm solves the general path problem in $\Theta(n^3)$ semiring operations $\oplus, \odot$ and *.*

Proof: Line (5) is executed once for each triple i, j, k with $1 \leq i, j, k \leq n$. ∎

Kleene's algorithm as described above uses $\Theta(n^3)$ storage locations. With a little skill this can be reduced to $\Theta(n^2)$ as follows. Note that we left open the order of execution in line (4). Therefore we can replace lines (4) to (7) by lines (4′) to (10′) where we use the identity $1 \oplus a_{kk} \oplus (a_{kk} \odot a_{kk}^{*} \odot a_{kk}) = 1 \oplus a_{kk} \oplus a_{kk}^2 \odot (\bigoplus_{i \geq 0} a_{kk}^i)$ in line (10′).

(4′) **for** $i, j \in \{1, \ldots, n\} - \{k\}$
(5′) **do** $a_{ij} \leftarrow a_{ij} \oplus (a_{ik} \odot a_{kk}^* \odot a_{kj})$ **od**;
(6′) **for** $i \in \{1, \ldots, n\} - \{k\}$
(7′) **do** $a_{ik} \leftarrow a_{ik} \oplus (a_{ik} \odot a_{kk}^* \odot a_{kk})$ **od**;
(8′) **for** $j \in \{1, \ldots, n\} - \{k\}$
(9′) **do** $a_{kj} \leftarrow a_{kj} \oplus (a_{kk} \odot a_{kk}^* \odot a_{kj})$ **od**;
(10′) $a_{kk} \leftarrow a_{kk}^*$.

5.2. Two Special Cases: Least Cost Paths and Transitive Closure

We take a closer look at two applications of the results of the previous section. Further applications can be found in the exercises.

Transitive Closure of Digraphs: Let $G = (V, E)$ be a digraph. Graph $H(G) = (V, E')$ is called transitive closure of G if $(v, w) \in E'$ if and only if there is a path from v to w in G. We use the boolean semi-ring and define

$$c(v, w) = \begin{cases} 1 & \text{if } (v, w) \in E; \\ 0 & \text{if } (v, w) \notin E. \end{cases}$$

Then $c(p) = 1$ for all paths p and therefore for every set P of paths

$$\bigoplus_{p \in P} c(p) = \begin{cases} 1 & \text{if } P \neq \emptyset; \\ 0 & \text{if } P = \emptyset. \end{cases}$$

We can thus apply Kleene's algorithm to compute the transitive closure of a graph. Because of $a^* = 1$ for all elements of the boolean semi-ring, line (5) of Kleene's algorithm simplifies to

$$a_{ij}^{(k)} \leftarrow a_{ik}^{(k-1)} \vee (a_{ik}^{(k-1)} \wedge a_{kj}^{(k-1)})$$

where $\wedge$ is boolean AND $\vee$ is boolean OR.

Theorem 1. *The matrix representation of the transitive closure of a digraph can be computed in time* $\Theta(n^3)$ *and space* $\Theta(n^2)$. ∎

Theorem 1 is not very impressive. After all, we already know an $\vee(n \cdot e_{red})$ algorithm from Sections 4.3 and 4.6 (e_{red} is the number of edges in a transitive reduction of G). However, we will see in Sections 4.4 and 5.5 that Theorem 1 can be improved to yield an $O(n^{2.39})$ algorithm.

All Pairs Least Cost Paths: Let $G = (V, E)$ be a digraph and let $l : E \to \mathbb{R}$ be a labelling of the edges with real numbers. We use the $(\min, +)$-semi-ring of reals. Then

$$\bigoplus_{\substack{p \text{ path from} \\ v \text{ to } w}} l(p) = \text{Inf}\{l(v, v_1) + l(v_1, v_2) + \cdots + l(v_k, w); \\ p = v, v_1, \ldots, v_k, w \text{ is a path from } v \text{ to } w\}$$

is the minimal cost of a path from v to w. We can thus use Kleene's algorithm to solve the all pairs least cost path problem.

Theorem 2. *The all pairs least cost path problem can be solved in time $\Theta(n^3)$ and space $\Theta(n^2)$ by Kleene's algorithm.* ∎

Theorem 2 has to be seen in contrast with Theorem 7 of Section 4.7.4 where we described an $O(n \cdot e \cdot (\log n)/\log(e/n)) = O(n^3)$ algorithm for the all pairs least cost path problem. Although the algorithm of 4.7.4 is asymptotically never worse than Kleene's algorithm, it is nevertheless inferior for small n or dense graphs.

5.3. General Path Problems and Matrix Multiplication

We resume the discussion on the general path problem. Let $G = (V, E)$ be a digraph with $V = \{v_1, \ldots, v_n\}$, let S be a closed semi-ring and let $c : E \to S$ be a labelling of the edges of G by elements of S. We have shown how to compute $\bigoplus_{p \in P_{ij}} c(p)$ for every i and j. By property (5) of closed semi-rings we can rewrite this sum as

$$\bigoplus_{l \geq 0} \bigoplus_{\substack{p \in P_{ij} \text{ and} \\ p \text{ has length } l}} c(p).$$

The inner sum is now easily represented as a matrix product. Let matrix $A_G = (a_{ij})_{1 \leq i,j \leq n}$ be defined as follows:

$$a_{ij} = \begin{cases} c(v_i, v_j) & \text{if } (v_i, v_j) \in E; \\ 0 & \text{otherwise.} \end{cases}$$

Then the sum above is equal to entry (i, j) of the l-th power A_G^l of matrix A_G, or more precisely:

Definition: Let M_n be the set of all $n \times n$ matrices with elements of a closed semi-ring S. Addition and multiplication of matrices are defined as usual, i.e., $(a_{ij}) \oplus (b_{ij}) = (a_{ij} \oplus b_{ij})$ and $(a_{ij}) \odot (b_{jk}) = (\bigoplus_{j=1}^{n} a_{ij} \odot b_{jk})$. We use 0 to denote the all zero matrix and $I = (\delta_{ij})$ to denote the identity matrix, i.e., $\delta_{ij} = 1$ if $i = j$ and $\delta_{ij} = 0$ if $i \neq j$. ∎

It is easy to see that $(M_n, \oplus, \odot, 0, I)$ is a closed semi-ring (Exercise 8). We define the powers of matrix $A \in M_n$ as usual:

$$A^0 = I$$
$$A^{k+1} = A \odot A^k$$

and the closure of A by

$$A^* = I \oplus A \oplus A^2 \oplus \cdots = \bigoplus_{l \geq 0} A^l.$$

We are now able to formalize the connection between the powers of A_G and the labels of paths of a certain length.

Theorem 1. *Let $A_G^l = (a_{ij}^{(l)})_{1\le i,j\le n}$ be the l-th power of matrix A_G. Then*

$$a_{ij}^{(l)} = \bigoplus_{\substack{p\in P_{ij}\\ \text{length}(p)=l}} c(p).$$

Proof: (By induction on l) For $l = 0$ and $l = 1$ the claim is obvious from the definition of I and A_G. Assume $l > 1$. A path p of length l from v_i to v_j consists of an edge, say from v_i to v_k, and a path of length $l-1$ from v_k to v_j. Hence

$$\begin{aligned}
\bigoplus_{\substack{p\in P_{ij}\\ \text{length}(p)=l}} c(p) &\overset{(5),(6)}{=} \bigoplus_{k=1}^{n}\Big(\bigoplus_{\substack{p'\in P_{kj}\\ \text{length}(p')=l-1}} c(v_i,v_k)\odot c(p')\Big)\\
&\overset{(6)}{=} \bigoplus_{k=1}^{n}\Big(c(v_i,v_k)\odot \bigoplus_{\substack{p'\in P_{kj}\\ \text{length}(p')=l-1}} c(p')\Big)\\
&\overset{\text{I.H.}}{=} \bigoplus_{k=1}^{n} a_{ik}^{(1)}\odot a_{kj}^{(l-1)}\\
&= a_{ij}^{(l)}.
\end{aligned}$$

∎

Corollary 1. *Let $A_G^* = (b_{ij})_{1\le i,j\le n}$. Then*

$$b_{ij} = \bigoplus_{p\in P_{ij}} c(p).$$

Proof: Follows immediately from Theorem 1 and the definition of A_G^*. ∎

General path problems are equivalent to computing the closure of matrix A_G. Kleene's algorithm allows us to compute the closure of a matrix with $\Theta(n^3)$ additions, multiplications and closure operatios of semi-ring elements. The same number of operations is required for multiplying two matrices according to the classical method, the school method. According to the school method we multiply two matrices A and B by computing the scalar product of every row of A with every column of B. We show next, that there is a deeper meaning behind the fact, that Kleene's algorithm for computing the closure of a matrix and the highschool method for multiplying matrices have the same complexity.

Theorem 2. *If there is an algorithm which computes the closure of an $n \times n$ matrix using $A(n)$ additions, multiplications and closure operations of elements of semi-ring S and if $A(3n) \leq c \cdot A(n)$ for some $c \in \mathbb{R}$ and all $n \in \mathbb{N}$ then there is an algorithm for multiplying two $n \times n$ matrices with $M(n) = O(A(n))$ additions and multiplications.*

Proof: Let A and B be the two $n \times n$ matrices. Let C be the following $3n \times 3n$ matrix:

$$C = \begin{pmatrix} 0 & A & 0 \\ 0 & 0 & B \\ 0 & 0 & 0 \end{pmatrix}.$$

The closure C^* of C can be computed in $A(3n) \leq c \cdot A(n)$ operations. Since

$$C^2 = \begin{pmatrix} 0 & 0 & A \odot B \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \text{ and } C^3 = C^4 = \cdots = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

we have

$$C^* = \begin{pmatrix} I & A & A \odot B \\ 0 & I & B \\ 0 & 0 & I \end{pmatrix}$$

and product $A \odot B$ can be found in the right upper corner of C^*. Thus the product of two $n \times n$ matrices can be computed in $M(n) = A(3n) \leq c \cdot A(n) = O(A(n))$ operations. ∎

Let us briefly discuss the assumption $\exists c : A(3n) \leq c \cdot A(n)$. First of all, this assumption stipulates a polynomial bound on the growth of $A(n)$,

$$A(n) \leq c \cdot A(n/3) \leq c^2 \cdot A(n/9) \leq \cdots \leq c^{\log_3 n} \cdot A(1)$$

for n a power of 3. Thus $A(n) = O(c^{\log_3 n})$. Since we already know how to compute the closure in $O(n^3)$ operations, a polynominal bound on $A(n)$ is not a severe restriction. Secondly, the assumption stipulates a certain "smoothness" of $A(n)$. Function $A(n)$ is not allowed to grow in jumps. Many functions such as n^α and $n^\alpha \cdot \log\ n$ where $\alpha \geq 0$, satisfy the assumption made in Theorem 2. Surprisingly, the reverse of Theorem 2 is also true.

Theorem 3. *If the product of two $n \times n$ matrices can be computed with $M(n)$ additions and multiplications of semi-ring elements and if $4 \cdot M(n/2) \leq M(n)$ and $M(2n) \leq c \cdot M(n)$ for some c and all n then the closure of an $n \times n$ matrix can be computed with $A(n) = O(M(n))$ additions, multiplications and closure operations of semi-ring elements.*

Proof: We describe a recursive algorithm which uses only $A(n) = O(M(n))$ semi-ring operations. Let X be any $n \times n$ matrix over a closed semi-ring. We assume at first that $n = 2^k$ is a power of 2 and extend the result to arbitrary n later on.

For $k = 0$ and hence $n = 1$ the closure of matrix $X = (x)$ is simply $X^* = (x^*)$. Thus $A(1) = 1$. Assume $k > 0$. We split X into four $n/2 \times n/2$ matrices B, C, D and E such that

$$X = \begin{pmatrix} B & C \\ D & E \end{pmatrix}$$

and interpret the splitting of X in terms of graphs. Matrix X corresponds to a graph $G = (V, E)$ with $V = \{v_1, \ldots, v_n\}$, $E = V \times V$, and labelling $c : E \to S$ with $c(v_i, v_j) = x_{ij}$. Let $V_1 = \{v_1, \ldots, v_{n/2}\}$ and $V_2 = \{v_{n/2+1}, \ldots, v_n\}$. Then B describes the labelling of edges which lead from nodes in V_1 to nodes in V_1, C describes the labelling of edges which lead from nodes in V_1 to nodes in V_2, ... Figure 1shows the relations in form of a transition diaram.

Figure 1. Relations after splitting X described by a graph

What is the interpretation of matrix

$$X^* = \begin{pmatrix} F & G \\ H & K \end{pmatrix}?$$

F is the sum of the labellings of all paths which lead from nodes in V_1 to nodes in V_1, ... Let $v, w \in V_1$. A path from v to w has the following form. It begins in v and then goes through some nodes in V_1 using edges in B, then leaves V_1 and enters V_2 via an edge in C, then goes through some nodes in V_2 using edges in E, ... More precisely, we can say that a path from v to w consists of elementary pieces which connect a node in V_1 with another node in V_1 without going through a node in V_1 on the way. Thus an elementary piece is either an edge between two nodes in V_1, i.e., an element of B, or it consists of a single edge in C followed by a path in V_2 consisting of edges in E only, i.e., an element of E^*, followed by an edge from V_2 to V_1, i.e., an element of D. Elementary pieces are thus given by $B \oplus (C \odot E^* \odot D)$. Hence

$$F = (B \oplus (C \odot E^* \odot D))^*.$$

Similarly,

$$G = F \odot C \odot E^*,$$

$$H = E^* \odot D \odot F$$

and

$$K = E^* \oplus (E^* \odot D \odot F \odot C \odot E^*).$$

We leave it to the reader to formally justify these identities. The formulae above suggest the algorithm of Program 2 for computing F, G, H and K from B, C, D and E.

$T_1 \leftarrow E^*$;
$T_2 \leftarrow C \odot T_1$;
$F \leftarrow (B \oplus (T_2 \odot D))^*$;
$G \leftarrow F \odot T_2$;
$T_3 \leftarrow T_1 \odot D$;
$H \leftarrow T_3 \odot F$;
$K \leftarrow T_1 \oplus (T_3 \odot G)$.

Program 2

The execution of this program supposes to compute the closure of two $n/2 \times n/2$ matrices, six products of $n/2 \times n/2$ matrices and two sums of $n/2 \times n/2$ matrices. If we use the same algorithm recursively then the closure of an $n/2 \times n/2$ matrix can be computed in $A(n/2)$ operations. Summing two $n/2 \times n/2$ matrices takes $(n/2)^2$ additions. Thus

$$A(n) = 2 \cdot A(n/2) + 6 \cdot M(n/2) + 2 \cdot (n/2)^2.$$

Since $M(n) \geq 4 \cdot M(n/2)$ and hence $M(n) \geq n^2 \cdot M(1) \geq n^2$ this is simplified to

$$A(n) \leq 2 \cdot A(n/2) + 8 \cdot M(n/2).$$

We show $A(n) \leq 4 \cdot M(n)$. Since $A(1) = M(1) = 1$ this is certainly true for $n = 1$. If $n = 2^k > 1$ then

$$\begin{aligned} A(n) &\leq 2 \cdot A(n/2) + 8 \cdot M(n/2) \\ &\leq 16 \cdot M(n/2) \\ &\leq 4 \cdot M(n) \end{aligned}$$

by assumption. If n is not a power of two we fill up matrix X with zeroes until we obtain a matrix $\overline{X}$ of dimension $2^{\lceil \log\ n \rceil}$

$$\overline{X} = \begin{pmatrix} X & 0 \\ 0 & 0 \end{pmatrix}$$

and compute $\overline{X}^*$. Since

$$\overline{X}^* = \begin{pmatrix} X^* & 0 \\ 0 & I \end{pmatrix}$$

we can read off X^* in $\overline{X}^*$. Thus

$$\begin{aligned} A(n) &\leq A(2^{\lceil \log\ n \rceil}) \\ &\leq 4 \cdot M(2^{\lceil \log\ n \rceil}) \\ &\leq 4 \cdot M(2n) && \text{(since } M \text{ is non-decreasing)} \\ &\leq 4 \cdot c \cdot M(n) && \text{(by assumption)} \end{aligned}$$

In either case, we have $A(n) = O(M(n))$. ∎

In Theorems 2 and 3 we established the claim that closure of a matrix and matrix product have the same order of complexity. Since matrix product is the more familiar operations we study its complexity in more detail in the subsequent sections.

5.4. Matrix Multiplication in a Ring

We all learned in school how to multiply two $n \times n$ matrices in $O(n^3)$ arithmetic operations. Surprisingly enough, the naive way of multiplying matrices is not the fastest. When we wrote this chapter, the asymptotically fastest algorithm was by Coppersmith and Winograd based on work by Strassen; their algorithm uses only $O(n^{2.39})$ arithmetic operations. In this section we describe an $O(n^{2.81})$ algorithm due to Strassen, the first algorithm which actually beat the $O(n^3)$ bound. Strassen's algorithm and all other fast multiplication algorithms for matrices do not work over semi-rings but only in the richer structure of rings.

Definition: An algebraic structure $(S, +, \cdot, 0)$ is a **ring** if

1) $(S, +, 0)$ is an abelian group, i.e.,

$$\begin{aligned} (a+b)+c &= a+(b+c) && \text{(associativity)} \\ a+b &= b+a && \text{(commutativity)} \\ a+0 &= a && \text{(neutral element)} \\ \forall a\ \exists b: \quad a+b &= 0 && \text{(inverses exist)} \end{aligned}$$

2) $(S, \cdot)$ is a semi-group, i.e.,

$$(a \cdot b) \cdot c = a \cdot (b \cdot c) \qquad \text{(associativity)}$$

3) the distributive laws hold, i.e.,

$$a \cdot (b+c) = a \cdot b + a \cdot c$$

$$(b+c) \cdot a = b \cdot a + c \cdot a$$

∎

Theorem 1. *The product of two $n \times n$ matrices over a ring can be computed in $O(n^{\log 7})$ ring operations.*

Proof: We give a recursive algorithm. Let $n = 2^k$ be a power of 2 (the general case is considered later) and let A and B be two $n \times n$ matrices. We split A and B into four $n/2 \times n/2$ matrices each.

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \qquad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

Then $C = A \cdot B$ can be written as

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

where

$$\begin{aligned} C_{11} &= A_{11} \cdot B_{11} + A_{12} \cdot B_{21}; \\ C_{12} &= A_{11} \cdot B_{12} + A_{12} \cdot B_{22}; \\ C_{21} &= A_{21} \cdot B_{11} + A_{22} \cdot B_{21}; \\ C_{22} &= A_{21} \cdot B_{12} + A_{22} \cdot B_{22}. \end{aligned}$$

It is precisely the same set of formulae that defines the product of two 2×2 matrices. Note however, that the elements of matrices A, B and C, when considered as 2×2 matrices are not ring elements but large $n/2 \times n/2$ matrices. The following lemma shows that this difference is inessential.

Lemma 1. *Let $m \in \mathbb{N}$ and let S be a ring. Then the set of $m \times m$ matrices over S forms a ring.*

Proof: Exercise 11. ∎

We still have to describe a fast method for multiplying two 2×2 matrices. What does fast mean? Note that we want to apply the algorithm recursively, i.e., the elements of the two 2×2 matrices to be multiplied are themselves large matrices. Therefore, multiplication of ring elements is much more costly than addition in the recursive application of the algorithm. It is therefore important to multiply two 2×2 matrices with a small number of multiplications of ring elements, in particular to use less than the 8 multiplications used in the school method. Strassen shows how to multiply two 2×2 matrices

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$$

with 7 multiplications and 18 additions and subtractions.

Compute

$$\begin{aligned}
m_1 &\leftarrow (a_{12} - a_{22}) \cdot (b_{21} + b_{22});\\
m_2 &\leftarrow (a_{11} + a_{22}) \cdot (b_{11} + b_{22});\\
m_3 &\leftarrow (a_{11} - a_{21}) \cdot (b_{11} + b_{12});\\
m_4 &\leftarrow (a_{11} + a_{12}) \cdot b_{22};\\
m_5 &\leftarrow a_{11} \cdot (b_{12} - b_{22});\\
m_6 &\leftarrow a_{22} \cdot (b_{21} - b_{11});\\
m_7 &\leftarrow (a_{21} + a_{22}) \cdot b_{11};
\end{aligned}$$

and then

$$\begin{aligned}
c_{11} &\leftarrow m_1 + m_2 - m_4 + m_6;\\
c_{12} &\leftarrow m_4 + m_5;\\
c_{21} &\leftarrow m_6 + m_7;\\
c_{22} &\leftarrow m_2 - m_3 + m_5 - m_7.
\end{aligned}$$

The reader can easily make sure that this algorithm actually computes the product of two 2×2 matrices. If we apply this algorithm recursively to compute C_{11}, C_{12}, C_{21} and C_{22} then the following recursion holds for $M(n)$, the number of ring operations required to multiply two $n \times n$ matrices by Strassen's algorithm

$$\begin{aligned}
M(1) &= 1 \quad \text{and}\\
M(n) &= 7M(n/2) + 18(n/2)^2
\end{aligned}$$

for n a power of 2. For $n = 2^k$ this recurrence has solution

$$\begin{aligned}
M(n) &= \sum_{i=0}^{k-1} 7^i \cdot 18 \cdot 2^{(k-i-1)2} \; + 7^k\\
&= 7^{k+1} - 6n^2 = 7n^{\log 7} - 6n^2.
\end{aligned}$$

If n is not a power of 2 we fill up matrices A and B with zeroes until we reach a power of 2 and compute the product of the padded matrices. This increases n by at most a factor of two. Thus for all n

$$M(n) \leq 7(2n)^{\log 7} = 49 \cdot n^{\log 7} = O(n^{\log 7}). \qquad \blacksquare$$

Strassen's method for multiplying matrices is asymptotically faster than the school method. Where is the cross-over point, i.e., from what n on is Strassen's algorithm superior to the school method? We confine ourselves to the case that $n = 2^k$ is a power of 2.

The school method requires n^3 multiplications and $n^3 - n^2$ additions for multiplying two $n \times n$ matrices. We want to find the smallest k_0 such that for all $k \geq k_0$

$$7^{k+1} - 6 \cdot (2^k)^2 \leq 2 \cdot (2^k)^3 - (2^k)^2.$$

A simple calculation shows that $k_0 = 10$, i.e., $n = 1024$. This is rather disappointing but it also shows how we can improve the recursive algorithm; the school method is faster than Strassen's algorithm for $n \leq 1024$. For example, Strassen's algorithm requires 25 operations for multiplying two 2×2 matrices and the school method requires only 12 operations. It is therefore senseless to use recursion all the way down to $n = 2$ in Strassen's algorithm. Where should we stop?

We pose the following question. From what point on is it cheaper to multiply directly by the school method than to use one further step of recursion? Let n be even and let A and B be $n \times n$ matrices. We split A and B each into 4 matrices of size $n/2 \times n/2$ and multiply A and B with 7 multiplications and 18 additions of $n/2 \times n/2$ matrices. The school method is used to multiply the smaller matrices. The total number of arithmetic operations used in this method is

$$7 \cdot (2(n/2)^3 - (n/2)^2) + 18(n/2)^2 = \frac{7}{4} \cdot n^3 + \frac{11}{4} \cdot n^2.$$

If the school method is used directly to multiply A and B then

$$2n^3 - n^2$$

arithmetic operations are required. Then

$$\begin{aligned} & \frac{7}{4} \cdot n^3 + \frac{11}{4} \cdot n^2 < 2n^3 - n^2 \\ \text{iff} \quad & \frac{15}{4} \cdot n^2 < \frac{1}{4} \cdot n^3 \\ \text{iff} \quad & 15 < n. \end{aligned}$$

We conclude that for even n we should use one more recursive step if $n \geq 16$. Suppose now that n is odd. We split off one column each from matrices A and B, i.e.,

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \qquad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

where A_{11}, B_{11} are $n-1 \times n-1$ matrices, A_{12}, B_{12} are column vectors of length $n-1$, A_{21}, B_{21} are row vectors of length $n-1$ and A_{22}, B_{22} are ring elements. If we compute $A \cdot B$ according to the school method then we also multiply $A_{11} \cdot B_{11}$ according to the school method. Recursion applied to product $A_{11} \cdot B_{11}$ saves operations whenever $n - 1 \geq 16$. These considerations lead to Program 3.

Theorem 2. *Procedure matmult for multiplying two $n \times n$ matrices has the following properties:*

1) *For $n < 16$ the algorithm uses the same number of arithmetic operations as the classical algorithm.*
2) *For $n \geq 16$ matmult uses strictly less arithmetical operations than the classical algorithm.*

---

procedure *matmult*(A, B, n);
co A and B are $n \times n$ matrices **oc**
if $n < 16$
then compute $A \cdot B$ according to the classical algorithm
else if n even
  then split A and B into four $n/2 \times n/2$ matrices each
    and apply the formulae given in the proof of Theorem 1;
    use *matmult* recursively to multiply $n/2 \times n/2$ matrices
  else split off one row and one column of matrices A and B
    and apply *matmult* recursively to the $n-1 \times n-1$ matrices
    obtained in this way;
    the remaining products are computed classically
  fi
fi
end.

Program 3

3) *matmult never uses more than* $4.8 \cdot n^{\log 7}$ *arithmetical operations.*

Proof: 1) and 2) are obvious from the preceding discussion. 3) remains to be proved. Let $M(n)$ be the number of arithmetical operations used by *matmult* on two $n \times n$ matrices. Then

$$\begin{array}{ll} M(n) = 2n^3 - n^2 & \text{if } n < 16; \\ M(n) = 7M(n/2) + \frac{18}{4} \cdot n^2 & \text{if } n \geq 16 \text{ and } n \text{ even}; \\ M(n) = 7M((n-1)/2) + \frac{42}{4} \cdot n^2 - 17n + \frac{15}{2} & \text{if } n \geq 16 \text{ and } n \text{ odd}. \end{array}$$

Define $\overline{M}$ for $x \in \mathbb{R}^+$ by

$$\begin{array}{ll} \overline{M}(x) = 2x^3 - x^2 & \text{if } x < 32; \\ \overline{M}(x) = 7\overline{M}(x/2) + \frac{42}{4} \cdot x^2 & \text{if } x \geq 32. \end{array}$$

Then $\overline{M}(n) \geq M(n)$ for all $n \in \mathbb{N}$. This is easily shown by direct calculation for $n < 32$ and by induction for $n \geq 32$. Furthermore

$$\overline{M}(x) = \sum_{i=0}^{k-1} 7^i \cdot \frac{42}{4} \cdot (x/2^i)^2 + 7^k \cdot [2(x/2^k)^3 - 2(x/2^k)^2]$$

for $x \geq 32$ where $k = \min\{l;\ x/2^l < 32\}$. This is easily verified by induction. With $k = \lfloor \log x \rfloor - 4 = \log x - t$ for some $t \in [4, 5)$ we obtain

$$\begin{aligned} \overline{M}(x) &\leq 7^{\log x} \cdot [13 \cdot (4/7)^t + 2 \cdot (8/7)^t] \\ &\leq 4.8 \cdot 7^{\log x} \end{aligned}$$

for all $x \geq 32$. For $x < 32$ it is easy to show directly that $\overline{M}(x) \leq 4.8 \cdot 7^{\log x}$. ∎

So far, we have only counted the number of arithmetic operations. We neither considered the additional administrative overhead required by Strassen's algorithm nor the difference in complexity of adding and multiplying ring elements. We will now sketch an analysis which takes all these facts into account. Let a be the time required to add two ring elements and let m be the time required to multiply two ring elements. Again we want to know from what n on a recursion step pays off. We neglect terms of linear order in the sequel. It takes time $n^2 \cdot a + n^2 \cdot c_1$ to add two $n \times n$ matrices on a RAM; here c_1 is the time required for storage access, index calculations and test for loop exit. Similarly, it takes $n^3 \cdot m + (n^3 - n^2) \cdot a + n^3 \cdot c_2 + n^2 \cdot c_3$ time units to multiply two $n \times n$ matrices according to the classical algorithm. Here c_2 and c_3 are the times required for storage access, index calculations and the test for loop exit in the innermost and next to innermost loop. A recursion step pays off, if

$$\begin{aligned} & 7 \cdot [(n/2)^3 \cdot m + ((n/2)^3 - (n/2)^2) \cdot a + (n/2)^3 \cdot c_2 + (n/2)^2 \cdot c_3] \\ & + 18 \cdot [(n/2)^2 \cdot a + (n/2)^2 \cdot c_1] \\ & < n^3 \cdot m + (n^3 - n^2) \cdot a + n^3 \cdot c_2 + n^2 \cdot c_3 \ , \end{aligned}$$

i.e.,

$$n > \frac{30a + 36c_1 + 6c_3}{m + a + c_2} \ .$$

It is beyond the scope of this book to determine constants c_1, c_2, c_3, a and m exactly. Realistic values are $a \leq c_1, c_2, c_3, m \leq 6a$. Then $n \geq n_0 \approx 40$, which agrees with experiments reported in the literature. We refer the reader to the literature (cf. Section 5.9) for a more detailed analysis.

The analysis of Strassen's algorithm given above may still be criticized. Suppose that we want to multiply two $n \times n$ matrices over the integers and that all entries are in the range $[0\,..\,M-1]$, i.e., all entries are numbers of at most $\log M$ bits. Let us assume further that it takes $a(k)$ resp. $m(k)$ time units to add resp. multiply k-bit numbers. The assumption here is that it takes one time unit to manipulate a single bit (cf. 5.7. for an exact definition). Then $a(k) = O(k)$ and $m(k) = O(k^2)$ by the classical methods. There are faster methods for multiplying numbers, an $O(k^{\log 3})$ algorithm is discussed in the exercises and an $O(k \cdot \log k \cdot \log\log k)$ algorithm can be found in Schönhage/Strassen (71). We use $m(k) = O(k^2)$ in the sequel. Then the following question arises. How large do the numbers become in Strassen's algorithm compared with the classical algorithm?

Using the classical algorithm we have to perform n^3 multiplications of $\log M$ bit numbers for a cost of $n^3 \cdot m(\log M)$. We then have to add numbers in the range $[0\,..\,n \cdot (M-1)^2]$. Thus the cost of the additions is bounded by $n^3 \cdot a(\log n + 2\log M)$. A more careful analysis allows us to drop the $\log n$ term in the bound on the cost of additions (Exercise 14). Thus the total cost of the classical method is bounded by $O(n^3 \cdot (\log M)^2)$.

What we can say about Strassen's method? The following simple observation is crucial. If $c_{ik} = \sum_j a_{ij} \cdot b_{jk}$ and $a_{ij}, b_{jk} \in [0\,..\,M-1]$ then $c_{ik} = \sum_j a_{ij} \cdot$

b_{jk} mod nM^2. The integers modnM^2 form a ring $\mathbb{Z}_{nM^2}$. We can therefore carry out Strassen's algorithm in the ring $\mathbb{Z}_{nM^2}$ of integers modnM^2. The cost of an addition or multiplication in that ring is certainly bounded by $m(\log n + 2\log M)$ and hence the total cost of Strassen's method is $O(n^{\log 7} \cdot (\log n + \log M)^2)$. Thus Strassen's method is asymptotically faster than the classical method not only with regard to the number of arithmetical operations but also with regard to the number of bit operations.

5.5. Boolean Matrix Multiplication and Transitive Closure

In this section we apply the results of the previous section to the boolean matrix product. Unfortunately, this is not possible directly. The boolean semi-ring $B = (\{0,1\}, \vee, \wedge, 0, 1)$ is not a ring. We have $0 \vee 1 = 1 \vee 1 = 1$, i.e., there is no additive inverse for element 1.

Let A and B be two boolean $n \times n$ matrices. We want to compute the boolean matrix product C of A and B using $\wedge$ as the multiplicative and $\vee$ as the additive operation.

Since the boolean semi-ring is not a ring we cannot apply the results of the previous section directly. A way out is to consider 0 and 1 as natural numbers and to compute the ordinary product $\hat{C}$ of A and B. Then

$$c_{ij} = \bigvee_{k=1}^{n} a_{ik} \wedge b_{kj} \quad \text{and} \quad \hat{c}_{ij} = \sum_{k=1}^{n} a_{ik} \cdot b_{kj}$$

and hence $c_{ij} = 0$ iff $\hat{c}_{ij} = 0$. We can thus directly translate matrix $\hat{C}$ into matrix C. Natural number 0 corresponds to the boolean constant 0 and natural numbers $\neq 0$ correspond to the boolean constant 1. We have

Theorem 1. *Let A and B be two boolean $n \times n$ matrices. Then the boolean matrix product of A and B can be computed in $O(n^{\log 7} \cdot (\log n)^2) = O(n^{2.82})$ bit operations.*

Proof: At the end of the previous section we have shown that the product of $(0,1)$-matrices can be computed in $O(n^{\log 7} \cdot (\log n)^2) = O(n^{2.82})$ bit operations. Recall that $\log 7 < 2.81$ and $\log n = O(n^\epsilon)$ for all $\epsilon > 0$. The discussion above shows that this is also true for the boolean matrix product. ∎

Theorem 2. *The closure of an $n \times n$ boolean matrix can be computed with $O(n^{\log 7} \cdot (\log n)^2)$ bit operations.*

Proof: Follows immediately from Theorem 1 and Theorem 3 of Section 5.3. ∎

Theorem 3. *The transitive closure of a digraph $G = (V, E)$ with $n = |V|$ can be computed in $O(n^{\log 7} \cdot (\log n)^2)$ bit operations.*

Proof: Obvious from Theorem 2 and Corollary 1 of Section 5.3. ∎

5.6. (min,+)-Product of Matrices and Least Cost Paths

We used the $(\min, +)$-semi-ring of reals to deal with least cost path problems. Let A and B be two $n \times n$ matrices with the entries in $[0\,..\,M-1] \cup \{\infty\}$. We want to compute the $(\min, +)$-semi-ring C of A and B, i.e.,

$$c_{ik} = \min_{1 \le j \le n} (a_{ij} + b_{jk}).$$

The classical method for computing C takes $O(n^3 \log M)$ bit operations. Again, the results of Section 5.4. cannot be applied directly because the $(\min, +)$-semi-ring of reals is not a ring.

An asymptotically faster algorithm is based on the following observation. If $a, b \in \mathbb{N}_0$ and $a \neq b$ then $\lim_{x \to 0}(x^a + x^b)/x^{\min(a,b)} = 1$ and hence $\min(a, b) \approx \log(x^a + x^b)/\log x$ for small x.

Lemma 1. *Let $b_1, \ldots, b_n \in \mathbb{N}_0$, let $f(x) = \sum_{k=1}^{n} x^{b_k}$ and let $a = \min(b_1, \ldots, b_n)$. Then*

$$a = \lceil -(1/m) \log f(2^{-m}) \rceil$$

for any $m > \log n$.

Proof: Let $a = b_h$. Then

$$f(2^{-m}) = \sum_{k=1}^{n} 2^{-m \cdot b_k} = 2^{-m \cdot a} \cdot (1 + \sum_{\substack{k=1 \\ k \neq h}}^{n} 2^{-m \cdot (b_k - a)})$$

$$= c \cdot 2^{-m \cdot a}$$

for some c with $1 \le c \le 1 + (n-1) = n$. Then

$$\lceil -(1/m) \log f(2^{-m}) \rceil = \lceil a - (\log c)/m \rceil = a$$

since $a \in \mathbb{N}_0$ and $0 \le (\log c)/m < 1$. ∎

Based on Lemma 1 we can use the following algorithm to compute C.

(1) Let $m = 1 + \lceil \log n \rceil$. Compute matrices $\hat{A}$ and $\hat{B}$ with

$$\hat{a}_{ij} = \begin{cases} 2^{-m \cdot a_{ij}} & \text{if } a_{ij} \neq \infty; \\ 0 & \text{if } a_{ij} = \infty \end{cases}$$

and

$$\hat{b}_{ij} = \begin{cases} 2^{-m \cdot b_{ij}} & \text{if } b_{ij} \neq \infty; \\ 0 & \text{if } b_{ij} = \infty. \end{cases}$$

(2) Compute $\hat{C} = \hat{A} \cdot \hat{B}$ by Strassen's algorithm.

(3) Compute c from $\hat{C}$ by

$$c_{ij} = \begin{cases} \infty & \text{if } \hat{c}_{ij} = 0; \\ \lceil -(1/m) \log \hat{c}_{ij} \rceil & \text{if } \hat{c}_{ij} \neq 0. \end{cases}$$

Theorem 1. *The* $(\min, +)$*-product of two matrices* A *and* B *with entries in* $[0\,..\,M-1] \cup \{\infty\}$ *can be computed in* $O(n^{\log 7})$ *arithmetical operations on real numbers or* $O(n^{\log 7} \cdot (M \log n)^2)$ *bit operations.*

Proof: m is easily computed from the binary representation of n. Then matrices $\hat{A}$ and $\hat{B}$ can be computed in $O(n^2)$ arithmetical and $O(n^2 M \log n)$ bit operations. Note that numbers $\hat{a}_{ij}$, $\hat{b}_{ij}$ have $O(M \cdot m) = O(M \log n)$ bits each. Step (2) clearly takes $O(n^{\log 7})$ arithmetical operations. It takes $O(n^{\log 7} \cdot (\log n + M \log n)^2) = O(n^{\log 7} \cdot (M \log n)^2)$ bit operations according to the discussion at the end of Section 5.4. Finally step (3) requires to take logarithms n^2 times. This can be done as follows.

Lemma 2. *Let* $x \in \mathbb{R}$, $0 < x < 1$, *and let* z *be the number of leading zeroes in the binary representation of* x. *Then* $\lceil -(1/m) \log x \rceil = 1 + \lfloor z/m \rfloor$.

Proof: Note first that

$$-\log x = z + 1 - \delta \qquad \text{for some } \delta,\ 0 \leq \delta < 1.$$

Hence

$$-(1/m) \log x = \lfloor z/m \rfloor + (z/m - \lfloor z/m \rfloor) + (1 - \delta)/m$$

Also

$$0 < (z/m - \lfloor z/m \rfloor) + (1 - \delta)/m \leq (m-1)/m + 1/m \leq 1$$

and hence

$$\lceil -(1/m) \log x \rceil = \lfloor z/m \rfloor + 1. \qquad \blacksquare$$

We conclude from Lemma 2 that step (3) takes $O(n^2)$ arithmetical operations and $O(n^2 \cdot (\log M + \log \log n)^2)$ bit operations. Note that $z \leq M \log n$. Altogether we have shown that $O(n^{\log 7})$ arithmetical operations over the reals and $O(n^{\log 7} \cdot (M \log n)^2)$ bit operations suffices. $\blacksquare$

Let us compare the classical algorithm with the new algorithm. The classical algorithm is clearly inferior with respect to arithmetical operations. The situation is not so clear with respect to the number of bit operations. The classical algorithm uses $O(n^3 \log M)$ bit operations. The new algorithm uses $O(n^{\log 7} \cdot (M \log n)^2)$ bit operations. Thus the classical algorithm is superior or at least competitive whenever M is of size $n^{3-\log 7} \approx n^{0.19}$ or larger.

In the boolean case we were able to obtain efficient algorithms for the transitive closure by applying Theorem 3 of Section 5.3. Is this also true here? The answer is "No"! Let us take a closer look at the proof of Theorem 3 of Section 5.3. In the recursive algorithm described there we have to multiply smaller matrices. In the case of $(\min, +)$-product all entries in these smaller matrices correspond to least cost paths in subgraphs of the given graph. Therefore the entries in these matrices are in the range $[0\,..\,nM]$. As above we assume that we start with a matrix with entries in $[0\,..\,M-1]$. Thus the multiplications on the way are extremely costly; their cost may be as large as $O(n^{\log 7} \cdot (nM \log n)^2)$ bit operations by Theorem 1.

This section has been inserted to give the reader a warning. It shows us that saving arithmetical operations may not always correspond to real savings in execution time, if the reduction in arithmetical operations implies a drastic increase in the size of the numbers which have to be handled by the algorithm. The additional time spent on realizing the basic arithmetic operations addition and multiplication then more than compensates the savings in the number of such operations. We conclude that it is not enough to analyze the *number* of arithmetic steps; it always has to be accompanied by an analysis of the *cost* of arithmetic steps.

V.7. A Lower Bound on the Monotone Complexity of Matrix Multiplication

In this section we will prove a lower bound on the complexity of matrix multiplication in a restricted model of computation: straight-line programs which use only monotone operations. A straight-line program is a program without loops and conditional statements. All programs which we have seen in this chapter are straight-line programs if we confine ourselves to fixed input size because for fixed input size we can eliminate all loops and procedure calls by explicit duplication of code. Monotone operations preserve the natural ordering on their domain. f is called monotone if $x_i \leq y_i$ for $1 \leq i \leq n$ implies $f(x_1, \ldots, x_n) \leq f(y_1, \ldots, y_n)$. In the arithmetical case (reals or integers) the operations addition and multiplication are monotone but subtraction is not. In the boolean case the operations AND and OR are monotone but NEGATION is not. (The natural ordering is $0 \leq 1$ in the boolean case.) We treat the boolean case first and then obtain the lower bound for other cases as a corollary.

Definition: A **straight-line program** β over set $X = \{x_1, \ldots, x_n\}$ of input variables, set $Z = \{z_1, \ldots, z_m\}$ of intermediate variables, and operations AND and OR

is a sequence $A_1, \ldots, A_m$ of assignment statements. The j-th assignment statement $A_j,\ 1 \leq j \leq m$, has the form

$$z_j \leftarrow v_{j1}\ \mathrm{op}_j\ v_{j2}$$

where $\mathrm{op}_j \in \{\mathrm{AND},\mathrm{OR}\}$, and $v_{jk} \in X$ or $v_{jk} = z_l$ for some $l < j$ and $k = 1, 2$. If the operation symbol in assignment A_j is AND res. OR then we refer to A_j as an AND-gate resp. OR-gate. Integer m is the length of the program β. ∎

The semantics of straight-line programs is straightforward. We associate with every variable $v \in X \cup Z$ of a straight-line program an n-ary boolean function $res_{\beta,v}$: $B^n \to B$ where $B = \{0, 1\}$. Let $\vec{b} = (b_1, \ldots, b_n) \in B^n$ be arbitrary.

If $v \in X$, say $v = x_i$, then $res_{\beta,v}$ is the i-th projection function, i.e.,

$$res_{\beta,x_i}(\vec{b}) = b_i.$$

If $v \in Z$, say $v = z_j$, and A_j has the form $z_j \leftarrow v_{j1}\ \mathrm{op}_j\ v_{j2}$ then

$$res_{\beta,z_j}(\vec{b}) = res_{\beta,v_{j1}}(\vec{b})\ \mathrm{op}_j\ res_{\beta,v_{j2}}(\vec{b})$$

where we took the liberty of using the same symbol op_j for the operation symbol and the operation itself.

Let F be a set of n-ary boolean functions. Then straight-line program β **computes** F if $F \subseteq \{res_{\beta,v};\ v \text{ is a variable of } \beta\}$. The **complexity** of F is the minimal length of any program β which computes it.

Example: Let β be the following program with input variables $a_{11},\ a_{21},\ b_{11},\ b_{12}$:

$z_1 \leftarrow\ a_{21} \vee b_{12}$;
$z_2 \leftarrow\ z_1 \vee b_{11}$;
$z_3 \leftarrow\ z_2 \wedge a_{11}$.

We used $\wedge$ to denote AND and $\vee$ to denote OR. (Frequently, we suppress the $\wedge$-symbol.) Then

$$res_{\beta,z_2} = a_{21} \vee b_{11} \vee b_{12} \quad \text{and} \quad res_{\beta,z_3} = a_{11}a_{21} \vee a_{11}b_{11} \vee a_{11}b_{12}.$$ ∎

Straight-line programs can be interpreted as circuits, the circuit corresponding to the program above is shown Ficure 2. The input variables correspond to the input ports of the circuit and the intermediate variables correspond to output gates.

A boolean function f is **monotone** if it can be computed by a straight-line program over operation set AND and OR. Equivalently, f is monotone if it preserves the natural ordering on B: B is ordered by $0 \leq 1$ and B^n is ordered componentwise.

We are interested in boolean matrix multiplication.

Figure 2. Circuit for example program

Definition: Let $r, p, q \in \mathbb{N}$ and let $A = (a_{ij}), B = (b_{jk})$, $1 \leq i \leq r,\ 1 \leq j \leq p,\ 1 \leq k \leq q$, be sets of boolean variables. The **(r, p, q) boolean matrix product** is the following set F of $r \cdot q$ monotone boolean functions:

$$F = \{\bigvee_{j=1}^{p} (a_{ij} \wedge b_{jk});\ 1 \leq i \leq r,\ 1 \leq k \leq q\}. \qquad \blacksquare$$

The definition of the matrix product suggests the school method for computing it. We will show that the school method is optimal.

Theorem 1.

a) *Every program (= monotone circuit) for the (r, p, q) boolean matrix product contains at least $r \cdot p \cdot q$ AND-gates and at least $r \cdot q \cdot (p-1)$ OR-gates.*

b) *The school method for boolean matrix product is the only (up to commutativity and associativity of AND and OR) monotone straight-line program which uses that number of AND- and OR-gates, i.e., the school method is the unique optimal monotone circuit for boolean matrix product.*

Proof: The proof of Theorem 1 is lengthy. We will first review some basic facts and concepts about boolean functions, then prove two theorems about the structure of optimal monotone circuits and then finally prove the lower bound on matrix multiplication.

Let $f, g : B^n \to B$ be boolean functions. We write $f \leq g$ if $f(\vec{b}) \leq g(\vec{b})$ for all $\vec{b} \in B^n$. A **monomial** is a product of input variables. Let m be a monomial. Then m is an **implicant** of f if $m \leq f$ and it is a **prime implicant** of f if $m \leq f$ and $m \leq m' \leq f$ implies $m = m'$ for all monomials m'. We use $Prim(f)$ to denote the set of prime implicants of f.

We will now state and prove two theorems on the structure of monotone circuits. We illustrate both theorems on the basis of the example given above. We assume that the three assignments given there are part of a program for the (r, p, q) boolean matrix product with $r \geq 2$, $p \geq 1$ and $q \geq 2$. We also assume that z_1 and z_3 but not z_2 are used in later statements of the program. We indicated this assumption in Figure 2 by the two wires leaving the bottom of the diagram.

Theorem 2. *Let β be a monotone circuit which computes F, let v be a variable in β, and let $Prim(res_{\beta,v}) = \{t_0, \ldots, t_k\}$. If there is no monomial t and no function $f \in F$ with $t_0 \wedge t \in Prim(f)$ then the following circuit β' also computes F: Circuit β' is obtained from β by replacing every access to variable v by an access to a new variable v' with $res_{\beta',v'} = t_1 \vee \ldots \vee t_k$.*

Remark: Circuit β' is not necessarily cheaper than circuit β because we might delete only one gate, namely v, but might have to add more than one gate to compute $t_1 \vee \ldots \vee t_k$. In our example, $a_{11}a_{21}$ is prime implicant of z_3; it is not part of any prime implicant $a_{ij}b_{jk}$ of any output function. Hence we can replace all accesses to z_3 by accesses to z_3' with $res_{\beta',z_3'} = a_{11}b_{11} \vee a_{11}b_{12}$. We can therefore delete gates z_2 and z_3 and replace them by gates which compute $a_{11}b_{11} \vee a_{11}b_{12}$. We obtain the curcuits of Figure 3.

Figure 3. Application of Theorem 2 to example circuit

Proof (of Theorem 2): For the sake of contradiction let us assume that β' does not compute F, say $f \in F$ is not computed. Then there must be a variable w with $res_{\beta,w} = f$. We conclude from the hypothesis of the theorem that $v \neq w$. Hence w exists also in circuit β' and realizes $f' = res_{\beta',w}$. By monotonicity we have $f' \leq f$ and since f is not computed by β' we even have $f' < f$. Thus there must be $\vec{b} \in B^n$ such that $f'(\vec{b}) = 0 \neq 1 = f(\vec{b})$. Since β and β' only differ in variables v and v' we

conclude further that $(t_1 \vee \ldots \vee t_k)(\vec{b}) = 0$ and $t_0(\vec{b}) = 1$. Hence, if we change the value of any variable which occurs in t_0 from 1 to 0 we also change the value of f from 1 to 0 and therefore there is a monomial t with $t_0 \wedge t \in \mathit{Prim}(f)$, contradiction. ∎

Consider variable z_3' in our example above. It has prim implicants $a_{11}b_{11}$ and $a_{11}b_{12}$. Both products have to be computed in any circuit for a matrix product but they have to be sent to different outputs. However, separating information is impossible in monotone computations as Theorem 3 shows.

Theorem 3. *Let v be a variable in a monotone circuit β which computes F. Assume further that $t \wedge t_1,\ t \wedge t_2 \in \mathit{Prim}(res_{\beta,v})$ for some monomials t, t_1, t_2 and that for all $f \in F$ we have: for all monomials s the inequalities $s \wedge t \wedge t_1 \leq f$ and $s \wedge t \wedge t_2 \leq f$ imply $s \wedge t \leq f$. Then the following circuit β' also computes F. Delete v from β and replace all accesses to v by accesses to v' with $res_{\beta',v'} = t \vee res_{\beta,v}$.*

Proof: For the sake of contradiction let us assume that β' does not compute F, say $f \in F$, is not computed. Let $f = res_{\beta,w}$ for some variable w and let $f' = res_{\beta',w}$ if $w \neq v$. If $w = v$ then let $f' = res_{\beta',v'}$. By monotonicity we have $f \leq f'$ and since f is not computed by β' we even have $f < f'$. Let $\vec{b} \in B^n$ be such that $f(\vec{b}) = 0 \neq 1 = f'(\vec{b})$. Then we must have $res_{\beta,v}(\vec{b}) = 0$ and $t(\vec{b}) = 1$.

As before, we conclude that if we change the value of any variable which occurs in t from 1 to 0 then f' changes its value from 1 to 0 and therefore we conclude that a monomial s with $s \wedge t \in \mathit{Prim}(f')$ exists.

From the structure of circuits β and β' we conclude that $s \wedge t \wedge t_1$ and $s \wedge t \wedge t_2$ are implicants of f. Hence $s \wedge t$ is an implicant of f and hence $f(\vec{b}) = 1$, contradiction! ∎

In our example we can apply Theorem 3 to z_1 and z_3'. Consider z_1 first. Let $f_{ik} = \bigvee_j (a_{ij} \wedge b_{jk})$ be an arbitrary output of boolean matrix product and let s be a monomial with $s \wedge a_{21} \leq f_{ik}$ and $s \wedge b_{12} \leq f_{ik}$. From $s \wedge a_{21} \leq f_{ik}$ we conclude that either $s \leq f_{ik}$ or $s = b_{1k}s'$ and $i = 2$. From $s \wedge b_{12} \leq f_{jk}$ we conclude that either $s \leq f_{ik}$ or $s = a_{i1}s''$ and $k = 2$. Thus either $s \leq f_{ik}$ or $s = a_{21}b_{12}s'''$ and $i = k = 2$. In either case we have $s \leq f_{ik}$. We can therefore apply Theorem 3 with $t = 1,\ t_1 = a_{21}$ and $t_2 = b_{12}$. This allows us to replace all accesses to z_1 by accesses to z_1' with $res_{\beta',z'} = 1$. Similarly, we can apply Theorem 3 to z_3' with $t = a_{11},\ t_1 = b_{11}$ and $t_2 = b_{12}$. We obtain Figure 4.

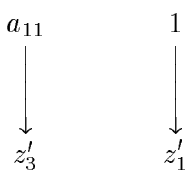

Figure 4. Application of Theorem 3 to example circuit

We can now start to really prove Theorem 1. Let β be an optimal circuit for boolean matrix product, i.e., a circuit of minimal length. We want to show that β is the school method. In the school method we have for every triple (i,j,k) an AND-gate which computes $a_{ij}b_{jk}$ and we have $p-1$ OR-gates for every pair (i,k) which sum the p products $a_{ij}b_{jk}$, $1 \leq j \leq p$. We try to locate these gates in circuit β. In order to do so we consider predicates P on boolean functions which have the property that they hold true for at least one output wire of any circuit for boolean matrix product and that they do not hold true for any input wire. Thus there must be a gate g in β such that P holds true for (the functions realized at) the output wire of g but does not hold true for any input wire of g. We denote this set of gates by $I(P)$. The gates in $I(P)$ are the gates to be located. Unfortunately, we will not be able to locate all $r \cdot p \cdot q$ AND-gates in one step. Instead we will locate only the $r \cdot q$ AND-gates which realize the products $a_{i1}b_{1k}$, then eliminate these gates by setting $a_{i1} = 1$ and $b_{1k} = 0$, and finally use induction on p.

The following notation helps to simplify the discussion. If g is a gate then we use $h\ (h_1, h_2)$ to denote the function realized by the output (left and right input) wire of g.

We will first locate the AND-gates. For $1 \leq i \leq r,\ 1 \leq k \leq q$, let P_{ik} be the following predicate on boolean functions

$$P_{ik}(h) \Leftrightarrow a_{i1}b_{1k} \leq h \text{ and } a_{i1} \not\leq h \text{ and } b_{1k} \not\leq h,$$

i.e., $a_{i1}b_{1k}$ is prime implicant of h.

Clearly, input variables of β do not satisfy P_{ik} and output f_{ik} satisfies P_{ik}. Hence $I(P)$ is not empty.

Lemma 1.

a) *If $g \in I(P_{ik})$ then g is an AND-gate and $a_{i1} \leq h_1$ and $b_{1k} \leq h_2$ (or vice versa).*

b) *If $(i_1,k_1) \neq (i_2,k_2)$ then $I(P_{i_1k_1}) \cap I(P_{i_2k_2}) = \emptyset$.*

Proof: a) Let $g \in I(P_{ik})$. First assume that g is an OR-gate. Then $h = h_1 \vee h_2$ and $P_{ik}(h)$, $\neg P_{ik}(h_1)$ and $\neg P_{ik}(h_2)$. From $a_{i1}b_{1k} \leq h$ we conclude that either $a_{i1}b_{1k} \leq h_1$ or $a_{i1}b_{1k} \leq h_2$. We may assume $a_{i1}b_{1k} \leq h_1$ w.l.o.g.. From $\neg P_{ik}(h_1)$ we conclude further that either $a_{i1} \leq h_1$ or $b_{1k} \leq h_1$ and hence $a_{i1} \leq h$ or $b_{1k} \leq h$. Thus $\neg P_{ik}(h)$, contradiction. This shows that all gates $g \in I(P_{ik})$ are AND-gates.

Let $g \in I(P_{ik})$ be an AND-gate. Then $h = h_1 \wedge h_2$ and $\neg P_{ik}(h_1)$ and $\neg P_{ik}(h_2)$. From $a_{i1}b_{1k} \leq h$ we conclude $a_{i1}b_{1k} \leq h_1$ and $a_{i1}b_{1k} \leq h_2$ and hence $a_{i1} \leq h_1$ or $b_{1k} \leq h_1$ and similarly for h_2. If $a_{i1} \leq h_1$ and $a_{i1} \leq h_2$ then $a_{i1} \leq h$, which is contradictory. Similarly, if $b_{1k} \leq h_1$ and $b_{1k} \leq h_2$ then $b_{1k} \leq h$. Thus $a_{i1} \leq h_1$ and $b_{1k} \leq h_2$ or vice versa.

b) Let $(i_1,k_1) \neq (i_2,k_2)$ and $I(P_{i_1k_1}) \cap I(P_{i_2k_2}) \neq \emptyset$, say $g \in I(P_{i_1k_1}) \cap I(P_{i_2k_2})$. Then one of the two cases applies (up to symmetrie).

Case 1: $a_{i_1 1} \leq h_1$, $a_{i_2 1} \leq h_1$
$b_{1k_1} \leq h_2$, $b_{1k_2} \leq h_2$

Case 2: $a_{i_1 1} \leq h_1$, $b_{1k_2} \leq h_1$
$a_{i_2 1} \leq h_2$, $b_{1k_1} \leq h_2$

In either case we can apply Theorem 3. If case 1 applies and $i_1 \neq i_2$ (the case that $k_1 \neq k_2$ is symmetric) then we can use Theorem 3 with $t = 1$, $t_1 = a_{i_1 1}$ and $t_2 = a_{i_2 1}$. We can therefore replace all accesses to h_1 by accesses to $h_1 \vee 1 = 1$. Thus one input of gate g becomes a constant and we can therefore eliminate gate g, which contradicts the optimality of β. If case 2 applies then we can set both inputs of g to 1 by Theorem 3 (cf. the example following Theorem 3) and therefore eliminate gate g. Thus we derived a contradiction to the minimality of β in either case. This proves part b). ∎

We turn to counting OR-gates next. Let Q_{ik} be the following predicate on boolean functions.

$$Q_{ik} \Leftrightarrow a_{i1} b_{1k} \leq h \leq A_i \vee b_{1k} \text{ and } h \not\leq b_{1k}$$

where $A_i = \bigvee_{j \neq 1} a_{ij}$. We have

Lemma 2.

a) *$I(Q_{ik}) \neq \emptyset$ if $p \geq 2$.*

b) *If $g \in I(Q_{ik})$ then g is an OR-gate and either $h_1 \leq b_{1k}$ or $h_2 \leq b_{1k}$.*

c) *The sets $I(Q_{ik})$ are pairwise disjoint.*

Proof: Similar to the proof of Lemma 1. ∎

What have we achieved at this point? For every pair (i, k) we have located an AND-gate in β which has a_{i1} as prime implicant of one of its input wires; for different pairs we identified different gates. Therefore, if we fix a_{i1} to the constant 1, $1 \leq i \leq r$, then we can eliminate $r \cdot q$ AND-gates from β. Similarly, if $p > 1$, then we have located an OR-gate g for each pair (i, k) such that either $h_1 \leq b_{1k}$ or $h_2 \leq b_{1k}$. Also, different gates were located for different pairs. Therefore, if we fix b_{1k} to the constant 0, $1 \leq k \leq q$, then we can eliminate $r \cdot q$ OR-gates from β. Finally, fixing $a_{i1} = 1$ and $b_{1k} = 0$ for $1 \leq i \leq r$ and $1 \leq k \leq q$ we transform β into a circuit for the functions

$$f_{ik} = \bigvee_{j=2}^{p} a_{ij} b_{jk},$$

i.e., into a circuit for the $(r, (p-1), q)$ boolean matrix product. If $p = 1$ then we can still eliminate $r \cdot q$ AND-gates. Part a) of Theorem 1 follows by a simple induction argument.

Part b) remains to be proved. Let β contain exactly $r \cdot p \cdot q$ AND-gates and $r \cdot (p-1) \cdot q$ OR-gates. If we apply the elimination process described above to β then we eliminate in each step *exactly* the gates in $I(P_{ik})$ and $I(Q_{ik})$, i.e., if a_{l1} is

a prime implicant of a gate g then g belongs to either $I(P_{ik})$ or $I(Q_{ik})$ for some i and k. In the latter case a_{l1} would also be prime implicant of the output of g, contradiction to $g \in I(Q_{ik})$. We conclude that a_{l1} is a prime implicant of input wires of AND-gates only, $1 \leq l \leq r$. Because of the symmetry of the boolean matrix product and because we can start the elimination process with any column of A and can also interchange the roles of A and B we conclude that variables are prime implicants of AND-gates only and that the prime implicants of inputs and outputs of OR-gates are monomials of at least two variables. We conclude further, that the inputs to a gate in $I(P_{ik})$ are *exactly* the variables a_{i1} and b_{1k}. Because of symmetry this is true for every triple (i,j,k), i.e., the $r \cdot p \cdot q$ AND-gates of β have the input pairs (a_{ij}, b_{jk}), $1 \leq i \leq r,\ 1 \leq j \leq p,\ 1 \leq k \leq q$.

Since the $r \cdot q$ output functions are disjunctions of p products $a_{ij}b_{jk}$ $(1 \leq j \leq p)$ each, the $r \cdot (p-1) \cdot q$ OR-gates are needed in order to sum up the outputs of the AND-gates. This proves part b) of Theorem 1. ∎

We want to draw one consequence from Theorem 1. Let $(S, \oplus, \odot, 0, 1)$ be an arbitrary semi-ring. We say that S has **characteristic 0** if $1 \oplus 1 \oplus \cdots \oplus 1 \neq 0$ for any number of ones to be added. We have

Theorem 4. *Let $(S, \oplus, \odot, 0, 1)$ be a semi-ring of characteristic 0. Then any straight-line program which computes the (r,p,q) matrix product of matrices over S using operations $\oplus$ and $\odot$ only contains at least $r \cdot p \cdot q$ multiplications and $r \cdot (p-1) \cdot q$ additions. Moreover, the school method is the unique optimal program.*

Proof: Let β be any straight-line program which computes the (r,p,q) matrix product of matrices over S using operations $\oplus$ and $\odot$ only. In order to prove the theorem it suffices to show that β is transformed into a monotone program β' for boolean matrix product by replacing $\oplus$ by $\vee$ and $\odot$ by $\wedge$.

This can be seen as follows. For $n \in \mathbb{N}$ let $\overline{n} = \underbrace{1 \oplus \cdots \oplus 1}_{n\text{-times}} \in S$. Then $\overline{n} \neq 0$ for all $n \in \mathbb{N}$ since S has characteristic zero, $\overline{n} + \overline{m} = \overline{n+m}$ by associativity and $\overline{n} \cdot \overline{m} = \overline{n \cdot m}$ by distributivity. Let $S_+ = \{0\} \cup \{\overline{n};\ n \in \mathbb{N}\}$. Then $(S_+, \oplus, \odot, 0, 1)$ is a semi-ring and the mapping $h : S_+ \to B$ with $h(0) = 0$ and $h(\overline{n}) = 1$ for $n \geq 1$ is a homomorphism into the boolean semi-ring $(B, \vee, \wedge, 0, 1)$. This shows that β' computes the (r,p,q) boolean matrix product. ∎

Theorem 4 applies to a large number of semi-rings, e.g., the reals under addition and multiplication $(\mathbb{R}, +, \cdot, 0, 1)$, the extended reals under minimum and addition $(\mathbb{R} \cup \{\infty\}, \min, +, \infty, 0)$, the extended reals under maximum and minimum $(\mathbb{R} \cup \{-\infty\}, \max, \min, -\infty, \infty), \ldots$.

Theorem 4 does *not* apply to the ring of integers mod p for some integer p. This ring does not have characteristic zero and in fact we can use Strassen's algorithm in this ring. (Note that $\underbrace{a + \cdots + a}_{(p-1)\text{-times}} = -a \bmod p$ and hence subtraction is reduced to addition.)

5.8. Exercises

1) Show that the following algebraic structures are closed semi-rings:

a) $(\mathbb{R}^+ \cup \{\infty\}, \max, \min, 0, \infty)$;

b) $(\mathbb{R}^+ \cup \{\infty, -\infty\}, \max, +, -\infty, 0)$, where $(-\infty) + \infty = -\infty$.

2) Compute the closure of the elements of the semi-rings of Exercise 1.

3) Show:

a) The null-element 0 of a semi-ring is uniquely defined.

b) $\bigoplus_{i \in \emptyset} a_i = 0$ in a semi-ring.

4) Let $G = (V, E)$ be a directed graph and let $c : E \to \mathbb{R}^+$ be a cost function. Define the capacity of a path as the minimum cost of any edge on the path. Show how to compute the path of maximal capacity from v to w for every pair (v, w) of points. [Hint: Use one of the semi-rings of Exercise 1.]

5) Let $G = (V, E)$ be a directed graph and let $c : E \to \mathbb{R}^+$ be a cost function. Show how to solve the all pairs maximum cost path problem.

6) Modify the all pairs least cost path algorithm so that it not only computes the cost of the least cost path but also the paths themselves.

7) Let $G = (V, E)$ be a directed graph and let $c : E \to \mathbb{R}_0^+$ be a cost function. Compute for each pair (v, w) of vertices the cost of the k least cost paths from v to w. Find an adequate closed semi-ring.

8) Show that the algebraic structure $(M_n, +, \cdot, 0, I)$ of $n \times n$ matrices over a closed semi-ring is a closed semi-ring.

9) Verify formally all identities which were used in the proof of Theorem 3 of Section 5.3.

10) Show that $M(n)$, the cost of multiplying two $n \times n$ matrices, is nondecreasing.

11) Show that the set of $n \times n$ matrices over a ring forms a ring.

12) Let $G = (V, E)$ be an acyclic directed graph and let $c : E \to S$ be a labelling of the edges of G with elements of a semi-ring S. Show how to solve the all pair path problem using $O(n \cdot e)$ semi-ring operations.

13) Let a and b be two n-bit integers. Let $k = \lceil n/2 \rceil$ and write $a = a_1 \cdot 2^k + a_2$ and $b = b_1 \cdot 2^k + b_2$ where a_1, a_2, b_1 and b_2 are k-bit integers. Then

$$a \cdot b = a_1 b_1 2^{2k} + (a_1 b_2 + a_2 b_1) 2^k + a_2 b_2.$$

Let

$$\begin{aligned} m_1 &\leftarrow (a_1 - a_2) \cdot (b_1 - b_2); \\ m_2 &\leftarrow a_1 b_1; \\ m_3 &\leftarrow a_2 b_2. \end{aligned}$$

Then $a_1 b_2 + a_2 b_1 = -m_1 + m_2 + m_3$, i.e., we can compute $a \cdot b$ using *three* multiplications of k-bit integers where $k = \lceil n/2 \rceil$. Show that this observation yields an algorithm which multiplies n-bit integers using $O(n^{\log 3})$ bit operations.

14) Let $x_1, \ldots, x_n$ be integers in the range $[0 \,..\, M-1]$. Show how to compute $x_1 + \cdots + x_n$ using $O(n \log M)$ bit operations. [Hint: Add the x_i's in the form of a binary tree and observe that the binary representation of a sum of 2^k x_i's has length $k + \log M$.]

15) State and prove theorems similar to Theorems 1 and 4 of Section 5.7 for transitive closure instead of matrix multiplication.

16) Let T be a binary tree with n leaves. For each node v let $w(v)$ be the number of leaves in the subtree with root v and let $bin(w(v))$ be the binary representation of $w(v)$. Show that the labelling $\{bin(w(v));\ v \text{ a node of } T\}$ can be computed in $O(n)$ bit operations. Note that the total length of all labels might be $O(n \log n)$ and therefore only an implicit representation of the labelling can be computed. The representation should be such that given v its label $bin(w(v))$ can be read off in time $O(|bin(w(v))|)$.

V.9. Bibliographic Notes

The algorithm for solving general path problems goes back to Kleene (56) who found it in connection with finite automata and regular expressions. The algebraic viewpoint was introduced by Aho/Hopcroft/Ullman (74) and later refined by Fletcher (80). The algorithms for the special cases of Section 5.2 are due to Roy (59), Warshall (62) and Floyd (62). The connection between matrix multiplication and transitive closure was established by Munro (71), Furman (70) and Fischer/Meyer (71). The fast matrix multiplication algorithm of Section 5.4 is due to Strassen (69); an $O(n^{2.39})$ algorithm was recently found by Coppersmith/Winograd (86) extending work of Strassen (86). The papers by Cohen/Roth (76) and Spieß (74) discuss the problem of implementing the fast matrix multiplication algorithm. Section 5.5 follows Fischer/Meyer (71), Section 5.6 follows Romani (80), and Section 5.7 discusses the papers of Paterson (75) and Mehlhorn/Galil (76). The $O(n^{\log 3})$ algorithm for integer multiplication (Exercise 13) is due to Karazuba/Offman (62); an $O(n \log n \cdot \log\log n)$ algorithm can be found in Schönhage/Strassen (71).