math

Report No. UIUCDCS-R-78-930

UILU-ENG 78 1721

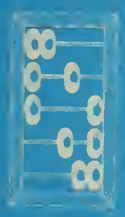
STUDIES IN CONJUGATION: HUFFMAN TREE CONSTRUCTION, NONLINEAR RECURRENCES, AND PERMUTATION NETWORKS

by

Douglass Stott Parker, Jr.

July 1978

NSF-OCA-MCS73-07980-000035



DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN · URBANA, ILLINOIS

Report No. UIUCDCS-R-78-930

STUDIES IN CONJUGATION: HUFFMAN TREE CONSTRUCTION, NONLINEAR RECURRENCES, AND PERMUTATION NETWORKS

by

Douglass Stott Parker, Jr.

July 1978

Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

*This work was supported in part by the National Science Foundation under Grant No. US NSF MCS73-07980 and was submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science, July 1978.

Dedicated to my parents

Acknowledgement

I would like to acknowledge the impact on my development of my advisor, Dr. David J. Kuck; throughout my stay at the University of Illinois he has provided me with unerringly good advice, constant support, and an extremely pleasant environment in which to work.

I would also like to thank Drs. A.H. Sameh, M.B. Pursley, D.H. Lawrie, C.L. Liu, S. Lang and C.K. Wong for discussions that have led to material improvements in my understanding of numerous points brought up by the problems discussed here. Many thanks are also due to Mme. Vivian Alsip, for her excellent typing of most of Chapter 4; and to the IBM Corporation, which supported me during all of the past year.

Table of Contents

1

Studies in Conjugation: Introductory remarks

Studies in Conjugation

A common theme in problem solving is the reduction of the problem one is faced with to another to which one already knows the solution. As a generic example, suppose that we are asked to evaluate the function

$$F: X \to X.$$

It is clear that this evaluation is equivalent to the evaluation of the conjugated function

$$G = \phi \circ F \circ \phi^{-1}: \phi(X) \to \phi(X)$$

where ϕ is some invertible "change of variables" and $\circ$ denotes functional composition, provided we can evaluate $\phi(x)$ and $\phi^{-1}(x)$. For then to compute $y = F(x)$ we can make the three step computation

$$a \leftarrow \phi(x)$$

$$b \leftarrow G(a)$$

$$y \leftarrow \phi^{-1}(b).$$

This approach is <u>useful</u> (practically speaking) if we can find a ϕ such that computation of ϕ, G, and ϕ^{-1} all together is simpler than direct computation of F itself, and we can say in this case that we have reduced the problem of evaluating F to the easier three-step evaluation.

Conjugation, as just illustrated, is a natural way to reduce one problem to another. Viewed abstractly, one is mapping a problem on one domain into another domain where the problem's structure is simpler, solving the problem there, and then mapping the computed solution back to the original solution space. The word "simpler" used here and in the paragraph above can entail many things -- computational ease of solution (e.g., reduced time or space complexity, or both, of an algorithm

which solves the problem), conceptual simplicity (intellectual manageability of the problem or brevity of an algorithm for solving the problem), and so on -- but the notion we wish to convey is that some cost criterion is being reduced through the use of conjugation.

Conjugation is, of course, not a new technique. It has been used effectively in computing convolutions (as well as the other diverse applications of the fast Fourier transform), reduction of NP-complete problems to one another, and mapping algorithms onto specific machine architectures, to name just a few areas. This dissertation simply points out that conjugation plays an important role in the analysis of three problems discussed here:

Construction of trees using the Huffman algorithm

Rapid evaluation of nonlinear recurrences

Design and assessment of permutation networks.

In each case conjugation reduces the computational or intellectual complexity of the problem to some degree, providing new insights about the problem structure and suggesting new ways old algorithms may be improved upon.

The Huffman algorithm is a well-known method for constructing optimal binary (or r-ary) trees on a given set of terminal nodes. In the type of tree construction considered here each node has some associated weight, and these weights are combined as the construction continues to form new weights for the interna. nodes of the tree; the Huffman algorithm

merely specifies which nodes are to be combined at any step of the construction process. Although Huffman's algorithm is extremely simple, it has important applications in many fields of computer science, from data compression to roundoff minimization to leaky pipeline detection. Here a new formulation of weighted tree construction is presented in a way that leads naturally to a solution of the following question: for exactly which weight combination functions does the Huffman algorithm produce optimal trees under exactly which tree cost criteria? It is shown that quasilinear combination functions (functions that are conjugate to linear functions) produce optimal trees in conjunction with the Huffman algorithm under very broad classes of cost criteria. In addition the known results about Huffman tree construction and related concepts from information theory and the theory of convex functions are tied together in a nice way, and some interesting applications are given.

The problem of evaluating nonlinear recurrences rapidly is a difficult one, but has important applications in the design of algorithms for parallel machines. Generally speaking, we are interested in transforming the m^{th}-order recurrence

$$x_k = F(x_{k-1}, x_{k-2}, \ldots, x_{k-m}) \qquad (1 \leq k \leq n)$$

to a simpler problem (say, a linear recurrence) which may be solved quickly in parallel. Until recently the only results for this problem were negative, but it is shown here how these negative results may be bypassed. In fact, the first-order, constant-coefficient case of this problem can always be solved on certain domains -- and the theoretical background and a semi-automatizable methodology for the solution of this case are

outlined and illustrated with a number of examples. Also, some techniques for reducing the higher-order, non-constant-coefficient recurrence to a system of linear recurrences are presented.

The section on permutation networks may be divided neatly in two sections. The first part analyzes the control complexity of the Rearrangeable Switching Network (RSN). This network achieves a significant savings in gate complexity over a crossbar through the use of a conjugated Shuffle/Unshuffle interconnection pattern, but suffers in that the resulting network is much harder to set to realize a desired permutation (to control). New control algorithms for the RSN are given here, and it is shown that if RSN's are recursively constructed in an intelligent way, then the switches may be controlled much more rapidly than was known before. Unfortunately, the results are asymptotic in the number of switch inputs, and are not good enough to be practically worthwhile.

The second part of this section analyzes a number of properties of Shuffle/Exchange networks. Once the proper machinery is established it is shown that Lawrie's inverse Omega network, Pease's indirect binary n-cube, and a network related to the RSN have identical switching capabilities. This result leads to a number of insights on the structure of the fast Fourier transform (FFT) algorithm, as well as a better general understanding of these switches: for example, it is shown that the Omega network is conjugate to the inverse Omega network under the bit reversal permutation. The inherent permuting power of the networks when used iteratively is then probed, leading to some non-intuitive results which have implications on the optimal control of Shuffle/Exchange-type

networks for realizing permutations and broadcast connections.

Further work concerning the methodological use of conjugation as a technique in problem solving is certainly in order, but will not be addressed here. It is already clear that good upper bounds on the complexity of a problem (i.e., good algorithms for solving it) may be derived by considering several conjugated forms of the problem. It would be interesting to study the "simplest" conjugate form of a problem; of course such a form exists, but if it could be exhibited then one could claim in a mildly-restricted sense that he had found an optimal algorithm for solving the problem. A great advance would be to determine which problem criteria guarantee us that the best possible form in which to solve the problem is a conjugate form, for then lower bounds on the problem's complexity could be studied as well. This would be true even if the simplest conjugate form could not be found explicitly.

Nuns...!

Nuns fret not at their convent's narrow room;
And hermits are contented with their cells;
And students with their pensive citadels;
Maids at the wheel, the weaver at his loom,
Sit blithe and happy; bees that soar for bloom,
High as the highest Peak of Furness-fells,
Will murmur by the hour in foxglove bells:
In truth the prison, into which we doom
Ourselves, no prison is: and hence for me,
In sundry moods, 'twas pastime to be bound
Within the Sonnet's scanty plot of ground;
Pleased if some Souls (for such there needs must be)
Who have felt the weight of too much liberty,
Should find brief solace there, as I have found.

-- W. Wordsworth, 1807

2
Analysis of the Huffman Tree Construction Algorithm

I. Introduction

Although Huffman's algorithm was first presented in 1952, and was developed then for a problem in discrete coding [Huf 52], it is still undergoing a considerable amount of research as more and more applications for it are uncovered in various fields. In the last year alone, Itai [Itai 76], van Leeuwen [vanL 76], Glassey and Karp [GK 76], and Golumbic [Gol 76] have presented new perspectives on how the algorithm works and how it or related algorithms can be employed in new ways. Until the present, however, all research has concentrated on two variations of the algorithm which respectively minimize total weighted path length, and measures akin to tree height, of the constructed tree. Applications for weighted path length minimization include (1) construction of optimal search trees [Zim 59],[HT 71],[Itai 76], (2) merging of lists [FB 72],[Liu 76], (3) minimization of absolute error in sums [Cap 75] and relative error in products [Sam 75], (4) text file compression [Rub 76], (5) optimal checking for leaky pipelines and water pollution [GK 76], and of course (6) construction of minimum redundancy codes [Huf 52]. Applications for tree height minimization include (1) optimal execution time for fanning-in data (in limited task-scheduling systems, and in arithmetic/Boolean sum- or product accumulation (e.g., in dot-products, matrix multiplication), etc.) and related problems related to speed in parallel processing [Gol 76], and (2) minimization of error bounds in parse trees of sums [Sam 75]. And this is by no means a complete list.

Our interest in the algorithm comes mainly from its import in compiling.

Not only does the algorithm build optimal trees with respect to execution time, space usage, and roundoff error for many classes of limited expressions, it does so in near linear time. If N is the number of leaves in the tree to be constructed, Huffman's algorithm can be implemented in time O(N logN) when a priority queue is used. Moreover, van Leeuwen has shown that this time bound can be reduced to O(N) if the leaf weights are in sorted order [vanL 76]. (This would suggest that the complexity of the algorithm is lower bounded by O(N log N), since sorting is at least that difficult. However an O(N log N) optimal parsing algorithm though not linear, is still respectable.) In our opinion the algorithm has great potential in the development of future compiling algorithms, as well as other areas of computer science.

This paper addresses and solves in part the following problem. The two variations of the Huffman algorithm mentioned above are based on the same construction process, but use different tree cost functions and node-merging methods. (Specifically, the weighted path-length variation produces internal nodes having weights equal to the sum of the weights of its sons, while the tree-height algorithm uses the maximum of the son weights plus some nonnegative constant. This will all be discussed in greater detail below.). First, it is not clear why these two apparently unrelated methods both produce optimal trees. Second, from the point of view of compiling it would be nice if we could use yet other methods

to construct trees optimal with respect to some other cost measure besides tree height and path length. For example, suppose we wish to construct parse trees for parallel evaluation of products of arithmetic expressions, optimal with respect to some measure of both roundoff and space used. Since error bounds in this case correspond to path-length, and execution time to tree height, an optimal parse tree cannot be constructed using the Huffman algorithm unless a node-merging method more complicated than the two above is used. This problem raises the following question: for exactly which methods will the Huffman algorithm produce optimal trees under exactly which cost? We will show a class of methods exists, encompassing the two standard methods above, which produces optimal trees with the Huffman algorithm under corresponding classes of tree cost functions and ties together in a nice way some results from information theory and the theory of convex functions.

II. Basic Machinery for Huffman Tree Construction

In this section we define the notation to be used for the rest of the paper. The exposition here is not really introductory and readers seeking more background are referred to [Knu 68] or [Eve 73]. For the time being we confine ourselves to binary tree construction until the essential results are established. The extension to r-ary trees is then straightforward.

In the binary tree construction problem one is given a set of n+1 leaves having corresponding weights $\{ w_1, w_2, \ldots, w_{n+1} \}$. Although in some problems a particular ordering is to be enforced on the leaves (e.g., [HT 71]) we drop these considerations and presume in this paper that the final order of the leaves in the constructed tree makes no difference. Furthermore the weights need not be normalized so that their sum comes out to be unity or anything like that; we require only that they be nonnegative and, for convenience, sorted by index:

$$0 \leq w_1 \leq w_2 \leq \cdots \leq w_{n+1} .$$

Construction of a (full) binary tree on these leaves is then effected by n merge operations of pairs of available nodes. Each of the nodes in the pair is marked unavailable and their father (the result of the merge) is marked available, having as his weight some function of the pair's weights. Each of the leaves is initially marked available, of course. Note that n merges are necessary and sufficient since all full binary trees on n+1 leaves have n internal nodes.

Simple examples of tree construction are given in Fig. 1 a&b. In Fig.1a the weight combination function used is the sum of the son weights; in Fig.1b it is their maximum plus 3.14 .

(a) weight(root) = weight(left son) + weight(right son)

(b) weight(root) = max(weight(left son), weight(right son)) + 3.14

Figure 1. Tree Construction

Note that each internal node defines the root of a full binary subtree of the constructed tree, so tree construction can be defined inductively in terms of forests (collections of trees) in the obvious way: the construction begins with a forest of n+1 one-node trees and repeatedly reduces the number of trees by 1 via merge operations until there is only one tree left.

With this in mind we adopt the following notation:

w_j -- j^{th} smallest leaf weight (i.e., w_1 is smallest, w_{n+1} largest)

ℓ_j -- path length (distance from the root) of the j^{th} leaf

W_i -- i^{th} smallest internal node weight

With each of these the name of the tree or forest in question will be added in parentheses whenever it is not clear from context which tree or forest is meant. Thus $W_i(T)$ would be the i^{th} smallest internal node weight

in the tree T, and $\ell_j(\mathcal{F})$ would be the current path length of w_j to the root of the tree containing it in the forest $\mathcal{F}$. For example, if we let T_1 and T_2 be the trees in Fig.1a and 1b respectively, then

$$w_1(T_1) = w_1(T_2) = 1$$
$$w_2(T_1) = w_2(T_2) = 2$$
$$w_3(T_1) = w_3(T_2) = 3$$
$$w_4(T_1) = w_4(T_2) = 4$$

$$(\ell_1, \ell_2, \ell_3, \ell_4)(T_1) = (3, 3, 2, 1)$$

$$(\ell_1, \ell_2, \ell_3, \ell_4)(T_2) = (2, 2, 2, 2)$$

and

$$(W_1, W_2, W_3)(T_1) = (3, 6, 10)$$

$$(W_1, W_2, W_3)(T_2) = (5.14, 7.14, 10.28)\ .$$

Finally, if we denote by R_+ the nonnegative reals, let us define the weight combination function $F: R_+^2 \to R_+$ to be the symmetric function used to produce the weight of internal nodes generated by a merge operation (cf. Fig.2), and the n-internal node tree cost function $G: R_+^n \to R$ to be a function on the weights of all the internal nodes of the tree:

$$\text{Cost}(T) = G(W_1(T), W_2(T), \ldots, W_n(T))\ .$$

Note that if such a tree cost is to be generally useful, it should be extensible to arbitrary numbers of arguments and not dependent on some fixed value of n.

Note that $F(x,y) = F(y,x)$

(order of leaves in tree is immaterial)

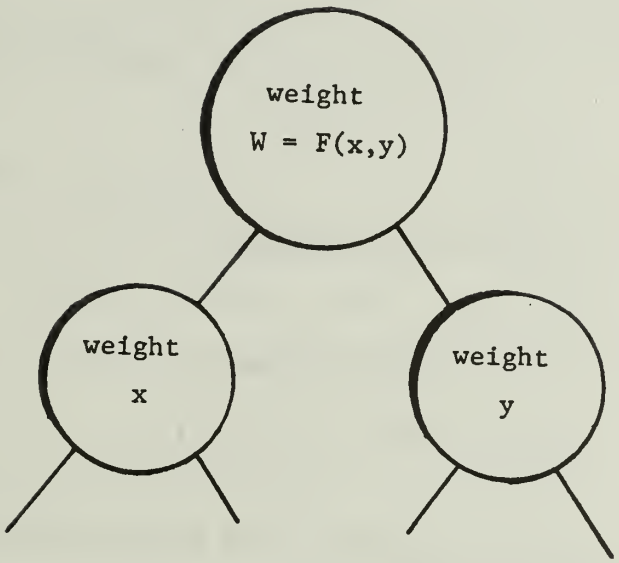


Fig. 2 Weight combination function F(x,y)

Huffman's algorithm for binary tree construction is now simple to state: To build the Huffman tree, merge at each step the two available nodes of smallest weight (with ties resolved arbitrarily). Now if

$$F(x,y) = x + y \qquad \text{and} \qquad G = \text{sum}$$

then it is not hard to show that the cost of any tree T in this system is

$$\sum_{1 \le j \le n+1} w_j(T)\, \ell_j(T)$$

which is called the weighted path length of T. Also, if

$$F(x,y) = \max(x,y) + c \quad (c \ge 0) \qquad \text{and} \qquad G = \max$$

then the cost of any tree T in this system is

$$\max_{1 \le j \le n+1} \left(w_j(T) + c \cdot \ell_j(T) \right)$$

and we call this a tree-height measure of T because when c=1 and $w_j=0$ for j=1,...,n+1 this cost is exactly the height of T (although it otherwise has nothing directly to do with tree height).

The importance of Huffman's algorithm is that it produces, in time O(n log n) or less, optimal trees in both of these systems. Proof of the optimality in the weighted path length system (the one originally considered by Huffman) can be found in a paper by Zimmerman [Zim 59]. To our knowledge a proof of the optimality of Huffman's algorithm in the tree-height system has never been published, possibly because Zimmerman's proof mutatis mutandis will work for it as well, possibly because the optimality is intuitively clearer. Examples of the construction in both systems has already been given in Figure 1. In both cases the trees illustrated are the unique optimal-cost trees; note that although they have identical initial weights their structures are entirely different.

III. General Characterization of Huffman Tree Construction

We begin this section with a result from a recent paper by Glassey and Karp [GK 76], and show how it may be extended in a natural way to characterize the weight structure obtained in trees constructed with the Huffman algorithm in general.

Definition A weight sequence $\mathbf{a}$ is a set of nonnegative numbers $[a_1, a_2, \ldots, a_m]$ such that $a_1 \leq a_2 \leq a_3 \leq \ldots \leq a_m$.

We define a partial order on weight sequences of equal length as follows.

Definition Given two weight sequences $\mathbf{a} = [a_1, a_2, \ldots, a_m]$ and $\mathbf{b} = [b_1, b_2, \ldots, b_m]$, we write

$$\mathbf{a} \preceq \mathbf{b}$$

if $\sum_{i=1}^{k} a_i \leq \sum_{i=1}^{k} b_i$ holds for all k, $1 \leq k \leq m$.

Theorem 1 (Glassey & Karp) Let $\mathbf{W(S)} = [W_1(S), W_2(S), \ldots, W_n(S)]$ be the weight sequence for the internal nodes in a tree constructed by the binary Huffman algorithm in the weighted path-length system, and let $\mathbf{W(T)} = [W_1(T), W_2(T), \ldots, W_n(T)]$ be the weight sequence for the internal nodes of any other tree on the same leaf weights. Then $\mathbf{W(S) \preceq W(T)}$.

Glassey and Karp actually prove the theorem for the general case of r-ary tree construction, where r may be greater than 2 and the trees need not be full. The proof, which may be found on pp. 371-373 of

[GK 76] establishes by induction on k that $\sum_{1}^{k} W_i(S) \leq \sum_{1}^{k} W_i(T)$, for $1 \leq k \leq m$. The theorem is a sharpening of the earlier result by Hu and Tucker that "Huffman's algorithm gives an optimal m-sum forest" in the weighted path-length system ([HT 71], p.518). In any case it is an important characterization of the Huffman algorithm and will give rise to most of the results in this paper.

We require a few definitions, including the usual ones for strict monotonicity and convexity (a functions $\phi:U \to R$ is convex if U is a convex subset of R and for all $x,y \in U$, $t \in [0,1]$, $\phi(tx+(1-t)y) \leq t\cdot\phi(x) + (1-t)\cdot\phi(y)$; ϕ is concave if $-\phi$ is convex). We say also that $\phi:U \to R$ is positive if $\phi(x) \geq 0$ for all x in U, negative if $-\phi$ is positive, and sign-consistent if ϕ is positive or negative.

Theorem 2 Let $\mathbf{a}$ and $\mathbf{b}$ be two weight sequences of length m such that $\mathbf{a} \trianglelefteq \mathbf{b}$. If ϕ is any concave, strictly increasing function and we define $\phi(\mathbf{a})$ to be the weight sequence $[\phi(a_1),\ldots,\phi(a_m)]$ and similarly for $\phi(\mathbf{b})$, then

$$\phi(\mathbf{a}) \trianglelefteq \phi(\mathbf{b}) \qquad \text{i.e.,} \quad \sum_{i=1}^{n} \phi(a_i) \leq \sum_{i=1}^{n} \phi(b_i) \quad \text{for } 1 \leq n \leq m.$$

Proof This result is typical in the theory of convex functions. An elegant proof can be adapted from that of Fuchs [Fuc 47], presented also in [Mit 70], for the analogous case where ϕ is convex. It is instructive to note that our partial order $\mathbf{a} \trianglelefteq \mathbf{b}$ on weight sequences is equivalent to the "majorization" relation $\mathbf{a} \succ \mathbf{b}$ of [HLP 34] (which appears widely in the literature) if and only if $\sum_{1}^{m} a_i = \sum_{1}^{m} b_i$.

Define $D_i = \left[\frac{\phi(a_i)-\phi(b_i)}{a_i - b_i}\right]$ for i=1, ..., m,

where we assume without loss of generality that $a_i \neq b_i$ for any i. (If $a_i=b_i$ we can delete both from the weight sequences a and b without disturbing the inequality we wish to prove). Since ϕ is increasing we have $D_i > 0$ for $1 \leq i \leq m$; also since $\mathbf{a} \leq \mathbf{b}$ and since ϕ is concave we get $D_i \geq D_{i+1}$. Set $A_k = \sum_{j=1}^{k} a_j$ and $B_k = \sum_{j=1}^{k} b_j$ for $1 \leq k \leq m$.

Then $\mathbf{a} \leq \mathbf{b}$ implies $A_k \leq B_k$ for all k, or $(A_k-B_k) \leq 0$.

Therefore for $1 \leq n \leq m$,

$$\sum_{k=1}^{n-1} (A_k-B_k)(D_k-D_{k+1}) + (A_n-B_n)D_n \leq 0$$

$$\sum_{k=1}^{n-1} A_k (D_k-D_{k+1}) + A_nD_n \leq \sum_{k=1}^{n-1} B_k(D_k-D_{k+1}) + B_nD_n$$

$$\sum_{i=1}^{n} (A_i-A_{i-1}) D_i \leq \sum_{i=1}^{n} (B_i-B_{i-1}) D_i$$

$$\sum_{i=1}^{n} a_i D_i \leq \sum_{i=1}^{n} b_i D_i \quad \Longrightarrow \quad \sum_{i=1}^{n} (a_i-b_i) D_i \leq 0$$

$$\sum_{i=1}^{n} \phi(a_i) - \phi(b_i) \leq 0$$

$$\sum_{i=1}^{n} \phi(a_i) \leq \sum_{i=1}^{n} \phi(b_i) . \qquad \text{QED}$$

With the above in mind we can make our extension of the Huffman tree construction process. For the rest of this paper we assume that our weight combination function $F(x,y)$ has the quasilinear form

$$F(x,y) = \phi^{-1}(\lambda\phi(x) + \lambda\phi(y))$$

where λ is some positive constant and ϕ is a continuous, strictly monotone function. We restrict the domain of definition of ϕ to some interval U of R_+, which we call the weight space, and require that $F: U^2 \to U$ so that F produces a weight when given two weights.

The conjugate linear class of functions this generates has a number of interesting properties: each such F is increasing since ϕ is monotone. Each F is symmetric in its variables and can be extended naturally to functions of more than two arguments. (This latter property will be useful at the end of this section when we consider the generalization of binary to r-ary tree construction.) Moreover when $\lambda = 1$ F is also associative, i.e.,

$$\begin{aligned} F(F(u,v), F(x,y)) &= F(u, F(v, F(x,y))) \\ &= F(F(x,v), F(u,y)) \\ &= \phi^{-1}(\phi(u) + \phi(v) + \phi(x) + \phi(y)). \end{aligned}$$

Also note that when $\lambda = 1$ and $\phi(x) = x$ we obtain

$$F(x,y) = x + y$$

-- the weight merging function for the weighted path-length system--, and when $\lambda = \exp(pc)$ $[c \geq 0]$ and $\phi(x) = \exp(px)$, then

$$\lim_{p \to \infty} F(x,y) = \max(x,y) + c$$

-- the function for the tree-height system. Thus this class of weight merging functions F is broad enough, in the limit at least, to encompass

the two known Huffman-optimal ones. The purpose of this paper is to show first, what the Huffman algorithm produces with these weight merging functions; second, which conditions are needed for this produce to be optimal; and third, why all this is useful.

One assumption we can make immediately is that the strictly monotone function ϕ is strictly <u>increasing</u> since F is invariant of changes to the sign of ϕ. For this reason we will frequently make statements below requiring ϕ to be increasing; if ϕ were actually taken to be decreasing then the statement in question would hold for $-\phi$. More restrictions must be made on ϕ and λ before we can prove the resulting function F will be useful to us; these restrictions are mainly embodied in the following lemma. First we know we must have $F: U^2 \to U$. Also, we need an analogue of the fact used in the proof of Theorem 1 that $F(x,y)=x+y$ is "non-shrinking" (in the sense that $F(x,y) \geq \max(x,y)$) which guarantees that the k smallest internal node weights of any constructed tree T comprise the weights of some subforest of T. We satisfy both these restrictions on F in the following way:

<u>Lemma 1</u> Let $\phi: U \to R$ be a strictly increasing function and λ be a positive constant. If $F(x,y) = \phi^{-1}(\lambda\phi(x)+\lambda\phi(y))$ is to satisfy $F:U^2 \to U$ and either $F(x,y) \leq \min(x,y)\ \forall_{x,y\in U}$ or $F(x,y) \geq \max(x,y)\ \forall_{x,y\in U}$, then we must have $\lambda \geq 1$, and ϕ must be sign-consistent on U.
Under these circumstances the quasilinear function F satisfies

$F(x,y) \leq \min(x,y)\ \forall_{x,y\in U}$ if ϕ is negative (increasing),

$F(x,y) \geq \max(x,y)\ \forall_{x,y\in U}$ if ϕ is positive (increasing).

Proof Since ϕ is increasing we have

$$F(x,y) \geq \max(x,y) \quad \text{iff} \quad \lambda\phi(x) + \lambda\phi(y) \geq \phi(x) \quad \text{for all } x \geq y \text{ in } U,$$

$$F(x,y) \leq \min(x,y) \quad \text{iff} \quad \lambda\phi(x) + \lambda\phi(y) \leq \phi(x) \quad \text{for all } x \leq y \text{ in } U.$$

Neither of these can be true if ϕ is <u>not</u> sign-consistent, for then the connectivity of the interval U and the continuity of ϕ imply a neighborhood of zero would exist in $\phi(U)$, and for example we could contradict the first inequality above by selecting $\phi(x) > 0$, $\phi(y) = -\phi(x)$, giving $\lambda \cdot 0 = 0 \geq \phi(x)$. So we conclude ϕ must be sign-consistent, and find the "shrinking" condition on F is satisfied when

$$\lambda \geq \lambda_1 \overset{\text{def}}{=\!=\!=} \sup_{x,y}\left(\frac{\phi(x)}{\phi(x)+\phi(y)}\right) \quad \text{and} \begin{cases} \phi \text{ is negative} & [F(x,y) \leq \min(x,y)] \\ \phi \text{ is positive} & [F(x,y) \geq \max(x,y)] \end{cases}$$

$$\lambda \leq \lambda_0 \overset{\text{def}}{=\!=\!=} \inf_{x,y}\left(\frac{\phi(x)}{\phi(x)+\phi(y)}\right) \quad \text{and} \begin{cases} \phi \text{ is negative} & [F(x,y) \geq \max(x,y)] \\ \phi \text{ is positive} & [F(x,y) \leq \min(x,y)] \end{cases}$$

We now show that the additional condition $\lambda \geq 1$ is necessitated by the requirement that $F: U^2 \to U$ by considering what happens to the above inequalities for all nonzero λ ($\lambda = 0$ is uninteresting, giving F = constant):

(1) Assume $\lambda > 1/2$. Then since $F: U^2 \to U$ we have $\lambda(\phi(U)+\phi(U)) \subseteq \phi(U)$, implying $\phi(U)$ must be unbounded, so we have $\lambda_0 = 0$ and $\lambda_1 = 1$. The only nontrivial condition we can satisfy is $\lambda \geq \lambda_1 = 1$, giving as stated $F(x,y) \geq \max(x,y)$ if ϕ is positive,
$F(x,y) \leq \min(x,y)$ if ϕ is negative.

(2) Assume $\lambda = 1/2$. Since we must always have $\lambda_0 < 1/2$ and $1/2 < \lambda_1$, there is no way to satisfy either $\lambda \leq \lambda_0$ or $\lambda \geq \lambda_1$.

(3) Assume $0 < \lambda < 1/2$. Then because $\lambda(\phi(U)+\phi(U)) \subseteq \phi(U)$ zero is a limit point of $\phi(U)$, so we get $\lambda_0 = 0$ and $\lambda_1 > 1/2$. Thus there is again no nontrivial way to satisfy either $\lambda \leq \lambda_0$ or $\lambda \geq \lambda_1$.

As an interesting sidelight, note that when ϕ is positive increasing, the tree constructed by the Huffman algorithm (for any positive λ) on the leaf weights $\{w_1, \ldots, w_{n+1}\}$ is topologically isomorphic to the tree that would be built by the Huffman algorithm with

$$F(x,y) = \lambda(x + y)$$

on the leaf weights $\{\phi(w_1), \ldots, \phi(w_{n+1})\}$, although the actual values of the internal node weights would be different unless $\phi(x)=x$. If ϕ is positive decreasing, by contrast, the tree constructed by the Huffman algorithm on $\{w_1, \ldots, w_{n+1}\}$ is topologically isomorphic to that which would be built by the anti-Huffman algorithm (the tree construction procedure in which the two nodes of greatest weight are merged at each step) with $F(x,y) = \lambda(x+y)$ on the leaf weights $\{\phi(w_1), \ldots, \phi(w_{n+1})\}$. This all follows from the "order-preserving" properties of monotone functions. It should be pointed out that when ϕ is positive decreasing and $\lambda \geq 1$ the Huffman algorithm could always produce the following tree, because then $F(x,y) \leq \min(x,y)$ and the smallest weight is always selected:

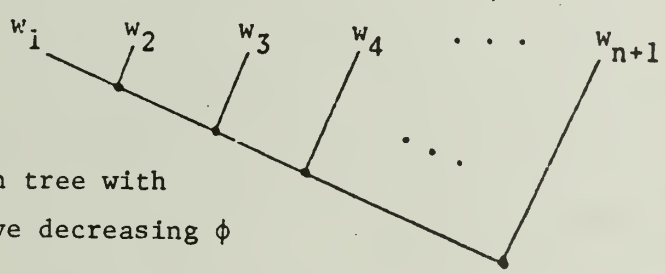


Fig. 3 Huffman tree with positive decreasing ϕ

This is also the structure of the tree that would be produced if ϕ were positive increasing, $\lambda \geq 1$, and the anti-Huffman algorithm were used, for then we would have $F(x,y) \geq \max(x,y)$ and the largest weight would always be selected. This type of tree construction is not particularly interesting but will be covered here for the sake of completeness.

Lemma 1 establishes when F is a "non-shrinking" or "strictly shrinking" weight combination function; this puts us in a position to extend the results of Theorem 1, once we make a few more observations. If $F(x,y) \geq \max(x,y)$, then it is clear that the k smallest internal node weights $[W_1(T),\ldots,W_k(T)]$ of a tree T define a subforest of T. (For, if there is any weight $W_i(T)$ in the set corresponding to an internal node whose son's weight $W_j(T)$ is not also contained in the set, then $W_i(T) < W_j(T)$, for otherwise we would have $W_j(T)$ in the set. But this is impossible because $F(x,y) \geq \max(x,y)$ implies $W_i(T) \geq W_j(T)$.) Thus Lemma 1 asserts that if ϕ is positive (resp. negative) increasing and $\lambda \geq 1$, the resulting internal node weights will have this subforest characterization: every collection of least (resp. greatest) node weights define some subforest.

The lemma also shows that, for "non-shrinking" (or "strictly shrinking") functions F we can assume ϕ is <u>positive</u> (and strictly monotone continuous) instead of assuming it is increasing, since again F is invariant of sign changes to ϕ, and ϕ must now be either positive or negative. This assumption seems to be the natural one to make in view of the following result.

Theorem 3 Let $F(x,y) = \phi^{-1}(\lambda\phi(x) + \lambda\phi(y))$ be the tree construction weight combination function where ϕ is convex, positive, and strictly monotone and $\lambda \geq 1$. If, as in Theorem 1, $\mathbf{W}(\mathbf{S})$ and $\mathbf{W}(\mathbf{T})$ are the weight sequences for the internal nodes of the trees S and T, constructed respectively by the Huffman algorithm and by any other way, then

$$\mathbf{W}(\mathbf{S}) \trianglelefteq \mathbf{W}(\mathbf{T}).$$

(The same results hold if ϕ is concave, negative, and strictly monotone.)

Proof The proof has two cases, accordingly as ϕ is strictly increasing or strictly decreasing.

Case 1: ϕ convex, positive, and strictly increasing.

We accomplish the proof in two steps: using the notation of Theorem 2, we first show that the weight sequences $\phi(\mathbf{W}(\mathbf{S}))$ and $\phi(\mathbf{W}(\mathbf{T}))$ satisfy

$$\phi(\mathbf{W}(\mathbf{S})) \trianglelefteq \phi(\mathbf{W}(\mathbf{T}))$$

and then, since ϕ^{-1} is concave increasing in this case, we can apply Theorem 2 to get $\mathbf{W}(\mathbf{S}) \trianglelefteq \mathbf{W}(\mathbf{T})$ as desired.

If there are n+1 initial leaf weights $w_1,\ldots,w_{n+1}$ we have as above $\mathbf{W}(\mathbf{S}) = [W_1(S),\ldots,W_n(S)]$ and $\mathbf{W}(\mathbf{T}) = [W_1(T),\ldots,W_n(T)]$ as the internal node weight sequences where W_i is the i^{th}-smallest such weight. In particular since $W_i(S)$ designates the weight of some internal node which is the root of some subtree S_i of the Huffman tree S, if we define

$$\mathcal{J}_i = \{ j \mid w_j \text{ is a leaf of } S_i \}$$

$$\ell_j(S_i) = \text{path length of weight } w_j \text{ in the subtree } S_i,$$

$$a_i = \sum_{j\in\mathcal{J}_i} \lambda^{\ell_j(S_i)} \phi(w_j)$$

then $W_i(S) = \phi^{-1}(a_i)$.

Defining $\mathcal{T}_i$ and $\ell_j(T_i)$ in an analogous manner, if we set

$$b_i = \sum_{j \in \mathcal{T}_i} \lambda^{\ell_j(T_i)} \phi(w_j)$$

then $W_i(T) = \phi^{-1}(b_i)$.

We now claim that the sets $\mathbf{a} = [a_1,\ldots,a_n]$ and $\mathbf{b} = [b_1,\ldots,b_n]$ are weight sequences satisfying $\mathbf{a} \preceq \mathbf{b}$. First of all since ϕ is positive increasing and $\mathbf{W(S)}$, $\mathbf{W(T)}$ are weight sequences, we know that $\mathbf{a} = \phi(\mathbf{W(S)})$ and $\mathbf{b} = \phi(\mathbf{W(T)})$ are weight sequences; in fact since ϕ "preserves order", if $W_i(S) < W_j(S)$ then $a_i = \phi(W_i(S)) < \phi(W_j(S)) = a_j$, and similarly for $\mathbf{W(T)}$ and $\mathbf{b}$.

Second, by Lemma 1 we know that $F(x,y) \geq x,y$ in this case, so the k smallest node weights $[W_1,\ldots,W_k]$ for either S or T correspond to a subforest F_k of S or T. This implies

$$\sum_{i=1}^{k} b_i = \sum_{i=1}^{k} \sum_{j \in \mathcal{T}_i} \lambda^{\ell_j(T_i)} \phi(w_j)$$

$$= \sum_{j=1}^{n+1} \left(\lambda + \lambda^2 + \ldots + \lambda^{\ell_j(F_k)} \right) \phi(w_j)$$

$$= \begin{cases} \left(\frac{\lambda}{\lambda-1}\right) \sum_{j=1}^{n+1} \left(\lambda^{\ell_j(F_k)} - 1 \right) \phi(w_j) & \text{if } \lambda > 1 \\ \sum_{j=1}^{n+1} \ell_j(F_k)\ \phi(w_j) & \text{if } \lambda = 1, \end{cases}$$

with a similar expression holding for $\sum_{i=1}^{k} a_i$.

We can now directly apply Glassey and Karp's method of proof for Theorem 1. The proof proceeds by induction on k, where we are trying to prove $\mathbf{a} \preceq \mathbf{b}$ by showing for all k that $\sum_{i=1}^{k} a_i \leq \sum_{i=1}^{k} b_i$.

The basis k=1 is trivial, and for the induction step there are two possibilities, depending on the relationship between $a_1 = \phi(W_1(S))$ and $b_1 = \phi(W_1(T))$.

Subcase 1: $a_1 = b_1$

In this case we know $\phi^{-1}(a_1) = \phi^{-1}(b_1) = F(w_1,w_2)$ and we are reduced to the proof on the set of leaf weights $\{F(w_1,w_2), w_3, \ldots, w_{n+1}\}$, for which we have by induction that

$$\sum_{i=2}^{k} a_i \leq \sum_{i=2}^{k} b_i .$$

So,

$$\sum_{i=1}^{k} a_i \leq \sum_{i=1}^{k} b_i .$$

Subcase 2: $a_1 < b_1$

As in Theorem 1, we show there is a tree $\overline{T}$ with internal weights $W_i(\overline{T}) = \phi^{-1}(c_i)$ where $\mathbf{c} = [c_1,\ldots,c_n]$ satisfies $\sum_{1\leq i\leq k} c_i \leq \sum_{1\leq i\leq k} b_i$ and, in addition, $c_1 = a_1$ so we have (by reduction to subcase 1)

$$\sum_{i=1}^{k} a_i \leq \sum_{i=1}^{k} c_i \leq \sum_{i=1}^{k} b_i$$

completing the proof that $\mathbf{a} \trianglelefteq \mathbf{b}$. This is easily done by taking the forest F_k corresponding to the least k weights $[W_1(T),\ldots,W_k(T)]$ of T and defining as before the maximum path length in this forest

$$\ell^k_{max} = \max_j \ell_j(F_k) .$$

We then choose an internal node having weight $W_p(T)=\phi^{-1}(b_p)=F(w_r,w_s)$ whose 2 (leaf) sons have path length ℓ^k_{max} in F_k and have leaf weights w_r and w_s. Since $a_1 < b_1 \leq b_p$ we know $\{w_r,w_s\}\cap\{w_1,w_2\} \neq \{w_1,w_2\}$. Assuming $w_r \leq w_s$, let $\overline{T}$ be the tree constructed exactly like T but with the leaf weights

w_r and w_1, w_s and w_2 interchanged. Then F_k is still a subforest of $\overline{T}$ (topologically T and $\overline{T}$ are isomorphic) and determines a subset of k of $\overline{T}$'s internal node weights, and consequently some k-subset of the weight sequence **c**. Specifically, if we define $\overline{T}_i$, $\overline{\mathcal{J}}_i$, $\ell_j(\overline{T}_i)$ exactly as above so that

$$c_i = \sum_{j\in\overline{\mathcal{J}}_i} \lambda^{\ell_j(\overline{T}_i)}\,\phi(w_j)$$

and $W_i(\overline{T}) = \phi^{-1}(c_i)$, then F_k defines the set

$$\mathcal{Q} = \{i \mid \phi^{-1}(c_i) \text{is the weight of some internal node in } F_k\}$$

and $|\mathcal{Q}| = k$. Moreover we must have

$$\sum_{i=1}^{k} c_i \leq \sum_{i\in\mathcal{Q}} c_i$$

since the first k weights c_i are the least such weights. But we also have

$$\sum_{i\in\mathcal{Q}} c_i \leq \sum_{i=1}^{k} b_i .$$

To show this we write for convenience

$$\Lambda_1 \equiv \lambda^{\ell_1(T)} = \lambda^{\ell_r(\overline{T})} \qquad \Lambda_2 \equiv \lambda^{\ell_2(T)} = \lambda^{\ell_s(\overline{T})}$$

$$\Lambda_m \equiv \lambda^{\ell^k_{max}(T)} = \lambda^{\ell_r(T)} = \lambda^{\ell_s(T)} = \lambda^{\ell_1(\overline{T})} = \lambda^{\ell_2(\overline{T})}$$

$$\phi_r \equiv \phi(w_r) \qquad \phi_s \equiv \phi(w_s) \qquad \phi_1 \equiv \phi(w_1) \qquad \phi_2 \equiv \phi(w_2)$$

Therefore $\phi_r \geq \phi_1$ and $\phi_s \geq \phi_2$, and in the case $\lambda > 1$ since

$$\ell_1(T),\ \ell_2(T) \leq \ell^k_{max}(T)$$

we have

$$\Lambda_1 \leq \Lambda_m \quad \text{and} \quad \Lambda_2 \leq \Lambda_m .$$

So, if $\lambda > 1$,

$$\sum_{i\in\mathcal{Q}} c_i - \sum_{i=1}^{k} b_i = \left(\frac{\lambda}{\lambda-1}\right)\Big[(\Lambda_m\phi_1 + \Lambda_m\phi_2 + \Lambda_1\phi_r + \Lambda_2\phi_s) - (\Lambda_1\phi_1 + \Lambda_2\phi_2 + \Lambda_m\phi_r + \Lambda_m\phi_s)\Big]$$

$$= \left(\frac{\lambda}{\lambda-1}\right)\Big[(\Lambda_m-\Lambda_1)(\phi_1-\phi_r) + (\Lambda_m-\Lambda_2)(\phi_2-\phi_s)\Big]$$

$$\leq 0 .$$

A trivial modification of this argument gives the proof for $\lambda=1$, so we omit it here. Thus we have shown

$$\sum_{i=1}^{k} c_i \leq \sum_{i\in\mathcal{Q}} c_i \leq \sum_{i=1}^{k} b_i$$

but since $c_1=\phi^{-1}(W_1(\overline{T})) = \phi^{-1}(F(w_1,w_2)) = a_1$ we have, by reduction to subcase 1, that

$$\sum_{i=1}^{k} a_i \leq \sum_{i=1}^{k} c_i .$$

Therefore

$$\sum_{i=1}^{k} a_i \leq \sum_{i=1}^{k} b_i$$

and Theorem 3 follows for Case 1, since we have shown that $\mathbf{a} \trianglelefteq \mathbf{b}$, and, since here ϕ^{-1} is concave increasing we can apply Theorem 2 to get immediately

$$\mathbf{W(S)} = \phi^{-1}(\mathbf{a}) \trianglelefteq \phi^{-1}(\mathbf{b}) = \mathbf{W(T)}.$$

Case 2: ϕ convex, positive, strictly decreasing

Actually in this case the Theorem is something of an understatement. We are comparing here the weight sequences

$$\mathbf{W(S)} = [W_1(S),\ldots,W_n(S)] = [\phi^{-1}(a_n),\ldots,\phi^{-1}(a_1)]$$

and $\mathbf{W(T)} = [W_1(T),\ldots,W_n(T)] = [\phi^{-1}(b_n),\ldots,\phi^{-1}(b_1)]$,

where $\mathbf{a} = [a_1,\ldots,a_n]$ and $\mathbf{b} = [b_1,\ldots,b_n]$ are weight sequences as in case 1. From the discussion following Lemma 1 we see that

$\mathbf{a}$ would be the internal node weight sequence formed using the anti-Huffman algorithm on the leaf weights $\{\phi(w_1),\ldots,\phi(w_{n+1})\}$. It follows that $a_k \geq b_k$ for $1 \leq k \leq n$, and thus we easily have both $\mathbf{a} \geq \mathbf{b}$ and $\mathbf{W}(S) \leq \mathbf{W}(T)$ as consequences.

Theorem 3 is apparently the most general possible result of its kind. To show what can happen when ϕ is not convex, we consider an example where ϕ is concave positive. Let $\phi(x) = \sqrt{x}$, $\lambda = 1$, and $U = R_+$ so that

$$F(x,y) = (\sqrt{x} + \sqrt{y})^2,$$

and suppose we are to build a tree given the leaf weights $\{1,2,3,4\}$. The Huffman algorithm produces the tree S

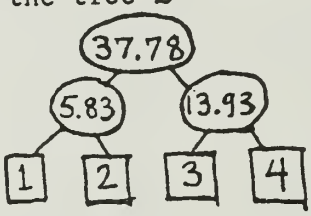


Fig. 4a Huffman tree S

for which we have $\sum_{i=1}^{3} W_i(S) = 5.83 + 13.93 + 37.78 = 57.73$, while the tree T

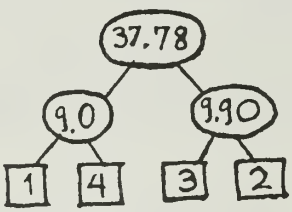


Fig. 4b Another tree T

has $\sum_{i=1}^{3} W_i(T) = 9 + 9.90 + 37.78 = 56.68$, so $\mathbf{W}(S) \not\leq \mathbf{W}(T)$. That this phenomenon will always happen when ϕ is not convex is a result of the converse of Theorem 2, which says that

$$\sum_{i=1}^{m} \phi(a_i) \leq \sum_{i=1}^{m} \phi(b_i) \text{ for all } \mathbf{a} \leq \mathbf{b} \implies \phi \text{ concave increasing.}$$

The proof is easy and we omit it.

In addition to Theorem 3 we have the following characterization of Huffman construction with the functions F considered in this paper.

Theorem 4 If $F(x,y) = \phi^{-1}(\lambda\phi(x)+\lambda\phi(y))$ where $\lambda \geq 1$ and ϕ is increasing, continuous, and sign-consistent (as in Lemma 1), then

$$W_1(S) \leq W_1(T) \quad \text{and} \quad W_n(S) \leq W_n(T),$$

i.e., the smallest and largest Huffman internal node weights are no larger than the corresponding smallest and largest node weights in any other tree.

Proof The proof is like that of Theorem 3. We discuss only the case where ϕ is positive, so $W_1(S) = F(w_1,w_2) \leq W_1(T)$ follows immediately and we must show $W_n(S) \leq W_n(T)$. If ϕ is negative, the proof is similar, but there $W_n(S) = F(w_1,w_2) \leq W_n(T)$ and we must show $W_1(S) \leq W_1(T)$. Here we have

$$W_n(S) = \phi^{-1}\Big(\sum_{j=1}^{n+1} \lambda^{\ell_j(S)} \phi(w_j)\Big),$$

$$W_n(T) = \phi^{-1}\Big(\sum_{j=1}^{n+1} \lambda^{\ell_j(T)} \phi(w_j)\Big).$$

and because ϕ is increasing

$$W_n(S) \leq W_n(T) \quad \underline{\text{iff}} \quad \phi(W_n(S)) \leq \phi(W_n(T))$$

or equivalently

$$\underline{\text{iff}} \quad \sum \lambda^{\ell_j(S)} \phi(w_j) \leq \sum \lambda^{\ell_j(T)} \phi(w_j).$$

We prove this inequality by induction on n, the number of internal nodes in the constructed tree. As a basis we have $W_1(S) = W_1(T)$ for n=1, and the theorem may be easily verified exhaustively for n=2. For the induction step we have the familiar two-case proof of Theorems 1 and 3. In the case $W_1(S) < W_1(T)$ we construct a tree $\overline{T}$ such that $\phi(W_n(S)) \leq \phi(W_n(\overline{T})) \leq \phi(W_n(T))$ in the usual manner: in the tree T we select an internal node having weight $W_p(T) = F(w_r,w_s)$ whose two (leaf) sons have maximal path length $\ell_{max} = \max_j \ell_j(T)$ in T. Since $W_1(S) \neq W_1(T)$ we know $\{w_r,w_s\} \cap \{w_1,w_2\} \neq \{w_1,w_2\}$. Assuming $w_r \leq w_s$, let $\overline{T}$ be the tree constructed exactly like T but with w_r and w_1, w_s and w_2 interchanged.

Then $W_1(\overline{T}) = F(w_1,w_2) = W_1(S)$ so by Case 1 above we have $W_n(S) \leq W_n(\overline{T})$. However we also have

$$\phi(W_n(\overline{T})) - \phi(W_n(T)) = (\lambda^{\ell_{max}} - \lambda^{\ell_1(T)})(\phi(w_1)-\phi(w_r)) + (\lambda^{\ell_{max}} - \lambda^{\ell_2(T)})(\phi(w_2)-\phi(w_s))$$

$$\leq 0 .$$

Consequently $W_n(\overline{T}) \leq W_n(T)$ and we obtain $W_n(S) \leq W_n(T)$ as desired.

Now that we know quasilinear weight combination functions are interesting, we must address the problem of determining whether an arbitrary function F(x,y) is in fact quasilinear. Fortunately this is not too difficult. Consider the following five properties of F:

(1) $F(x,y) = F(y,x)$ for all x,y in U

(2) $F(x,y) \geq \max(x,y)$ for all x,y in U (or $\leq \min(x,y)$)

(3) $F(x,y) \leq F(x,z)$ if $y \leq z$ for all x,y,z in U (F is increasing)

(4) $F(F(u,v),F(x,y)) = F(F(u,x),F(v,y))$ for all u,v,x,y in U (F is bisymmetric)

(5) $\frac{\partial F}{\partial x}$ and $\frac{\partial F}{\partial y}$ are bounded on U

We claim that if F satisfies the first four conditions then it satisfies the requirements of Theorem 4. In addition the fifth condition must be fulfilled if F is to satisfy the requirements of Theorem 3.

The necessity of the first three conditions is clear, in view of Lemma 1 and the fact that monotonicity of ϕ implies F is increasing. The fourth condition is less obvious, but Aczél has shown [Acz 66, §6.4]

that functions satisfying this bisymmetry functional equation are quasilinear. Thus the first four conditions ensure $F(x,y) = \phi^{-1}(\lambda\phi(x)+\lambda\phi(y))$ with monotone ϕ and $\lambda \geq 1$. We contend that condition (5) is also satisfied when ϕ is convex, which would permit F to satisfy Theorem 3. Using dot notation for derivatives we have

$$\frac{\partial F}{\partial x}(x,y) = \dot{\phi}^{-1}(\lambda\phi(x)+\lambda\phi(y)) \cdot \lambda\dot{\phi}(x) = \frac{\lambda\dot{\phi}(x)}{\dot{\phi}(F(x,y))}$$

since $\dot{\phi}^{-1}(x) = 1 \,/\, \dot{\phi}(\phi^{-1}(x))$. If ϕ is strictly increasing positive then $F(x,y) \geq \max(x,y)$ and $\dot{\phi} > 0$; if ϕ is strictly decreasing positive then $F(x,y) \leq \min(x,y)$ and $\dot{\phi} < 0$ (Lemma 1). So if ϕ is <u>convex</u> increasing positive we have that $\dot{\phi}$ is positive increasing, in which case

$$\frac{\lambda\dot{\phi}(x)}{\dot{\phi}(F(x,y))} \leq \frac{\lambda\dot{\phi}(x)}{\dot{\phi}(\max(x,y))} \leq \lambda.$$

If ϕ is convex decreasing positive we obtain the same bound using $\min(x,y)$ since then $|\dot{\phi}|$ is positive decreasing. Unfortunately condition (5) does not imply ϕ <u>must</u> be convex, since it is true for lumpy, but near-convex, functions. It does appear to be a fairly potent test, however: for the example in Figure 4, we find $\frac{\partial F}{\partial x} = 1 + (y/x)^{1/2}$, which is unbounded on $U = R_+$. This condition seems to characterize when the Huffman algorithm works: if F grows too quickly, then the algorithm makes mistakes in its "greedy" selection of nodes to merge. To actually test whether ϕ is convex or not, the only method currently known is to derive a power series for ϕ^{-1}, either by repeated differentiation of the functional equation

$$F(\phi^{-1}(x),\phi^{-1}(x)) = \phi^{-1}(2\lambda x)$$

followed by equating of coefficients, or else using iterative methods like the ones in [BK 76] to converge to a truncated series.

These results will be exploited in section IV. We finish this section by outlining how the above characterization extends to r-ary tree construction.

Theorem 5 Let everything be defined as in Theorem 3, with the exception that we let $F:U^r \to U$ be the r-ary function $(r \geq 2)$

$$F(x_1,x_2, \ldots ,x_r) = \phi^{-1}(\lambda \sum_{i=1}^{r} \phi(x_i)).$$

Then the results of Theorems 3 and 4 still hold.

We omit the proof, which is virtually identical to that of these theorems, with the changes that we must now define F on less that its full r arguments in the natural way (In the binary case all constructed trees are full, but that is no longer true in the r-ary case. If $n+1$ leaf weights are provided, the Huffman algorithm selects exactly the $2 + [(n-1) \bmod (r-1)]$ smallest weights for the first weight combination, and this quantity is not necessarily equal to r; however choosing this many weights guarantees that all future weight combinations can merge r weights.), and the details of showing that the tree $\overline{T}$ gives us inequalities like $W(S) \leq W(\overline{T}) \leq W(T)$ are slightly more complicated but no different in method. These details are covered in Glassey and Karp's proof in [GK 76].

IV. Cost Functions under which Huffman Trees are Optimal

In section III we described the properties of the internal node weights in Huffman trees with the weight combination function $F(x,y) = \phi^{-1}(\lambda\phi(x)+\lambda\phi(y))$. In this section we exhibit several classes of tree cost functions for which the Huffman trees are optimal by exploiting these results as much as possible. As indicated above in section II we are considering cost a function of the constructed internal node weights, so formally

$$\text{Cost}(T) = G(\mathbf{W}(\mathbf{T})) = G(W_1(T), \ldots, W_n(T)).$$

Thus $G: U^n \to R$ is to be a function under which Huffman internal node weight sequences have smallest image. We show now that cost functions that are "Schur concave" (defined momentarily) are important when all the internal node weights $W_i(T)$ are to be taken into consideration. If one is only interested in $\max W_i(T)$ or $\min W_i(T)$ (so: $W_n(T)$ or $W_1(T)$, exactly which depending on whether $|\phi|$ is increasing or decreasing, where ϕ is the function defining F above) then the cost function need only be increasing. These cost functions are apparently the most general possible for Huffman construction to be optimal when the weight combination function F is quasilinear as above. Applications will be taken up in the next section.

Definition $G: U^n \to R$ is a Schur concave function if

$$(x_i - x_j)\left(\frac{\partial G}{\partial x_i} - \frac{\partial G}{\partial x_j}\right) \leq 0$$

holds for all $x_i, x_j \in U$, $i,j \in \{1,\ldots,n\}$.

(Note: in the literature on inequalities, when Schur functions are defined the inequality above is normally reversed. We make our meaning clear by appending "concave".)

Theorem 6 (Schur & Ostrowski) $G(\mathbf{a}) \leq G(\mathbf{b})$ is true for all weight sequences $\mathbf{a} \preceq \mathbf{b}$ if and only if G is Schur concave.

Proof The proof appearing here is adapted from [Sch 23] and [Ost 52].

($\Longrightarrow$ (Schur))

Select any $\mathbf{a} = [a_1,\ldots,a_n]$ such that $a_1 \leq \ldots \leq a_n$, and set $b_1 = (1-\varepsilon)\, a_1 + \varepsilon\, a_2$, $b_2 = \varepsilon\, a_1 + (1-\varepsilon)\, a_2$, and $b_i = a_i$ for $i > 2$. Then for $\varepsilon \leq 1/2$ we have $b_1 \leq b_2$ and $\mathbf{a} \preceq \mathbf{b}$. Moreover,

$$\frac{G(b) - G(a)}{\varepsilon} = \frac{G((1-\varepsilon)a_1+\varepsilon a_2, \varepsilon a_1+(1-\varepsilon)a_2, a_3, \ldots) - G(a_1,a_2,a_3,\ldots)}{\varepsilon}$$

$$= \frac{\frac{\partial G}{\partial x_1}(\alpha_1,a_2,a_3,\ldots,a_n)\cdot\varepsilon(a_2-a_1) + \frac{\partial G}{\partial x_2}((1-\varepsilon)a_1+\varepsilon a_2,\alpha_2,a_3,\ldots)\cdot\varepsilon(a_1-a_2)}{\varepsilon}$$

where $\alpha_1 \in [a_1 , (1-\varepsilon)a_1+\varepsilon a_2]$ and $\alpha_2 \in [a_2, \varepsilon a_1+(1-\varepsilon)a_2]$, by the mean value theorem. As ε approaches zero the right hand side approaches

$$-(a_2 - a_1)\cdot\left(\frac{\partial G}{\partial x_2}(\mathbf{a}) - \frac{\partial G}{\partial x_1}(\mathbf{a})\right)$$

Thus if we are to have $G(\mathbf{a}) \leq G(\mathbf{b})$ this quantity must be positive, so G must be Schur concave, since this argument can be repeated for all pairs of indices i and j (not just 1 and 2).

($\Longleftarrow$ (Ostrowski))

Given G is Schur concave, fix b and assume that there is an $\mathbf{a} \preceq \mathbf{b}$ such that $G(\mathbf{a}) > G(\mathbf{b})$. In particular there will be a maximum such $\mathbf{a}$ -- so

assume without loss of generality that $G(\mathbf{a})$ is a maximum. Select indices k, ℓ, i, and j such that $b_k > b_\ell$ and $a_i \leq a_j$, and define a weight sequence $\bar{\mathbf{a}}$ such that $\mathbf{a} \lesssim \bar{\mathbf{a}} \lesssim \mathbf{b}$ by setting $\bar{a}_i = a_i + \varepsilon(b_k - b_\ell)$ $\bar{a}_j = a_j + \varepsilon(b_\ell - b_k)$, and $\bar{a}_m = a_m$ for $m \neq i, j$. [Note: if there are no indices k and ℓ such that $b_k > b_\ell$, we can construct a new weight sequence $\bar{\mathbf{b}}$ such that $\mathbf{a} \lesssim \bar{\mathbf{b}} \lesssim \mathbf{b}$ which does have such indices and which can be used to replace $\mathbf{b}$ in this proof.] Now set $\phi(\varepsilon) = G(\bar{\mathbf{a}})$. Then

$$\begin{aligned}
\phi'(\varepsilon) &= (b_k - b_\ell)\frac{\partial G}{\partial x_i}(\bar{\mathbf{a}}) + (b_\ell - b_k)\frac{\partial G}{\partial x_j}(\bar{\mathbf{a}}) \\
&= (b_k - b_\ell)\left(\frac{\partial G}{\partial x_i}(\bar{\mathbf{a}}) - \frac{\partial G}{\partial x_j}(\bar{\mathbf{a}})\right) \\
&> 0.
\end{aligned}$$

This contradicts the supposition that $\mathbf{a}$ was a maximal point. So there can be no point $\mathbf{a} \lesssim \mathbf{b}$ such that $G(\mathbf{a}) > G(\mathbf{b})$ -- we must have $G(\mathbf{a}) \leq G(\mathbf{b})$.

It is worth mentioning that all strictly concave functions G (so $G'' < 0$) are Schur concave -- see [Sch 23, p.12]. Generally speaking the importance of this theorem has not been properly appreciated; recently Wong and Yue have found a number of uses for it in storage applications. See for example [WY 73].

The next three theorems follow as corollaries from Theorem 6 and Section III. In each we compare the cost of trees S and T built using the weight combination function $F(x,y)$ of section III, where S is the tree built by the Huffman algorithm and T is any other tree. As usual, $\mathbf{W}(S) = [W_1(S),\ldots,W_n(S)]$ and $\mathbf{W}(T) = [W_1(T),\ldots,W_n(T)]$ denote the internal node weight sequences for these trees.

Theorem 7 Let F be as in Theorem 3. Then the Huffman tree will have least cost when G is any Schur concave function of the internal node weights.

Proof This is a simple corollary of Theorem 6, since Theorem 3 guarantees $\mathbf{W}(S) \preceq \mathbf{W}(T)$, so $G(\mathbf{W}(S)) \leq G(\mathbf{W}(T))$.

Theorem 8 Let $F(x,y) = \phi^{-1}(\lambda\phi(x)+\lambda\phi(y))$ with $\lambda \geq 1$ and ϕ positive monotone continuous, analogous to Theorem 4. Then the Huffman tree will have least cost when G is a function of the following form:

If ϕ is increasing, $G = \tilde{G} \cdot \phi$ where $\tilde{G}$ is Schur concave.

If ϕ is decreasing, $G = \tilde{G} \cdot \phi$ where $\tilde{G}$ is monotone decreasing

(i.e., $\tilde{G}(x_1,\ldots,x_i,\ldots,x_n) \leq \tilde{G}(x_1,\ldots,x_i',\ldots,x_n)$ if $x_i \geq x_i'$).

Proof Note $G(\mathbf{W}(T)) = \tilde{G}(\phi(\mathbf{W}(T))) = \tilde{G}([\phi(W_1(T)),\ldots,\phi(W_n(T))])$. Using an argument as in the proof of Theorem 3, it is clear that if ϕ is increasing then $\phi(\mathbf{W}(S)) \preceq \phi(\mathbf{W}(T))$, and, if ϕ is decreasing, then not only $\phi(\mathbf{W}(S)) \preceq \phi(\mathbf{W}(T))$ but also $\phi(W_i(S)) \leq \phi(W_i(T))$ for i=1,...,n. Theorem 6 gives us the first part of the theorem; the second is easy.

Theorem 9 Let F be as in Theorem 4. Then the Huffman tree will have least cost when G is of the form $G(\mathbf{W}(T)) = \psi(\max W_i(T))$ or $G(\mathbf{W}(T)) = \psi(\min W_i(T))$, where ψ is any monotone increasing function.

Proof Immediate from Theorem 4.

Although these are the only cost functions we discuss here, it should be made clear that there may be other Huffman-optimal ones. The functions here prey (thoroughly) on the properties of Huffman tree internal node weights; discovery of other properties could lead to other cost criteria favorable for Huffman trees.

It must be emphasized that varying the weight space U can greatly affect the performance of the Huffman algorithm. Consider the weight combination function $F(x,y) = xy$. On $U = [0,1]$ we can take $\phi(x) = -\log(x)$, a positive convex decreasing function (the base of the logarithm is immaterial); from Theorem 6 we know that under cost functions like $G = \text{sum}$, Huffman trees will be optimal. However on $U = [1,\infty)$ we have $\phi(x) = +\log(x)$, a positive concave increasing function, so under the cost $G = \text{sum}$ there is no guarantee that a Huffman tree will be best. Even worse, if we choose $U = [0,\infty)$ there is then no sign-consistent strictly monotone function ϕ determined by F. Thus some of the above theorems are more restrictive than they appear at first.

V. Bounds on the weights of Huffman trees

We now know that under certain circumstances generalized from the weighted path-length tree construction system discussed in section II, the Huffman algorithm will produce an optimal tree. This generalization involved appropriate application of quasilinearity and Schur concaveness. We are led to ask the natural question of whether the "Noiseless Coding Theorem" (NCT) of information theory generalizes using these notions also. The NCT states that for the r-ary weighted path-length construction system, if $\ell_1,\ldots,\ell_n$ denote the respective path lengths of the leaf weights $w_1,\ldots,w_n$ in the Huffman tree, then we have the inequality

$$(1) \qquad -\sum_{i=1}^{n+1} w_i \log_r(w_i/w) \;\leq\; \sum_{i=1}^{n+1} w_i \ell_i \;<\; -\sum_{i=1}^{n+1} w_i \log_r(w_i/w) + w$$

where $w = \Sigma\, w_i$, and equality holds on the left iff $w_i = r^{-\ell_i}$ for all i.

See, for example, [Gal 68,pp.50-55]. Since the Huffman tree has the smallest weighted path length, the inequality gives us a lower bound on the weighted path-length of any tree; surprisingly it also gives us a relatively tight upper bound on the Huffman tree. This inequality is referred to as the NCT because of its original application in estimating the average number of code symbols per message required to send a set of encoded messages across a noiseless channel.

Happily the NCT does indeed generalize for tree construction with quasilinear weight combination functions. We show the generalization in three consecutive theorems, each involving more general functions than its predecessor. It is important to notice that the latter two theorems give bounds only on the root weight of the constructed tree; neither can be

extended to a statement about weighted path length as long as λ, the constant used in defining F, is greater than one. The schism corresponding to choosing $\lambda = 1$ or $\lambda > 1$ was already encountered in the proof of Theorem 3.

Define the Rényi entropy of order α, H_α, of a collection of probabilities $\{p_1, \ldots, p_m\}$ (so $\Sigma\, p_i = 1$) by

$$H_\alpha(p_1,p_2, \ldots, p_m) = \frac{1}{1-\alpha} \log_r\Big(\sum_{i=1}^{m} p_i^{\alpha}\Big).$$

It is well known that the limit as $\alpha \to 1$ of this Rényi entropy is simply the Shannon entropy

$$H(p_1,p_2, \ldots, p_m) = -\sum_{i=1}^{m} p_i \log_r(p_i).$$

We now have the following theorems.

Theorem 10 Consider construction in the weighted path-length system, so $F(x_1,\ldots,x_r) = x_1+\ldots+x_r$. Then

$$w\cdot H(w_1/w,w_2/w,\ldots,w_n/w) \leq \sum_{i=1}^{n+1} w_i\ell_i < w\cdot(H(w_1/w,w_2/w,\ldots,w_n/w) + 1)$$

where $w = \Sigma\, w_i$ and $\ell_1, \ell_2, \ldots, \ell_n$ are the path lengths of $w_1, \ldots, w_{n+1}$ in the Huffman tree. Equality on the left is achieved iff $w_i = r^{-\ell_i}$ for all i.

Proof This is the Noiseless Coding Theorem.

Theorem 11 If W is the root node weight produced by r-ary Huffman construction with the weight combination function $F(x_1,\ldots,x_r) = \lambda \sum_{i=1}^{r} x_i$ $(\lambda > 1)$, then

$$w \cdot \lambda^{H_\alpha(\, w_1/w,\ \ldots,\ w_{n+1}/w\,)} \leq W < w \cdot \lambda^{H_\alpha(\, w_1/w,\ \ldots,\ w_{n+1}/w\,) + 1}$$

where $\alpha = 1/(1+\log_r(\lambda))$ and $w = \Sigma\, w_i$. Equality holds on the left iff

$$r^{-\ell_i} = w_i^{\alpha} / \left(\sum_{i=1}^{n+1} w_i^{\alpha} \right) \qquad \text{for all } i.$$

Proof Note first that $W = \sum_{i=1}^{n+1} w_i \lambda^{\ell_i}$ with this weight combination function, where $\{\ell_i\}$ are the path lengths as in Lemma 1. With this, Campbell proves the left inequality in [Cam 66] by appealing to Hölder's inequality

$$\left(\Sigma\, \xi_j^{\,p} \right)^{1/p} \left(\Sigma\, \eta_j^{\,q} \right)^{1/q} \leq \Sigma\, \xi_j \eta_j \qquad \left(\frac{1}{p} + \frac{1}{q} = 1,\ p<0 \right)$$

with the substitutions $p = \log_r(1/\lambda)$, $q = 1 - \alpha$, $\xi_j = (w_j/w)^{1/p}\, r^{-\ell_j}$, and $\eta_j = (w_j/w)^{-1/p}$. The equality condition above is that of Hölder, stating when the values $\xi_j^{\,p}$ and $\eta_j^{\,q}$ are "proportional". The upper bound is established in two steps: first one shows that choosing the path lengths ℓ_i to be

$$\ell_i = \lceil -\log_r(w_i^{\alpha} / (\Sigma\, w_i^{\alpha})) \rceil$$

(from the equality condition) leads to a set of path lengths of a valid r-ary tree whose root node weight undercuts the stated upper bound. One then invokes Theorem 4: since the Huffman tree has the smallest root node weight of all trees, it also undercuts the bound.

Theorem 12 If W is the root node weight produced by Huffman construction with

$$F(x_1, \ldots, x_r) = \phi^{-1}\Big(\lambda \sum_{i=1}^{r} \phi(x_i) \Big) \qquad (\lambda > 1)$$

where ϕ is a positive, increasing function, then

$$\phi^{-1}\Big(w^{\phi} \; \lambda^{H_\alpha(\phi(w_1)/w^{\phi},\ldots,\phi(w_{n+1})/w^{\phi})} \Big) \leq W$$

$$W < \phi^{-1}\Big(w^{\phi} \; \lambda^{H_\alpha(\phi(w_1)/w^{\phi},\ldots,\phi(w_{n+1})/w^{\phi})} + 1 \Big)$$

where $\alpha = 1/(1+\log_r(\lambda))$ and $w^{\phi} = \Sigma\, \phi(w_i)$, with equality holding iff

$$r^{-\ell_i} = \phi(w_i)^{\alpha} \Big/ \Big(\sum_{j=1}^{n+1} \phi(w_j)^{\alpha} \Big) \qquad \text{for all } i.$$

Proof Follows directly from Theorem 11 by replacing $w_i \mapsto \phi(w_i)$ and applying ϕ^{-1} to the bound there. We are using the observation after Lemma 1 in Section III, that Huffman construction on $\{w_1,\ldots,w_{n+1}\}$ with F as in Theorem 12 results in a tree that is identical to what we would obtain using Huffman construction on $\{\phi(w_1),\ldots,\phi(w_{n+1})\}$ with F as in Theorem 11.

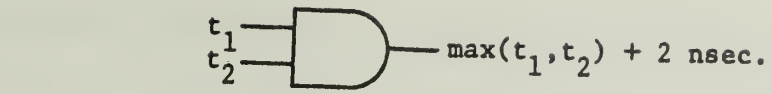
Corollary 1 Let W be the root weight of the Huffman tree in the tree-height construction system, with $F(x_1,\ldots,x_r) = \max(x_1,\ldots,x_r) + c$ $(c > 0)$. Then

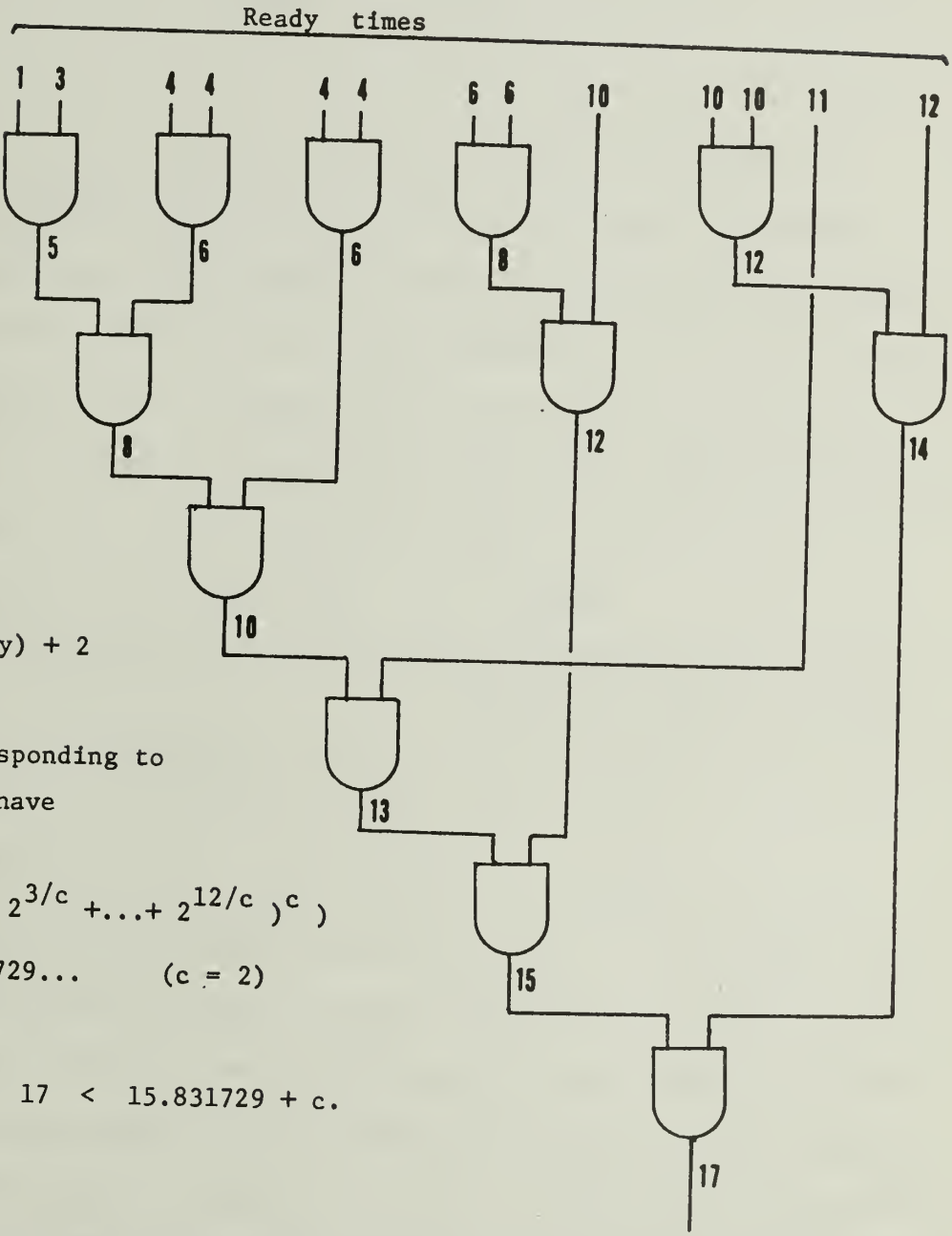
$$\log_r\Big(\Big(\sum_{i=1}^{n+1} r^{w_i/c} \Big)^{c} \Big) \leq W < \log_r\Big(\Big(\sum_{i=1}^{n+1} r^{w_i/c} \Big)^{c} \Big) + c.$$

Moreover, if $\ell_1,\ldots,\ell_{n+1}$ are the respective path lengths of the leaves in the tree, then equality is attained on the lower bound iff, for all i,

$$\ell_i = \log_r\Big(\Big(\sum_{j=1}^{n+1} r^{w_j/c} \Big) \Big/ r^{w_i/c} \Big).$$

Proof Set $\phi(x) = r^{px}$, $\lambda = r^{pc}$ in Theorem 12 and let $p \to \infty$. This extends the work of Golumbic [Gol 76], who has proved the above inequality for the useful case where $c = 1$ and all the weights w_i are integral. An example of the application of this corollary is shown in Figure 5.





$F(x,y) = \max(x,y) + 2$

Note that corresponding to Corollary 1 we have

$$\log_2\left(\left(2^{1/c} + 2^{3/c} + \ldots + 2^{12/c} \right)^c \right) = 15.831729\ldots \qquad (c = 2)$$

and

$$15.831729 \leq 17 < 15.831729 + c.$$

Figure 5. Application of Corollary 1: Huffman AND-tree bounds

VI. Applications and Open Problems

We have just shown that for wide classes of tree construction systems the Huffman algorithm produces optimal trees. As a first application of the theorems above (which were derived as extensions of the traditional weighted path length system F = sum, G = sum) we prove that Huffman construction in the tree height system

$$F(x_1, \ldots, x_r) = \max(x_1, \ldots, x_r) + c \qquad (c > 0)$$
$$G(\mathbf{W}(\mathbf{T})) = \max W_i(T)$$

is optimal. The demonstration was hinted at in Section III: if we consider the family of functions

$$F(x_1, \ldots, x_r) = \phi^{-1}\left(\lambda \sum_{i=1}^{r} \phi(x_i) \right)$$

$$G(\mathbf{W}(\mathbf{T})) = \phi^{-1}\left(\Sigma\, \phi(W_i(T)) \right) \quad \underline{\text{or}} \quad = \max W_i(T)$$

where $\phi(x) = r^{px}$, $\lambda = r^{pc}$. Then since ϕ is convex increasing and $\lambda \geq 1$, Theorems 8 or 9 imply Huffman trees will have least cost. Since in the limit as $p \to \infty$ we approach the max functions F and G of the of the tree height construction system, we have established that Huffman's algorithm is in this case optimal. Demonstrating this connection was one of the main purposes for starting this work.

Another application of this extension of Huffman tree construction is the generation of codes which are optimal under criteria other than Huffman's original one, equivalent to weighted path-length [Huf 52]. A moderate literature has grown up around this subject; it is surprising that no corresponding analogue of Huffman's algorithm has also been developed. We outline several known results, including interesting bounds on average codeword length like that of the Noiseless Coding Theorem, and then present these Huffman analogues.

In the context of coding the leaf weights $\{w_1,\ldots,w_{n+1}\}$ are probabilities (so $\Sigma w_j = 1$), representing the relative frequencies of occurrence of a set of (n+1) messages which are to be encoded into D-ary codewords ($D \geq 2$). Let the length of the message with probability w_j be called ℓ_j; we are then interested in minimizing the "quasiarithmetic mean codeword length" [Acz 74], [Cam 66]

$$L(\mu,\{w_j\},\{\ell_j\}) = \mu^{-1}\left(\sum_{j=1}^{n+1} w_j\, \mu(\ell_j)\right)$$

or some similar code cost measure; here μ is a continuous, strictly increasing function on R_+. For example, when $\mu(x) = x$ we get the traditional weighted path-length; other "translative" forms of L have been considered in [Cam 66], [Acz 74], and [Nath 75]. Although this measure of codeword length is quite general, most special cases treated in the literature can be handled by the extended Huffman construction presented here. We consider three cases one by one; each is based on Rényi's entropy of order α

$$H_\alpha(w_1,\ldots,w_{n+1}) = \frac{1}{1-\alpha} \log_D\left(\sum_{j=1}^{n+1} w_j^{\alpha} \right).$$

Here D is the size of the code letter alphabet, i.e., codewords can be viewed as D-ary numbers. As mentioned in Section V, Rényi's entropy has the property that its limit, as $\alpha \to 1$, is the usual Shannon entropy

$$H(w_1,\ldots,w_{n+1}) = -\sum_{j=1}^{n+1} w_j \log_D(w_j).$$

Campbell [Cam 65] now defines an exponential codeword length average L(t) by setting $\mu(x) = D^{tx}$ so that

$$L(t) = \frac{1}{t} \log_D\left(\Sigma\, w_j\, D^{t\ell_j} \right) = \log_\lambda\left(\Sigma\, w_j\, \lambda^{\ell_j} \right)$$

where $t > 0$ and $\lambda = D^t > 1$. He then proves that

(1) $\lim_{t\to 0} L(t) = \sum_j w_j \ell_j$

(2) $\lim_{t\to\infty} L(t) = \max_j \ell_j$

(3) $H_\alpha(w_1,\ldots,w_{n+1}) \leq L(t)$ where $\alpha = \frac{1}{1+t} = \frac{1}{1 + \log_D(\lambda)}$

with equality holding when $D^{-\ell_j} = w_j^{\alpha}/(\Sigma w_j^{\alpha})$.

Now consider general Huffman construction as discussed in section II with $F(x,y) = \lambda(x+y)$ and $G(\mathbf{W}(T)) = \log_\lambda(W_n(T))$. Then

$$\text{Cost}(T) = G(\mathbf{W}(T)) = L(t) = L(\log_D(\lambda)),$$

so Huffman construction with this weight combination function F produces optimal exponential-length-cost trees by Theorem 9.

Aczél [Acz 74], besides citing results of Campbell for the degenerate case $t<0$ ($\lambda<1$) above, considers the result when $\mu(x) = (\lambda^x-1)/(\lambda-1)$ (again, $\lambda = D^t$) and shows that

$$L(\mu) = \mu^{-1}\left(\sum_{j=1}^{n+1} w_j\, \mu(\ell_j) \right)$$

satisfies $((\Sigma w_j^{\alpha})^{1/\alpha} - 1)/(\lambda - 1) \leq \mu(L(\mu))$, where again $\alpha = 1/(1+t) = 1/(1 + \log_D \lambda)$. But notice that when $F(x,y) = \lambda(x+y)$ and $G(W(T)) = \mu^{-1}(\frac{1}{\lambda} \Sigma W_i(T))$, then because $\mu(m) = 1 + \lambda + \ldots + \lambda^{m-1}$

$$\text{Cost}(T) = G(W(T)) = L(\mu).$$

So, by Theorem 7, since G is Schur concave, Huffman construction with this function F again produces the optimal code tree (identical to the one constructed for Campbell's average codeword length).

Lastly, Nath has come up with nice results by defining what he calls the average codeword length of order α ($\alpha > 1$) [Nath 75]

$$L(\alpha) = (\alpha - 1)^{-1} \log_D \Big(\sum_{j=1}^{n+1} w_j^{\alpha} D^{(\alpha-1)\ell_j} / w^{\alpha} \Big)$$

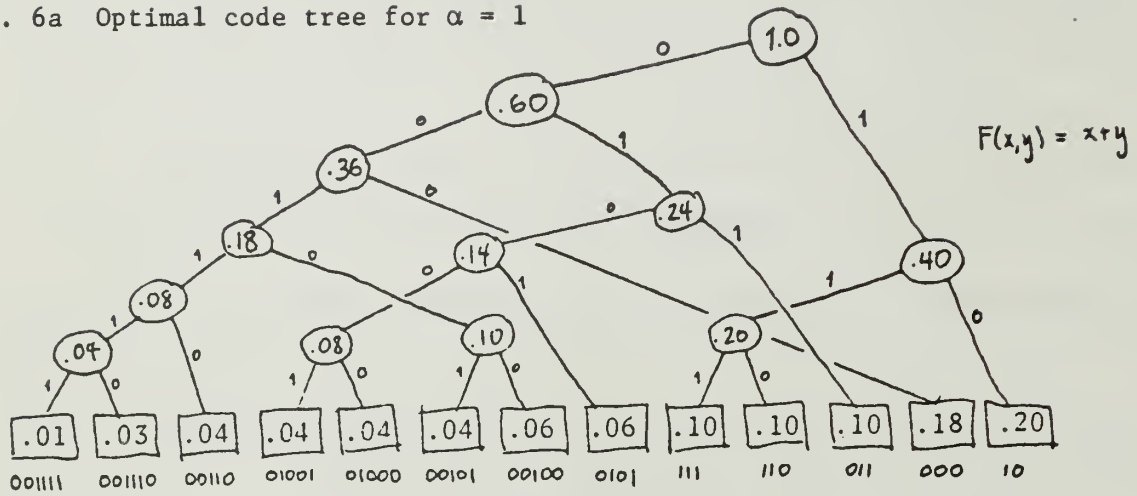
$$= \log_{\lambda} \Big(\sum_{j=1}^{n+1} w_j^{\alpha} \lambda^{\ell_j} / w^{\alpha} \Big)$$

where $w^{\alpha} = \Sigma w_j^{\alpha}$ and $\lambda = D^{(\alpha-1)}$. He shows that

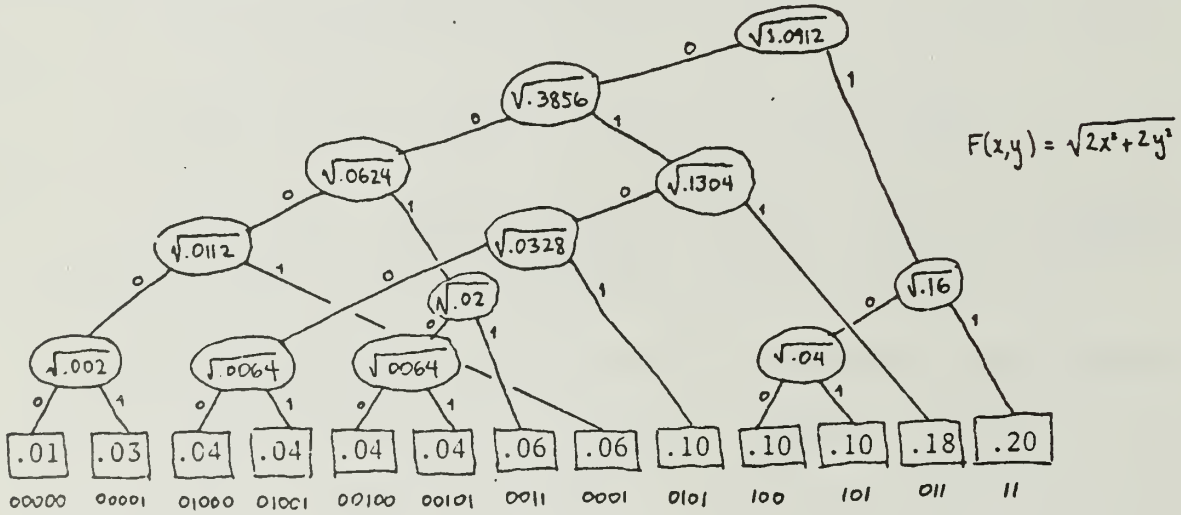
$$H_{\alpha}(w_1,\ldots,w_{n+1}) \leq L(\alpha) \quad \text{with equality iff } w_j = D^{-\ell_j} \text{ for all } j.$$

Now when $F(x,y) = (\lambda x^{\alpha} + \lambda y^{\alpha})^{1/\alpha}$ and $G(W(T)) = \log_{\lambda}(W_n(T)^{\alpha} / w^{\alpha})$ we find $\text{Cost}(T) = G(W(T)) = L(\alpha)$, so by Theorem 9 Huffman construction with this function F produces optimal trees here, i.e., produces code trees of least average length $L(\alpha)$. To illustrate this, we consider construction of an optimal binary code for the ensemble of 13 messages given in [Huf 52]. One of the nice features of $L(\alpha)$ is that its limit as $\alpha \to 1$ is the traditional average codeword length (weighted path-length); so in Figure 6 we display optimal code trees for the ensemble under the cost function $L(\alpha)$ for both $\alpha = 1$ and $\alpha = 2$, giving codeword assignments and $L(\alpha)$ in each case.

Fig. 6a Optimal code tree for $\alpha = 1$



$$L(1) = \sum w_j \ell_j = \sum w_i = 3.42$$



$$L(2) = \log_2\left[\sum w_j^2 \, 2^{\ell_j} / \sum w_j^2\right] = \log_2\left[W_n^2 / \sum w_j^2\right] \approx 3.22$$

Fig. 6b Optimal code tree for $\alpha = 2$

The bounds derived in Corollary 1 of Section V may be used in any problem where one is trying to minimize parallel processing time, as Golumbic has emphasized [Gol 76]; the optimal circuit for fanning-in data which are ready at different times is the Huffman tree with the max + c weight combination function, c being the time required to merge the r inputs of any internal node. M. Dale Skeen of the University of Illinois has pointed out that Corollary 1 could be used to prove the following result.

We are concerned with constructing a large multiplexor from many smaller ones. The small multiplexors should all be the same size (have the same number of inputs), but we are interested in seeing the behavior of the circuit completion time T_r as we increase the number r of inputs of these small multiplexors. Assume that a multiplexor with fan-in r takes time $\lceil \log_2(r) \rceil$ to achieve stable output after all its inputs become stable. Then it is clear that, since a circuit built from binary (r=2) multiplexors can always be derived immediately from a circuit built from r-ary (r > 2) multiplexors simply by replacing each r-ary node with a small balanced binary tree, we must have $T_2 \leq T_r$ -- the completion time of the binary Huffman multiplexor is no greater than that of the r-ary Huffman multiplexor. However the following theorem shows that using r > 2 does not significantly hurt timing:

Theorem (Skeen) $\quad T_2 \leq T_r < T_2 + \lceil \log_2(r) \rceil.$

Proof By Corollary 1, $T_r < \log_r((\Sigma\, r^{w_i/c})^c) + c$

where $c = \lceil \log_2(r) \rceil$. Also by Corollary 1 we know that

$$T_2 \geq \log_2\left(\Sigma\, 2^{w_i}\right) = \log_{2^c}\left(\left(\Sigma\, (2^c)^{w_i/c}\right)^c\right)$$

$$\geq \log_r\left(\left(\Sigma\, (r)^{w_i/c}\right)^c\right).$$

The last inequality follows since $\log_a\left(\Sigma\, a^{x_i}\right)$ is an increasing function of a, provided all the values x_i are positive. Combining these results on T_2 and T_r we find $T_r - T_2 < c = \lceil \log_2(r) \rceil$ as desired.

Other possible applications of this theory being investigated currently include the construction of optimal restricted-height trees (a much harder problem than that of restricted-height search trees discussed in [Itai 76], since no obvious dynamic programming solution exists) and construction of optimal weighted trees where the weights are vectors with multiple components.

There are several open problems. First it would be nice if there were some criterion like condition (5) at the end of section III which would enable us to determine whether F satisfies the requirements of Theorem 3 without having to know explicitly what the conjugating function ϕ is. Secondly, it is natural to ask whether there are other nontrivial construction systems, apart from those considered here, which are optimal under the Huffman algorithm -- or whether we have categorized the most general circumstances under which Huffman construction is optimal. That F must necessarily be quasilinear if G is Schur concave, etc., seems very plausible yet difficult to prove.

VII. References

[Acz 66] Aczél, J. Lectures on Functional Equations and their Applications. NY: Academic Press, 1966.

[Acz 74] _____. "Determination of all additive quasiarithmetic mean codeword lengths". Z. Wahrsch. verw. G. 29, 351-360 (1974).

[BK 76] Brent, R. & H.T. Kung. "Fast Algorithms for Manipulating Formal Power Series". Tech. rept., Carnegie-Mellon Univ., Jan. 1976.

[Cam 65] Campbell, L.L. "A coding theorem and Rényi's entropy". Information and Control 8, 423-429 (1965).

[Cam 66] _____. "Definition of Entropy by Means of a Coding Problem". Z. Wahrsch. verw. G. 6, 113-118 (1966).

[Cap 76] Caprani, O. "Roundoff Errors in Floating-Point Summation". BIT 15, 5-9 (1975).

[Eve 73] Even, S. Algorithmic Combinatorics, Ch. 7. NY: Macmillan, 1973.

[FB 72] Frazer, W.D. & B.T. Bennett. "Bounds on Optimal Merge Performance, and a Strategy for Optimality". JACM 19, 641-648 (Oct. 1972).

[Fuc 47] Fuchs, L. "A New Proof of an Inequality of Hardy-Littlewood-Pólya". Mat. Tidsskr. B 1947, pp. 53-54.

[Gal 68] Gallager, R.G. Information Theory and Reliable Communication. NY: John Wiley & Sons, Inc., 1968.

[GK 76] Glassey, C.R. & R.M. Karp. "On the Optimality of Huffman Trees". SIAM J. Appl. Math 31, 2, 368-378 (Sept. 1976).

[Gol 76] Golumbic, M.C. "Combinatorial Merging". IEEE Trans. Comput. C-25, 11, 1164-1167 (Nov. 1976).

[HLP 34] Hardy, G.H., J.E. Littlewood, G. Pólya. Inequalities. Cambridge: The University Press, 1934.

[HT 71] Hu, T.C. & A.C. Tucker. "Optimal Computer Search Trees and Variable-length Alphabetical Codes". SIAM J. Appl. Math. 21, 4, 514-532 (1971).

[Huf 52] Huffman, D.A. "A Method for the Construction of Minimum-redundancy Codes". Proc. IRE 40, 1098-1101 (1952).

[Itai76] Itai, A. "Optimal Alphabetic Trees". SIAM J. Comput. 5, 9-18 (1976).

[Knu 68] Knuth, D.E. Fundamental Algorithms: The Art of Computer Programming vol. 1 (section 2.3.4.5 on "Path length"). Reading, MA: Addison-Wesley, 1968.

[vanL76] van Leeuwen, J. "On the Construction of Huffman Trees". Proc. 3rd International Colloquium on Automata, Languages, and Programming. Edinburgh, July 1976, pp.382-410.

[Liu 76] Liu, J.W.S. "Algorithms for Parsing Search Queries in Inverted File Document Retrieval Systems". ACM Trans. Database Systems 1, 4, 299-316 (Dec. 1976).

[Mit 70] Mitrinović, D.S. Analytic Inequalities. NY: Springer-Verlag, 1970.

[Nath75] Nath, P. "On a Coding Theorem Connected with Rényi's Entropy". Information and Control 29, 234-242 (1975).

[Ost 52] Ostrowski, A. "Sur quelques applications des fonctions convexes et concaves au sens de I. Schur" (offert en homage à P. Montel). J. Math. Pures Appl. 31, 253-292 (1952).

[Rub 76] Rubin, F. "Experiments in Text-file Compression". CACM 19, 11, 617-623 (Nov. 1976).

[Sam 75] Sameh, A.H. Private communication.

[Sch 23] Schur, I. "Ueber eine Klasse von Mittelbildungen mit Anwendungen auf die Determinantentheorie". Sitzungsber. Berl. Math. Ges. 22, 9-20 (1923).

[WY 73] Wong, C.K. & P.C. Yue. "A Majorization Theorem for the Number of Distinct Outcomes in N Independent Trials". Discrete Mathematics 6, 391-398 (1973).

[Zim 59] Zimmerman, S. "An Optimal Search Procedure". AMM 66, 690-693 (1959).

Last night I went and raced with the Highway Patrol
But that Pontiac done had more guts than mine.
And so I wrapped my tail around a telephone pole
And now my baby she just sits a cryin'.
I'm up in heaven, darlin', now don't you cry;
Ain't no reason why you should be blue.
Just go on out and race a cop in Daddy's old Ford
And you can join me up in heaven, too.

-- T. Pynchon, V.

3
Techniques for Evaluating Nonlinear Recurrences

I. Introduction

We are concerned here with the evaluation of an m^{th}-order recurrence

$$x_k = F(x_{k-1},\ldots,x_{k-m}) \qquad (1 \leq k \leq m)$$

on a parallel machine. It is now well-known that if F is a linear function of its arguments then there exist efficient, stable parallel algorithms for evaluating all n iterates in O(log n) time [CKS 76]. If F is nonlinear, however, then no general methods besides the obvious O(n) one were known until recently for evaluating the recurrence. In fact Kung proved that when F is a rational function (ratio of two polynomials) of degree greater than one and algebraic methods are used, then parallelism can speed up the recurrence only by at most a constant factor. Thus a corollary would be that the Newton-Raphson square root iteration

$$x_{k+1} = \frac{1}{2}\left(x_k + \frac{A}{x_k} \right) = (x_k^2 + Ax_k)/(2x_k),$$

a first order recurrence of degree two, cannot be sped up significantly by algebraic changes of variables (replacing x_k by a rational function of x_k for all k), forward substitution, and so forth.

Interest in this problem evolved from continuing work on parallelism supervised by D.J. Kuck at the University of Illinois. Nonlinear recurrences arise in many serial algorithms (linear algebra routines in particular) so methods to solve them quickly in parallel would improve the overall effectiveness of parallel algorithm design. In addition to this, the PARAFRASE project [Kuck76] has confirmed that nonlinear recurrences crop up in real FORTRAN programs (though not nearly as often as do linear

recurrences), suggesting it might be beneficial if a compiler producing parallel code for these programs could generate something better than the obvious serial code.

It will be shown here that in many cases (notably the general first-order nonlinear recurrence with constant coefficients) nonlinear recurrences can be transformed into problems which can be rapidly solved on a parallel machine. The extent to which this transformation process can be automated is discussed below, and it turns out that the first-order, constant-coefficient case is essentially automatizable (and hence could be embedded in a compiler). However, it is not clear that this would be a useful compiler feature; instead, it would probably be more cost-effective to train the compiler to recognize and transform several frequently-used nonlinear recurrences which are linearizable. A number of special linearizable forms are listed in section III.

This discussion is an expansion of the material appearing in [Par 77], being much more complete with regard to detail. We concentrate our attention on the real first-order nonlinear recurrence $x_k = F(x_{k-1})$ since it is already difficult and since results for higher orders rely on the first-order theory. The main thrust of [Par 77] was to show that there often exist non-algebraic transforms of this problem (bypassing Kung's theorem by the use of transcendental functions, evaluated within limited precision) which are linear. More precisely we look for a nonalgebraic "change of variables" function $\phi(x_k) = y_k$ such that, for example,

$$F(\phi^{-1}(y)) = \phi^{-1}(sy)$$

where s is a constant. Then our iteration becomes

$$y_{k+1} = s\, y_k$$

with $y_0 = \phi^{-1}(x_0)$ and $x_k = \phi(y_k)$ for $k > 0$. This linearized problem is seen to be much easier to evaluate in parallel than the original nonlinear problem; in fact we have, for any positive k,

$$x_k = F^{[k]}(x_0) = \phi^{-1}(s^k\, \phi(x_0)),$$

$F^{[k]}$ denoting the k-fold composition of F.

Unfortunately, after [Par 77] appeared I was made aware that the above approach to solving nonlinear recurrences has been studied for some time, though mainly outside the U.S. and then as much as fifty years ago or more. Schröder [Sch 1871] is given credit for having first analyzed the problem of determining the function ϕ which, for a given function F, satisfies the functional equation

$$\phi(F(x)) = s\, \phi(x).$$

This Schröder function ϕ, if invertible, is easily shown to be equivalent to the change of variables ϕ derived above. Since Schröder's work appeared a large number of papers have accumulated discussing one aspect or another of the nonlinear iteration problem. Probably the most complete reference is Kuczma's book [Kuc 68], which is quite thorough and contains an extensive bibliography. An interesting overview on iteration also appears in Chapter 2 of [Mel 73]. This discussion will therefore survey the practical implications of the theoretical background of the problem only very briefly, giving references to fuller analyses in the literature, and will lay emphasis instead on some new work concerning how first-order nonlinear recurrence simplification might

actually be used in the development of new parallel algorithms or implemented in a compiler. This new work consists of, apart from the synthesis of germane old material useful for computational purposes, exhibiting a number of linearizable nonlinear recurrence forms (especially in section III.3) and devising a methodology for the linearization of the first-order, constant-coefficient iteration (which is applied to several examples in section IV). Throughout the intent has been to make the subject accessible to parallel algorithm designers interested in the known results, or in working along these lines.

II. Theoretical Considerations

In this section we give briefly the definitions needed to discuss real iterations, particularly of the kind one is apt to find in programs. An iteration $\langle F, x_0, n\rangle$ is a (possibly infinite) sequence of real values $\{x_k\}_{1 \le k \le n}$ where each x_k is defined recursively in terms of its predecessor by $x_k = F(x_{k-1})$, F being a real-valued function. The modulus E of F is a subset of the reals on which F is injective ($F:E \to E$) and on which the iteration is defined. Note that the starting point x_0 of any iteration must be contained in the modulus. A submodulus I is a subset of the modulus which also enjoys the injective property, i.e., $F:I \to I$. (Note: I may be an open or semi-open set).

We write $F \in C^r[I]$ to mean that F is r-times differentiable on the set I. If r=0 this means F is continuous, and if $r=\infty$ then F has derivatives of all orders -- the case we will normally be interested in. A fixed point ξ of F is a point in F's modulus such that $F(\xi) = \xi$. Supposing that $F \in C^1[I]$, where ξ is in I, we say ξ is attractive if $|F'(\xi)| < 1$; repulsive if $|F'(\xi)| > 1$; and indifferent if $|F'(\xi)| = 1$. (Similar definitions of attractiveness can be made if F is only continuous.) Also, ∞ can be a fixed point, but we alter the definition of attractiveness to mean that $F(x) > x$ for all sufficiently large x. The intuition behind this terminology is simply that iterations normally converge to attractive fixed points and diverge from repulsive ones; Figure 7 should help clarify this. Formally, if for every fixed point ξ of a continuous function F we define the attractive domain $A_F(\xi)$ of ξ to be those points x such that

$$\lim_{k \to \infty} F^{[k]}(x) = \xi$$

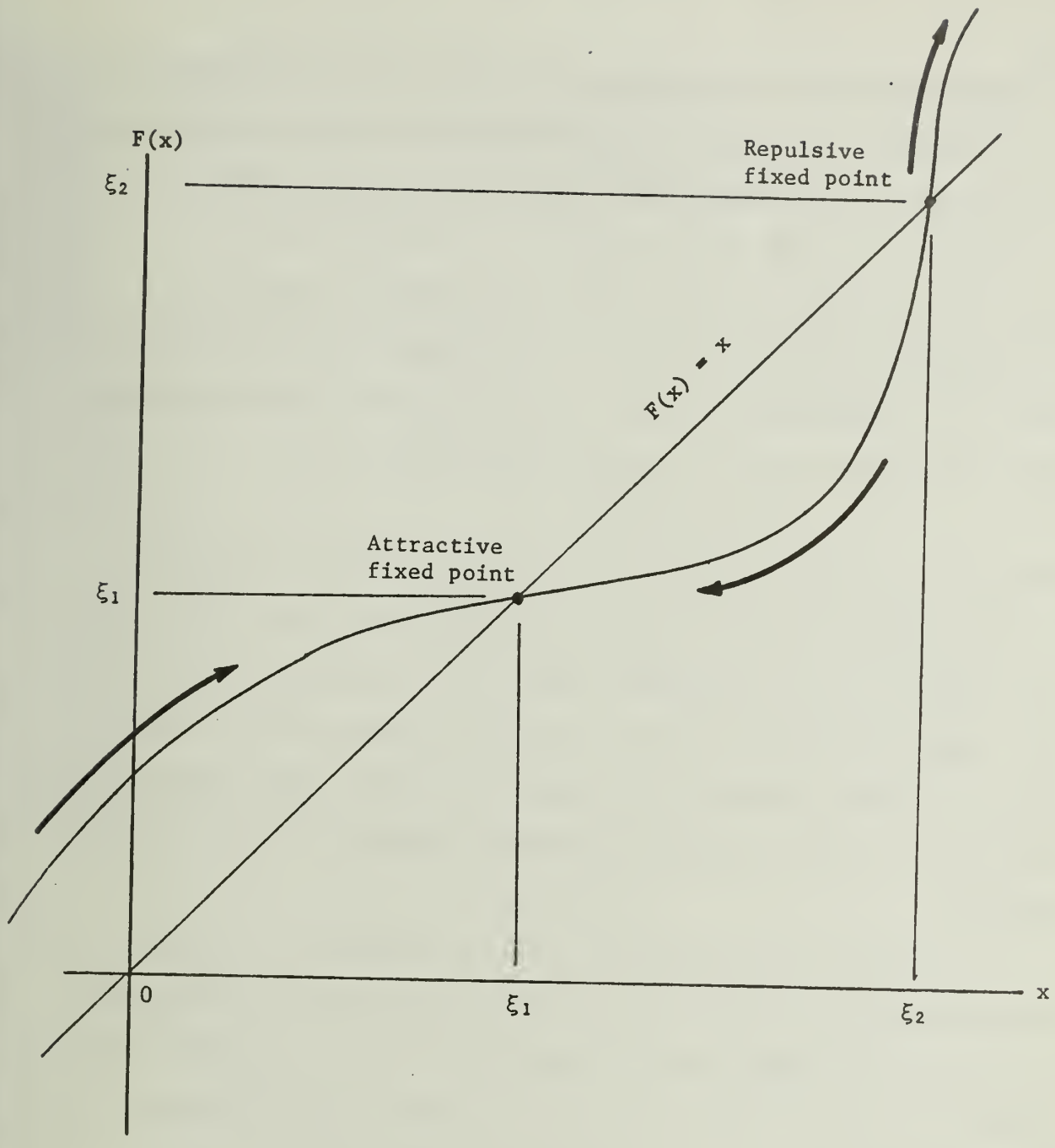


Figure 7. Iteration function F with fixed points ξ_1, ξ_2

($F^{[k]}$ denoting the k-fold composition of F), then it is a theorem that every attractive fixed point has an open attractive domain contining it, and that this attractive domain is a submodulus for F [Kuc 68,Thm 0.3].

We should point out that the theory of iteration of a real C^{∞} function (the problem we will be studying) is embedded inextricably in the corresponding theory for analytic functions defined on the complex plane, and a thorough understanding of the former necessitates knowledge of the latter. For example to explain why the function

$$F(x) = 1/(1-x)$$

satisfies $F(F(F(x))) = x$ requires us to note that F has fixed points at $(1 \pm i\sqrt{3})/2$ and that F's derivatives at these points are primitive third roots of unity. Note however, that if ξ is a complex number then $A_F(\xi)$ cannot contain any real point since F(x) is always real if x is. Thus it is possible to study real iteration in its own right, though a greater understanding of what is going on will require study of complex variables.

We will always be concerned with real iterations near attractive fixed points, since it turns out that these are the iterations that can be linearized by finding changes of variables. It is very possible that a general iteration could get in a cycle of length m, such that $x_{k+m} = x_k$ for all suitably large k. Such cycles would tend to proliferate near indifferent fixed points and in areas where F is "noninvertible" (see below). Except when m=0 or 1 there is no way that this behavior can be transformed into a linear behavior, and consequently any linearization approach used is doomed to fail <u>a priori</u>. However, if we operate only in the attractive domains of attractive fixed points then something like

Schröder functions can be found and the problem can be locally linearized. Thus our whole approach is to find an invertible change of variables $\phi = \phi_\xi$ for each attractive domain $A_F(\xi)$ which linearizes F on that domain, i.e.,

(1) $$F(x) = \phi^{-1}(s\, \phi(x)) \quad \text{for } x \text{ in } A_F(\xi).$$

The iteration is to be run in the slow, obvious way to start with until an iterate x_k enters an attractive domain, and then the linearizing transform can be made and the iteration finished rapidly. Naturally, many iterations will be started in attractive domains.

Consequently, the results derived here are restrictive in that they can only be used in attractive domains. Equation (1) above implies a further restriction, however. To be computationally useful our change of variables ϕ must be invertible, but it is easily seen from (1) that <u>if F is linearizable then it too is invertible</u>. Therefore, assuming $F \in C^1[A_F(\xi)]$, F must be strictly monotone and we must have

(2) $$F'(x) \neq 0 \quad \text{on } A_F(\xi)-\{\xi\}.$$

Note that $F'(\xi) = 0$ is possible since it is reasonable that $\phi(\xi) = 0$ or $\pm\infty$; in fact if F is any superlinearly convergent iteration like Newton's method then we do have $F'(\xi) = 0$. However, by differentiating Schröder's equation $\phi(F(x)) = s\phi(x)$ we find

(3) $$\phi'(\xi)\, F'(\xi) = \phi'(\xi)\, s$$

which implies that, if ϕ' is defined and nonzero at ξ, we must have

(4) $$F'(\xi) = s$$

determining the constant s of the linearized map (1). Intuitively, in the linearly convergent case we are finding the map ϕ which "untwists" the nonlinear function F(x) into the linear function sx, and are solving our recurrence in the untwisted space.

Finally we assume for computational reasons that the functions F we are dealing with are C^∞ (hence representable by a power series) on $A_F(\xi)$. It is known that if $F \in C^r[A_F(\xi)]$ $(r \geq 1)$ is invertible with $F'(\xi) \neq 0$, then there exists a one-parameter set of C^r solutions ϕ of equation (1) [Kuc 68,Thms. 6.1-6.2]; and, if ϕ is C^r then clearly F must be also. In addition since we will want to compute ϕ using power series it will be expedient to assume a power series expansion for F is not only available at the fixed points but is convergent in neighborhoods of them also. Briefly, we are assuming that F is a real analytic function which is invertible around its fixed points, or more concisely, that F is a C^∞-morphism on all domains of interest to us. Of course, these regularity assumptions on F could be replaced for computation purposes by the supposition that F can be well approximated by C^∞-morphisms on specific domains.

With these assumptions we can prove the existence of a real analytic change of variables function satisfying something like the Schröder equation (1) on an attractive domain. We use the hedge "something like" since we will not use the Schröder equation in all cases. To do so would lead to problems: observe that if we use the natural equation (4) above and plug it into equation (1) when $F'(\xi)=0$ or $F'(\xi)=1$ we produce unintelligent results. It turns out that these problems can be circumvented by not using (4) and by tolerating singularities in the change of variables function at the fixed point -- that is, one can always find a solution ϕ to the Schröder equation at an attractive fixed point (or indifferent fixed point with nontrivial attractive domain) which is

analytic except possibly at the fixed point itself. (Novice reader of [Kuc 68] beware: this point is not very clearly made.) However it is more convenient to simply shift to Böttcher's equation

$$\beta(F(x)) = \beta(x)^{\mu} \tag{5}$$

when $F'(\xi) = 0$, and to Abel's equation

$$\alpha(F(x)) = \alpha(x) + 1 \tag{6}$$

when $|F'(\xi)| = 1$ because the solutions to these equations are easier to handle near the fixed point than the Schröder solution. It is easy to see that Schröder's, Böttcher's, and Abel's equation are all equivalent in that an invertible, analytic solution of one leads to an invertible analytic solution of another (except at the fixed point). The equivalence of these and several other related functional equations is clarified in Figure 8. Note that all of these equations are equally potent in evaluating iterations quickly.

Before we actually state our result on the existence of linearizing changes of variables, we make the following final assumption. For simplicity we can assume that the fixed point ξ (in whose attractive domain we are determining ϕ) is equal to 0. That we can do this without any loss of generality lies in the following observation: Given F(x) with fixed point $\xi \neq 0$, we replace it with

$$G(x) = \tau(F(\tau^{-1}(x))) \tag{7}$$

where $\tau(x) = x + \xi$ if ξ is finite and $\tau(x) = 1/x$ otherwise. Then it is easy to see that G has fixed point 0 and if we find ψ such that

$$G(x) = \psi(s\ \psi^{-1}(x))$$

then clearly $F(x) = \phi(s\ \phi^{-1}(x))$ with ϕ defined by

$$\phi(x) = \tau^{-1}(\psi(x)).$$

Also if ξ is finite we find the useful relationship

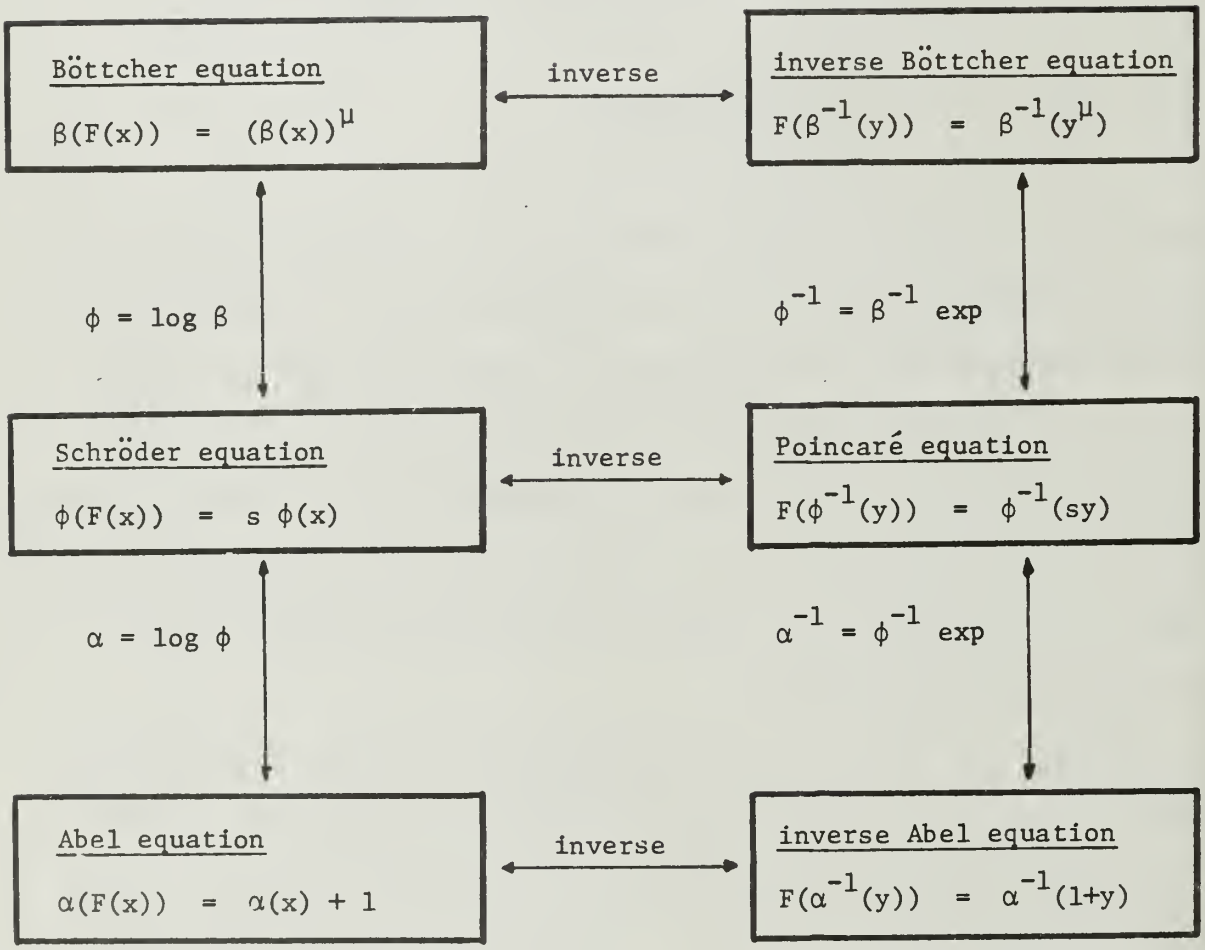


Figure 8. Equivalences between functional equations

(8) $$G'(x) = F'(x+\xi),$$

so G's derivatives are the same as F's at its fixed point. This puts us finally in a position to prove the following theorem.

Theorem Given a function F which is analytic and invertible on the attractive domain $A_F(0)$ of the attractive or indifferent fixed point 0, there exists a change of variable function α, β, or ϕ such that

$$F(x) = \begin{cases} \beta^{-1}((\beta(x))^{\mu}) & \text{if } F'(0) = 0 \\ \phi^{-1}(s\,\phi(x)) & \text{if } 0 < |F'(0)| < 1 \\ \alpha^{-1}(\alpha(x) + 1) & \text{if } |F'(0)| = 1 \end{cases}$$

for x in $A_F(0)$.

Proof

The proof is broken down into the different cases determined by $|F'(0)|$:

Case 1 $0 < |F'(0)| < 1$.

In this case we set the Schröder constant $s = F'(0)$ (so s is positive iff F is strictly increasing). This case has been well-understood since Koenigs analyzed it in 1884 [Koe 1884], and is known in the literature as the "regular case". Koenigs proved that for any analytic F there exists a one-parameter family of solutions ϕ of (1) which are analytic throughout the attractive domain $A_F(0)$ [Kuc 68,pp.139-141], given formally by

$$\phi_c(x) = c \lim_{n\to\infty} F^{[n]}(x)/s^n.$$

Power series for a particular ϕ may be easily derived as in [LL 59,pp.131-2] using the power series coefficients of F. If we write

$$F(x) = sx + \sum_{m=2}^{\infty} a_m x^m$$

then the Poincaré equation (inverse Schröder equation) $F(\phi^{-1}(y)) = \phi^{-1}(sy)$ leads to

$$\phi^{-1}(y) = y + \sum_{m=2}^{\infty} c_m y^m$$

where

$$c_2 = a_2 / (s^2 - s)$$

$$c_3 = a_3 / (s^3 - s) + 2a_2 / ((s^3 - s)(s^2 - s))$$

$$c_4 = a_4 / (s^4 - s) + (3s+5)\, a_3 a_2 / ((s^4 - s)(s^3 - s)) + (s+5)\, a_2^3 / ((s^4 - s)(s^3 - s)(s^2 - s))$$

and so forth. Once an expansion for ϕ^{-1} has been derived, the series for ϕ is easily obtained by just "reverting" the series for ϕ^{-1}; we have, corresponding to the expansion above,

$$\phi(x) = x + \sum_{m=2}^{\infty} d_m x^m$$

where

$$d_2 = -c_2$$

$$d_3 = 2c_2^2 - c_3$$

$$d_4 = 5c_2c_3 - c_4 - 5c_2^3$$

$$d_5 = 6c_2c_4 + 3c_3^2 + 14c_2^4 - c_5 - 21c_2^2c_3$$

etc. Again, this expansion for ϕ may be derived by comparing power series coefficients (in this case, of $\phi(\phi^{-1}(x)) = x$). However, the formulas one gets grow very complicated very quickly. Better computational methods for producing these series have been developed by Brent, Kung, and Traub [BK 76],[KT 78],[TB 78]. These methods will be discussed below in section IV.

Inspection of equation (1) shows that $\phi(x)$ can be replaced by $c \cdot \phi(x)$ for any nonzero constant c without disturbing these results (c is the parameter of the one-parameter of the one-parameter family of solutions ϕ; above we arbitrarily chose solutions with leading coefficient one). Koenigs' theorem guarantees not only the existence of ϕ but also that it is convergent for some nonzero x, although convergence on all of $A_F(0)$ is not guaranteed. The power series expansion for $F^{[k]}(x)$ for any fixed k obtained by forming the composition $\phi^{-1}(s^k \phi(x))$ should be convergent -- but again, note that computational problems with significance can arise if $|s|$ is either very close to zero or very close to one, as should be evident from the coefficient formulas.

Case 2 $F'(0) = 0$.

This is known in the literature as a singular case, with multiplier zero. What is indicated is that F is very flat in a neighborhood of zero, so the iteration's convergence to zero will be rapid (superlinear) there. Clearly the approach used in Case 1 will not work since setting $s = F'(0)$ does not provide us with anything useful.

One way to show the existence of a C^1 change of variables is to reduce this case to Case 1 by the following artifice of Szekeres [Sze 58, p.215],[Kuc 68,p.146]. We assume here that

$$F(x) = x^{\mu} A(x) = x^{\mu} (a_0 + a_1 x + a_2 x^2 + \ldots)$$

where μ is a positive integer greater than 1 and a_0 is nonzero. (This same technique works when μ is any nonzero real number, like 1/2, but F is not analytic at zero unless μ is a positive integer.) Write

$$F^*(x) = \frac{1}{-\log(F(\exp(-1/x)))} = \frac{1}{\mu/x - \log(A(\exp(-1/x)))}$$

which is a transformation of the form (7) above but with $\tau(x) = 1/\log(1/x)$. We verify that $F^*(0) = 0$, and importantly

$$\lim_{x\to 0+} \frac{d}{dx} F^*(x) = \lim_{x\to 0+} \frac{d}{dx} \left(\frac{x}{\mu} \Big/ \left(1 - \frac{x}{\mu} \log A(e^{-1/x})\right)\right) = \frac{1}{\mu}.$$

Since $0 < |1/\mu| < 1$, we know by reduction that Case 1 that F^* has a Schröder function ψ satisfying

$$F^*(x) = \psi^{-1}\left(\frac{1}{\mu}\psi(x)\right).$$

Therefore if we put $\phi(x) = \psi(1/\log(1/x))$ we obtain the Schröder solution

$$F(x) = \phi^{-1}\left(\frac{1}{\mu}\phi(x)\right);$$

moreover we can show formally that

$$\phi(x) = \lim_{n\to\infty} \log(1/F^{[n]}(x)) \Big/ \mu^n .$$

It must be pointed out that the function F^* above is <u>not</u> analytic at zero -- in fact it is very nonanalytic, as can be seen by approaching zero from both sides on the real line -- and our "reduction" to Case 1 relies on the fact that if F is C^r then there exists a C^r Schröder function satisfying (1), for any $r > 0$ [Kuc 68,p.137]. Thus ψ is not necessarily analytic and neither is ϕ. It is important to notice that ϕ cannot be differentiable or even continuous at zero here. If it were differentiable at zero, then equation (3) would give us $s = F'(0) = 0$. Kuczma repeatedly asserts that the only continuous solution of (1) is $\phi \equiv 0$ in this case, which is confusing until one realizes he assumes that ϕ is defined at ξ.

The problems with using Schröder's equation in this case are now apparent. We circumvent them by solving Böttcher's equation (5) instead. Analogous to Case 1, Levy and Lessman have shown that the inverse Böttcher equation $F(\beta^{-1}(y)) = \beta^{-1}(y^\mu)$ for the (popular) case $\mu = 2$ can be solved

by equating power series coefficients to yield

$$\beta^{-1}(y) = \sum_{m=1}^{\infty} c_m y^m$$

where

$$c_1 = 1/a_0$$

$$c_2 = -a_1 / (2a_0^3)$$

$$c_3 = 5a_1^2 / (8a_0^5) - a_2 / (2a_0^4) - a_1 / (4a_0^3)$$

and so on [LL 59,p.133]. Computational methods for deriving β^{-1} to any desired degree are given below in section IV for all $\mu \geq 2$. Again once β^{-1} has been discovered, $\beta(x)$ can be derived straightforwardly by reversion as in Case 1. The nice thing about Böttcher's solution is that β is an analytic function and does not have the logarithmic singularity at zero that the Schröder function $\phi(x) = \log \beta(x)$ does. This was the whole purpose of changing functional equations.

Case 3 $|F'(0)| = 1$.

This is referred to as a singular case, with multiplier unity, and is the hardest of the three cases to deal with. Here zero is an indifferent fixed point, but we assume $A_F(0)$ is a nontrivial attractive domain, so if $F'(0) = +1$ we would expect $F(x) < x$ for x in some interval $(0,c)$. For example $F(x) = \sin(x)$ satisfies $F(0) = 0$, $F'(0) = 1$, $F(x) = x - O(x^3) < x$, and always produces an iteration converging to zero. The convergence for functions in this case is extremely slow (sublinear) when it exists, however -- the sine iteration produces iterates x_k which eventually decrease to $\sqrt{3}/k\ (1 + O(\log(k)/k))$, irrespective of the starting position [Mel 73],

[deB 61]. This convergence rate is evident from the fact that $|F(x)| \simeq |x|$ for small $|x|$, hence $|x_{k+1}| \simeq |x_k|$ when $|x_k|$ is small.

In the literature F is normally taken to be a complex function, and $|F'(0)| = 1$ has a number of possible ramifications depending on the precise character of the complex number $F'(0)$ (see [Kuc68,ch.VI§§7-10]). Here we have simply $F'(0) = \pm 1$ since we are concentrating on real functions. Note if $F'(0) = -1$, however, then $F^{[2]}(x) = F(F(x))$ satisfies $F^{[2]'}(0) = +1$, so with a small amount of work we can restrict our attention to the case $F'(0) = +1$ and assume

$$F(x) = x - ax^{\mu+1} + \sum_{m=\mu+2}^{\infty} a_m x^m \qquad (\mu \geq 2). \tag{9}$$

Now it is easy to show by direct substitution that the only analytic functions F having analytic solutions ϕ to the Schröder equation (1) are functions which satisfy

$$F^{[p]}(x) = x$$

for some positive integer p [Kuc 68,p.147], i.e., functions which are equivalent to the identity when forward-substituted p times. If p is small this is a result of purely negative use to us, since it says a Schröder change of variables can be found only when we wouldn't need it. Additionally it is very rare for a function to satisfy $F^{[p]}(x) = x$: Kung's results [Kung 76] show that the only rational functions having this property are those of first degree, i.e., of the form

$$F(x) = \frac{ax + b}{cx + d}.$$

(Observe that $F(x) = x/(x-1)$ satisfies $F^{[2]}(x) = x$, and Boole's function $F(x) = 1/(1-x)$ [Boo 70,pp.292-3] satisfies $F^{[3]}(x) = x$. Boole derives general conditions on a,b,c,d for F to satisfy $F^{[p]}(x) = x$ [Boo 70,pp.298-9].)

Because of the difficulty in obtaining a change of variables function the normal shift at this point is to move away from Schröder's equation to Abel's equation (6). Regrettably the Abel solution $\alpha(x)$ cannot be analytic at zero either, as (6) should make clear, but its behavior there can be precisely studied. In fact α has a pole of order μ (cf. (9)) at the origin, and possibly other lesser singularities as well. Otherwise $\alpha(x)$ is analytic for $x > 0$, behaves like $-1/(\mu a x^{\mu})$ near zero, is unique up to an additive constant, and if y is any point in $A_F(0)$ then α can be expressed as [Sze 58,p.218]

$$\alpha(x) = \lim_{n\to\infty} \left(a^{1/\mu} (\mu n)^{1+1/\mu} (F^{[n]}(x) - F^{[n]}(y)) \right).$$

In spite of this information it is very difficult in general to derive precise expressions for $\alpha(x)$. There are special cases: when

$$F(x) = x/(1+x) = x - x^2 + x^3 - \ldots$$

one can show that $\alpha(x) = (x-1)/x$, $\alpha^{-1}(x) = 1/(1-x)$ satisfy (6); but here

$$F^{[n]}(x) = x / (1 + nx)$$

for all x, so we do not really need changes of variables. DeBruijn [deB 61] develops at length the leading terms of $\alpha(x)$ for the sine iteration. Methods for dealing with the difficulty of handling $\alpha(x)$ are discussed below in section IV, but these methods consist only in deriving a series for $F^{[n]}(x)$. A general procedure for deriving explicit changes of variables is not known for this case.

This concludes the discussion of the theorem and the theoretical background of the problem. There are a number of other interesting related topics that will not be touched on here, such as fractional iteration (how does one evaluate $F^{[1/2]}(x)$? etc.). The reader looking for more information on this, on the complex case, or on higher-order iterations is referred to [Kuc 68].

III. Special Recurrence Forms

In this section we describe three classes of nonlinear recurrences which cover almost all the examples of linearizable iterations known to the author. Examples described herein have been compiled from [Boo 70], [Kuck74], [Kuc 68], [LL 59], [Mel 73], [M-T 51], [Par 77], [Sch 1871], and other lesser sources. Some of the recurrences are not first order and do not fit directly in the theoretical discussion above, but are included here for completeness. Almost all nonlinear recurrences

$$x_k = F(x_{k-1}, \cdots, x_{k-m})$$

which are linearizable in the sense being discussed in this chapter satisfy the quasilinearity property

(10) $$F(v_1,\ldots,v_m) = \phi^{-1}(L(\phi(v_1), \cdots, \phi(v_m)))$$

where L is a linear function and ϕ is an invertible map on the domain of iteration. We could close the section at this, but there are certain maps ϕ which produce interesting forms for F which bear mentioning, and there are some recurrences -- discussed in the third section -- for which completely different techniques are successful.

1. Simple Quasilinear Recurrences

Provided L is a linear function, any recurrence of the form (10) can be linearized with the change of variables $y_k = \phi(x_k)$. For example taking $\phi(x) = x^2$ we see

$$x_k = \sqrt{x_{k-1}^2 - x_{k-2}^2 + 2x_{k-3}^2}$$

is linearizable, and if $\phi(x) = \log(x)$ then so is

$$x_k = 2\, x_{k-1}\, x_{k-2}^2 \,/\, x_{k-3} .$$

There are many interesting forms when ϕ is a simple map. Taking $\phi(x) = \log(1+x)$ and $L(x) = 2x$, we find

$$F(x) = x^2+2x \quad \Longrightarrow \quad F^{[n]}(x) = (x+1)^{2^n} - 1.$$

If $\phi(x) = (1 + \log(x))/(1 - \log(x))$ and $L(x) = 3x$ then

$$F(x) = x(x^2+3)/(3x^2+1) \quad \Longrightarrow \quad F^{[n]}(x) = (1+x^{3^n})/(1-x^{3^n}).$$

In general, the idea here is that a judicious set of functions ϕ composed of exponentials, logs, and rational transformations will serve to generate many interesting rational functions F. Perhaps the most simple class of such functions are the linear fractional transformations

$$F(x) = \frac{ax + b}{cx + d}$$

where a,b,c,d are constants; this class arises from choosing ϕ itself to be a linear fractional transform. Assuming F is non-degenerate (so $ad-bc \neq 0$, $c \neq 0$) we can get the following closed form for its iterates. From

$$F(x) = \frac{ax + b}{cx + d} = \frac{a}{c} - \frac{ad - bc}{c(cx+d)}$$

we obtain

$$F^{[n]}(x) = \frac{(r_1 + d/c)^n(r_1x + b/c) - (r_2 + d/c)^n(r_2x + b/c)}{(r_1 + d/c)^n(x - r_2) - (r_2 + d/c)^n(x - r_1)}$$

where r_1 and r_2 are the roots of $F(r) = r$. This may also be written as

$$F^{[n]}(x) = u \tanh\left(\operatorname{arctanh}\left(\frac{x-v}{u}\right) + nw \right) + v$$

where $u = \frac{\sqrt{(a-c)^2 + 4bc}}{2c}$, $v = \frac{a - d}{2c}$, and $w = \operatorname{arctanh}\left(\frac{2cu}{a+d} \right)$.

If $r_1 = r_2 = r$ we get the special form

$$F^{[n]}(x) = \frac{(rx + b/c)n + (r + d/c)x}{(x - r)n + (r + d/c)} .$$

To reiterate, to reiterate, there are <u>many</u> simple quasilinear recurrences. It is impossible to give a complete enumeration here, since any list could be extended indefinitely just by repeatedly selecting new maps ϕ and conjugating the list with respect to them. Probably the best method for determining whether a simple map ϕ exists for any given F is to go ahead and derive the Schröder function corresponding to F (at least the first few terms in its power series expansion) using the methods described in section IV. Even if closed form for the change of variables cannot be gleaned from its power series, its general behavior (exponential, logarithmic, rational) can be, and this information used to make more intelligent guesses about its nature.

2. Special Forms arising from Algebraic Addition Theorems

From the identity

$$(\sin 2y)^2 = 4(\sin y)^2(1 - (\sin y)^2)$$

we notice that the recurrence $x_{k+1} = 4x_k(1 - x_k)$ can be transformed to $y_{k+1} = 2y_k$ under the change of variables $(\sin y_k)^2 = x_k$, provided that $0 \leq x_0 \leq 1$. It follows that

$$x_k = (\sin(\, 2^k \arcsin(\sqrt{x_0})\,))^2$$

for any k in this iteration. Similar identities give the following list:

F(x)	$F^{[n]}(x)$	comments
$4x(1-x)$	$(\sin(\, 2^n \arcsin(\sqrt{x})\,))^2$	if $0 < x_0 < 1$
	$-(\sinh(2^n \operatorname{arcsinh}(\sqrt{-x})\,))^2$	if $x_0 < 0$ (equivalent to above)
	$-(\sinh(2^{n-1} \operatorname{arcsinh}(\sqrt{-4x(1-x)})\,))^2$	if $x_0 > 1$
$4x(1+x)$	$(\sinh(2^n \operatorname{arcsinh}(\sqrt{x})\,))^2$	separate cases similar to example above
$2x^2-1$	$\cos(\, 2^n \arccos(x)\,)$	if $x^2 < 1/2$
	$\cosh(2^n \operatorname{arccosh}(x)\,)$	if $x^2 > 1/2$
$4x^3 + 3x$	$\sinh(3^n \operatorname{arcsinh}(x))$	
$4x^3 - 3x$	$\cos_{\cosh}(3^n \operatorname{arc}^{\cos}_{\cosh}(x))$	
$3x - 4x^3$	$\sin(3^n \arcsin(x))$	
$2x/(1-x^2)$	$\tan(2^n \arctan(x))$	
$2x/(1+x^2)$	$\tanh(2^n \operatorname{arctanh}(x))$	
$(x^2-1)/2x$	$\cot(2^n \operatorname{arccot}(x))$	
$(x^2+1)/2x$	$\coth(2^n \operatorname{arccoth}(x))$	
$(x^2+A)/2x$	$\sqrt{A}\coth(2^n \operatorname{arccoth}(x/\sqrt{A})$	Newton-Raphson square-root iteration

This list can also be extended indefinitely by examining things of the form $F(x) = t(m\ t^{-1}(x))$, where t is a trigonometric function. When m is an integer the result is typically a rational function. For example the Chebyshev polynomials

$$T_m(x) = \cos(\ m \arccos(x)\ )$$

enjoy the relationships

$$T_m{}^{[n]}(x) = T_{m^n}(x) \quad \text{and} \quad -iT_m{}^{[n]}(ix) = -iT_{m^n}(ix).$$

Note $-iT_m(ix) = \cosh(\ m \operatorname{arccosh}(x)\ )$ and the first few values of T_m are

$$T_2(x) = 2x^2 - 1$$

$$T_3(x) = 4x^3 - 3x$$

$$T_4(x) = 8x^4 - 8x^2 + 1.$$

Also if we set

$$S_m(x) = \left[\frac{\sum_{k=0}^{\infty} (-1)^k\ (m)_{2k+1}\ x^{2k+1}}{\sum_{k=0}^{\infty} (-1)^k\ (m)_{2k}\ x^{2k}} \right] = \tan(\ m \arctan(x)\ )$$

where $(m)_p = (m)(m-1)\ldots(m-p+1)$ and $(m)_0 = 1$, then we can derive more recurrence forms for $F(x) = S_m(x)$ and $-i\ S_m(ix)$. We find

$$S_2(x) = 2x/(1-x^2)$$

$$S_3(x) = (3x-6x^3)/(1-6x^3).$$

Of course, further rational forms can be obtained by conjugating this list with invertible rational functions. Schröder goes on, for example, to compute $F^{[n]}(x)$ when

$$F(x) = \phi^{-1}(\ 2\ \phi(x)\ /\ (1 + (\phi(x))^2)\ )$$

like the tangent transform, but where ϕ is an arbitrary linear fractional transformation. It is clear that F(x) will be a rational function of degree two with three degrees of freedom in its coefficients.

The above results are all derivative of what are known as addition theorems for complex functions. A function ϕ is said to have an algebraic addition theorem if there exists a polynomial $P(x,y,z)$ such that, for all u and v,

$$P(\phi(u),\phi(v),\phi(u+v)) \equiv 0.$$

Weierstrass showed that the only functions $\phi(u)$ capable of having an algebraic addition theorem are algebraic functions of u, of $\exp(i\pi u/\omega)$, or of the Weierstrass elliptic function $\wp(u|\omega_1,\omega_2)$, where ω, ω_1 and ω_2 are suitable periodicity constants [Mel 73,p.56],[For 18,ch.XIII]. The trigonometric functions are degenerate elliptic functions. One can derive further results from the elliptic functions themselves. For example, since

$$sn(2u) = 2sn(u)cn(u)dn(u) \;/\; (1 - m\, sn(u)^4)$$

where $sn(u|m)$, $cn(u|m) = \sqrt{1 - sn^2(u)}$, $dn(u|m) = \sqrt{1 - m\, sn^2(u)}$ are Jacobian elliptic functions (parametrized by m, with $0 \leq m \leq 1$), squaring both sides of the identity gives us that

$$F(x) = \frac{4x(1-x)(1-mx)}{(1-mx^4)}$$

satisfies

$$F^{[n]}(x) = (sn(\, 2^n\, sn^{-1}(\sqrt{x} \mid m) \mid m)\,)^2.$$

Finally, other types of recurrences may also be solvable using trigonometric substitutions. Consider the second-order recurrence

$$x_{k+2} = A\,(x_{k+1} + x_k) \;/\; (x_{k+1}\, x_k - A)$$

having the difference-equation format $x_{k+2}x_{k+1}x_k = A(x_{k+2}+x_{k+1}+x_k)$. Then putting $y_k = \sqrt{|A|}\,\tan(x_k)$ we obtain the linear relationship

$$y_{k+2} + y_{k+1} + y_k = \arcsin(0) = n\pi.$$

Interestingly, Milne-Thomson shows [M-T 51,p.431] that recurrences of this same difference-equation format, but extended in the obvious way to any order, have an explicit solution involving primitive roots of unity.

3. Trading off recurrence order for linearity

When we discussed fractional linear transformations above it was taken for granted that the coefficients in the (first-order) iteration function F were constant. If instead we have the non-constant-coefficient iteration

$$x_{k+1} = F_k(x_k) = \frac{a_k x_k + b_k}{x_k + d_k} \tag{11}$$

then the change of variables approach as dictated above will fail (miserably). Consider however setting

$$x_k = u_k / v_k$$

where the u's and v's are new variables. Substituting this expression for x in (11) produces

$$\frac{u_{k+1}}{v_{k+1}} = \frac{a_k u_k + b_k v_k}{u_k + d_k v_k}$$

which can be divided into the first-order, coupled linear system

$$\begin{aligned} u_{k+1} &= a_k u_k + b_k v_k \\ v_{k+1} &= u_k + d_k v_k \,. \end{aligned} \tag{12}$$

This is a derivation of the fact that compounded fractional linear transforms can be represented as matrix products:

$$\begin{bmatrix} u_{k+1} \\ v_{k+1} \end{bmatrix} = \begin{pmatrix} a_k & b_k \\ 1 & d_k \end{pmatrix} \cdots \begin{pmatrix} a_1 & b_1 \\ 1 & d_1 \end{pmatrix} \begin{pmatrix} a_0 & b_0 \\ 1 & d_0 \end{pmatrix} \begin{bmatrix} x_0 \\ 1 \end{bmatrix}.$$

Matrix multiplication being associative, the value x_k can be rapidly evaluated on a parallel machine. (Note also that a reformulation of this can be used to find a fast parallel algorithm for first-order linear iteration.)

Surprisingly, there is another transformation one can make here. Setting

$$x_k = y_{k-1} / y_k + a_{k-1}$$

where the y's are new variables, and substituting for x_k in (11), gives the second-order linear recurrence

$$y_{k+1} = \frac{(a_{k-1} + d_k)}{(b_k - a_k d_k)} y_k + \frac{1}{(b_k - a_k d_k)} y_{k-1} . \tag{13}$$

This recurrence can be solved rapidly in parallel as long as $(a_k d_k - b_k) \neq 0$ for each k, i.e., as long as each application of F_k is non-degenerate as a fractional linear transformation. Taking the boundary conditions $y_0 = 1$, $y_1 = (x_0+d_0)/(b_0-a_0 d_0)$, we can also write this recurrence as a matrix product

$$\begin{bmatrix} y_{k+1} \\ y_k \end{bmatrix} = \begin{pmatrix} (a_{k-1}+d_k)/\delta_k & 1/\delta_k \\ 1 & 0 \end{pmatrix} \cdots \begin{pmatrix} (a_0+d_1)/\delta_1 & 1/\delta_1 \\ 1 & 0 \end{pmatrix} \begin{bmatrix} y_1 \\ y_0 \end{bmatrix}$$

where $\delta_k = b_k - a_k d_k$ for all k.

Except for a few more recurrences to be included below, this discussion should be closed at this point since it is already far beyond the scope of the rest of this chapter. However, because the type of recurrences here are of interest to algorithm designers and compiler writers (of more interest, probably, than first-order constant-coefficient iteration) we digress momentarily and present a theory which may be of use in linearizing recurrences of this kind. The area is especially interesting because it provides another attack on nonlinear recurrences that is not subject to the negative results of Kung [Kung76].

In both of the examples above, a first-order nonlinear recurrence was changed to a linear one by expanding its order with the introduction of auxiliary variables. Expansion of order is a little-studied technique in mathematics, since normally one is interested in just the opposite. Reduction of order is often accomplished by detecting invariants in the recurrence system and changing variables in such a way that a dimension of the system is annihilated. For example the arithmetic-harmonic mean iteration

$$a_{k+1} = \frac{1}{2}(a_k + h_k)$$
$$h_{k+1} = 2a_k h_k/(a_k + h_k) = 1/\left(\frac{1}{2}\left(\frac{1}{a_k} + \frac{1}{h_k}\right)\right) \tag{14}$$

satisfies the invariant $a_k h_k = a_0 h_0$ for all k. If $a_0 h_0 = A$ then the iteration converges to $\sqrt{A}$, and one can show by induction that in this case the Newton-Raphson square root iteration

$$x_{k+1} = \frac{1}{2}\left(x_k + \frac{A}{x_k}\right) \tag{15}$$

is equivalent to the arithmetic-harmonic mean, in the sense that $x_k = a_k = A/h_k$ for all k. Thus the Newton-Raphson iteration is a reduction of the arithmetic-harmonic mean.

What we wish to accomplish here is in some sense the inverse operation of reduction, but with the goal of producing a linear recurrence. There are at least two possible expansion strategies: expansion of an m^{th}-order nonlinear recurrence into a coupled system of ℓ m^{th}-order linear recurrences with $\ell \geq 2$, or expansion of the nonlinear recurrence into an $(m+\ell)^{th}$-order linear recurrence using a different family of iterates. The fractional linear transformation problem above exhibits the use of both strategies. We describe the theory behind each one separately.

With the coupled system strategy one takes the recurrence

$$x_{k+1} = F_k(x_k)$$

and finds an expansion function Ψ of ℓ argument variables u,v,... such that substitution of

$$x_k = \Psi(u_k, v_k, \dots)$$

into the recurrence definition produces

(16) $$F_k(\Psi(u_k, v_k, \dots)) = \Psi(L_1(u_k, v_k, \dots), L_2(u_k, v_k, \dots), \dots)$$

where $L_1, L_2, \dots, L_\ell$ are linear functions. Then since $x_{k+1} = \Psi(u_{k+1}, v_{k+1}, \dots)$, the nonlinear recurrence has been expanded into the linear system

$$u_{k+1} = L_1(u_k, v_k, \dots)$$

$$v_{k+1} = L_2(u_k, v_k, \dots)$$

$$\vdots$$

The fractional linear transformation example (12) showed that $\Psi(u,v) = u/v$ linearizes (11) since

$$F_k(\Psi(u,v)) = \Psi(a_k u + b_k v,\ u + d_k v).$$

The system strategy generalizes in the obvious way for the m^{th}-order iteration

$$x_k = F_k(x_{k-1}, x_{k-2}, \dots, x_{k-m});$$

each x_{k-i} is replaced by $\Psi(u_{k-i}, v_{k-i}, \dots)$ and one seeks to produce a system of ℓ $\underline{m}^{th}$-order linear equations defining $u_k, v_k, \dots$ in terms of their predecessors.

The intent of the system strategy is therefore to solve the functional equation (16). The difficulty of finding a solution will rest on the form of F -- for a good general reference on functional equations, see [Acz 66].

Two techniques that may be useful in determining Ψ are differentiating (16) with respect to different variables (note the assumption that Ψ is differentiable) and checking its form for different variable values. For example, suppose we were trying to find an expansion $x_k = \Psi(u_k,v_k)$ for the Newton-Raphson iteration (15) satisfying the functional equation

$$F(\Psi(u_k,v_k)) = \Psi(au_k+bv_k, cu_k+dv_k),$$

for some constants a,b,c,d. From (15) we find

$$\frac{1}{2}(\Psi(u,v) + A/\Psi(u,v)) = \Psi(au+bv, cu+dv).$$

However, when we set $u = v = 0$ we get

$$\Psi(0,0) = \sqrt{A}.$$

This suggests that, even if we could find an expression for $\Psi(u,v)$, it would not be very useful computationally (since it seems to require knowledge of the value of the square root it should help derive). Finding a linearizing for the Newton-Raphson iteration which does not use square roots seems very difficult, as we saw in section III.2 and will see below in section IV.

The <u>different family strategy</u> seeks to convert the nonlinear iteration with iterates $\{x_k\}$ into a linear one with iterates $\{y_k\}$, where changes of variables are given by

$$x_k = \Psi_k(y_k,y_{k-1}, \cdots ,y_1,y_0)$$

$$y_k = \Phi_k(x_k,x_{k-1}, \cdots ,x_1,x_0).$$

The intent is that the relationship

$$x_{k+1} = \Psi_{k+1}(y_{k+1},y_k, \cdots ,y_0) = F_k(\Psi_k(y_k, \cdots ,y_0)) = F_k(x_k)$$

expresses a linear recurrence among the y's (possibly not a banded recur-

rence) and that

$$y_k = \Phi_k(\Psi_k(y_k,\ldots,y_0), \Psi_{k-1}(y_{k-1},\ldots,y_0), \ldots, \Psi_0(y_0))$$

holds for all k. In the fractional linear transform example (13) we took

$$x_k = y_{k-1}/y_k + a_{k-1} = \Psi_k(y_k,y_{k-1},\ldots,y_0).$$

Solving this recursively for y_k with the boundary conditions given above produces

$$y_k = 1 / \prod_{j=1}^{k} (x_j - a_{j-1}) = \Phi_k(x_k,x_{k-1},\ldots,x_0).$$

Restricted forms of the linear fractional transformation recurrence (11) give rise to other interesting changes of variables. If

$$x_{k+1} = x_k/(x_k + d_k)$$

we obtain the linear recurrence $y_{k+1} = d_k y_k + 1$ when we set $x_k = 1/y_k$, or formally

$$x_k = \Psi_k(y_k,y_{k-1},\ldots,y_0) = 1/y_k$$

$$y_k = \Phi_k(x_k,x_{k-1},\ldots,x_0) = 1/x_k.$$

If we have the <u>continued fraction iteration</u>

$$x_{k+1} = a_k + b_k/x_k$$

we get the second-order linear recurrence

$$y_{k+1} = a_k y_k + b_k y_{k-1}$$

with the change $x_k = y_k/y_{k-1}$, $y_k = \prod_{j=0}^{k} x_j$. Essentially this change of variables was used by Stone to produce his "recursive doubling" LU algorithm for solving tridiagonal systems in parallel [Sto 73], and by Sameh and Kuck for the parallel Givens reduction of tridiagonal matrices [SK 75]. Sameh and Kuck have also employed it in a parallel QR algorithm for symmetric tridiagonal matrices [SK 77].

It is difficult to say which of the two strategies presented here is "better", although it is clear that the solution produced by the different family approach can always be converted to a coupled system by changing the linear recurrence $y_k = L_k(y_{k-1}, y_{k-2}, \cdots, y_{k-m})$ to

$$\begin{bmatrix} y_k \\ y_{k-1} \\ \cdot \\ \cdot \\ \cdot \\ y_{k-m+1} \end{bmatrix} = \begin{pmatrix} \ell_{k,1} & \ell_{k,2} & \cdots & \ell_{k,m} \\ 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ & & & \\ 0 & 0 & \cdots 1 & 0 \end{pmatrix} \begin{bmatrix} y_{k-1} \\ y_{k-2} \\ \cdot \\ \cdot \\ \cdot \\ y_{k-m} \end{bmatrix} .$$

Thus in some sense the coupled system strategy is more general. Kuck has studied the use of the different family strategy in "continued parenthesis" and related recurrence forms [Kuck74]. He shows that putting $x_k = y_k y_{k-1}$ converts

$$x_k = a_k \frac{x_{k-1}\, x_{k-3} \cdots x_{k-2m+1}}{x_{k-2}\, x_{k-4} \cdots x_{k-2m}}$$

to $y_k = a_k / y_{k-2m-1}$, which can be linearized in turn like continued fractions. He goes on to show that systems like

$$x_k = y_{k-1}\,(a_k - y_{k-3}/x_{k-2})$$

$$y_k = x_{k-1}\,(b_k - x_{k-3}/y_{k-2})$$

can be changed into linear systems by setting $x_k = u_k v_{k-1}$, $y_k = v_k u_{k-1}$.

Other techniques may work depending on the context. If we examine

$$x_{k+1} = \frac{1}{2}\left(x_k - \frac{1}{x_k}\right) = (x_k^2 - 1)/(2x_k) \tag{17}$$

then we find that putting $x_k = u_k/v_k$ gives

$$\frac{u_{k+1}}{v_{k+1}} = \frac{u_k^2 - v_k^2}{2u_k v_k} .$$

This equation is very suggestive when broken into a coupled system on u_k and v_k, since it implies that creating the complex number

$$z = u_0 + iv_0 = x_0 + i$$

yields

$$u_k = \mathrm{Re}(\, z^{2^k} \,), \quad v_k = \mathrm{Im}(\, z^{2^k} \,)$$

because $(u+iv)^2 = (u^2-v^2) + i(2uv)$. Therefore

$$x_k = \mathrm{Re}(\, (x_0+i)^{2^k} \,) \,/\, \mathrm{Im}(\, (x_0+i)^{2^k} \,) = u_k/v_k.$$

This can be reexpressed in matrix form using the standard representation of complex numbers in 2x2 real matrices

$$z = u + iv \longmapsto \begin{pmatrix} u & v \\ -v & u \end{pmatrix}.$$

If we put

$$z = \begin{pmatrix} x_0 & 1 \\ -1 & x_0 \end{pmatrix}$$

then

$$z^2 = \begin{pmatrix} x_0^2-1 & 2x_0 \\ -2x_0 & x_0^2-1 \end{pmatrix}$$

and in general, if x_k is defined by (17) then

$$z^{2^k} = \begin{pmatrix} x_k^2-1 & 2x_k \\ -2x_k & x_k^2-1 \end{pmatrix}.$$

We have already shown in III.2 that x_k is exactly $\cot(2^k \operatorname{arccot}(x))$, but it is interesting to see that completely different methods can be used to obtain closed form for the same iterates. Note also that if we put

$$w = \begin{pmatrix} x_0 & 1 \\ 1 & x_0 \end{pmatrix}$$

then

$$w^{2^k} = \begin{pmatrix} x_k^2+1 & 2x_k \\ 2x_k & x_k^2+1 \end{pmatrix}$$

where x_k is defined by

$$x_{k+1} = \frac{1}{2}\left(x_k + \frac{1}{x_k}\right),$$

the square-root iteration (15) when A = 1. Again we know that x_k is $\coth(2^k \operatorname{arccoth}(x))$, but the elegant matrix form for the iterates is surprising.

Lastly we point out that some recurrences may behave linearly depending on the initial data. In some cases this will be obvious, since a particular initial value may zero out a nonlinear subexpression in the iteration. But linearity may sometimes be covert. J.L. Pietenpol (AMM 68, p.379, 1961) showed that

$$x_{k+1} = (1 + x_k x_{k-1})/x_{k-2}$$

is equivalent to the linear recurrence

$$y_{k+1} = 4y_{k-1} - y_{k-3}$$

with $x_k = y_k$ for all k, provided $x_0 = x_1 = x_2 = 1$ and $y_3 = x_3 = 2$. The proof is based on the fact that the y's then satisfy the invariant

$$y_{k+1}y_{k-2} - y_k y_{k-1} = 1,$$

i.e., $y_{k+1} = (1+y_k y_{k-1})/y_{k-2}$, as can be shown by induction.

In summary, we have exhibited two strategies that may be useful in solving general nonlinear recurrences, both based on the expansion of order of the original recurrence through the introduction of new variables. However the theory of linearization of general nonlinear recurrences of the type described in this section is still very much an open field. It remains to be established whether the strategies presented here are useful or whether there can be any useful general strategies, and also if there

exists a methodology for linearization. Beyond this, the numerical quality of the linearized recurrences must be studied -- for example, although the different family solution (13) for the fractional linear transformation iteration can be solved quickly in parallel, it will produce questionable results if any of the matrices $\begin{pmatrix} a_k & b_k \\ 1 & d_k \end{pmatrix}$ be near-singular, and also, as Sameh and Kuck pointed out for the continued-fraction special case [SK 77,p.152], there is a good possibility of overflow or underflow in computation of the y's. Although speed may be won by linearization, the price in accuracy may be too great to make the linear algorithm viable.

IV. Linearization Methods

In this section we discuss briefly how nonlinear recurrences can be linearized in a semi-automated way, permitting both algorithm designers and intelligent compilers to transform their problems into forms more efficiently solved on a parallel machine. The caveat must be made that a practical application of these methods has not yet been found: the convergent first-order recurrences known to the author do not require sufficiently many iterations to make the parallel approach championed here truly worthwhile, particularly if the recurrence is superlinearly convergent. However some almost-practical applications are described, and it is hoped that useful problems may someday be solved using the following methods.

Consider the linearization of the general first-order, constant-coefficient iteration $x_{k+1} = F(x_k)$ discussed above. A procedure for accomplishing this can be broken down into six steps:

(1) Determination of the fixed points ξ of F

(2) Expansion of the functions

$$G(x) = F(x-\xi)+\xi \quad (\text{or } 1/F(1/x) \text{ if } \xi = \pm\infty)$$

in a power series for each ξ

(3) Determination of attractive domains $A_G(0)$ of attractive fixed points

(4) Derivation of the appropriate change of variables function (ϕ or β) for each power series expansion

(5) Computation of the inverses of the change of variables functions

(6) Construction of the main procedure which detects when iterates enter attractive domains and finishes the recurrence accordingly.

It is clear that each of the above steps can be partially automated. Moreover, all of the steps can be entirely automated if the programs doing the work are made intelligent enough. There are, of course, many considerations involved in the design of an automated linearizer, but total automation seems an unlikely alternative: First of all, one generally knows in advance which attractive domain an iteration will take place in, so it makes little sense for the linearizer to evaluate all possible attractive domains and prepare changes of variables for each one. Second, the class of iteration functions F(x) is likely to be limited (say, to rational functions) and it seems unreasonable for the linearizer to get ready to expand all real analytic functions in power series, as well as find their fixed points. Nevertheless each of the five steps can be automated, and we outline how.

Step 1: Determination of fixed points

Fixed points of F may be rapidly determined by applying root-finding techniques to the function F(x)-x. Obviously knowledge of the form of F can lead to more efficient search for these roots. For example, if F is a polynomial then the roots may be found very rapidly.

Step2: Power Series expansion of F

Recall we are assuming F is analytic, so a power series exists. Thus the symbolic expression for F can be symbolically differentiated to any desired order using techniques like those in standard algebraic manipulation packages. The resulting derivative expressions can then be evaluated at the fixed points computed in Step 1, and the power series

for $G(x) = F(x-\xi) + \xi$ generated (using equation (8)). Again, knowledge of the form of F can produce savings in the complexity of the algebraic differentiator.

Step 3: Determination of attractive domains

The first derivative of G may be used to discern the attractive fixed points from the nonattractive ones. A root finder can then be used to hunt for zeroes of G' near these points: if found, a zero delimits the extent of the attractive domain on which G is invertible, and if not then the domain extends to the adjacent repulsive fixed point. Indifferent fixed points require some special treatment to ensure the existence of an attractive domain, but are otherwise treated identically.

Step 4: Derivation of the change of variables functions

There are a number of approaches to be used here depending on the accuracy required in the computation and the rate of convergence of F at the fixed point ξ. If $|G'(0)| < 1$ then the formulas of Levy and Lessman given in Cases 1 and 2 of the theorem in section II above can be used to produce a short power series for the change of variables (the formulas could possibly be extended to something like tenth order reasonably). This attack is straightforward but restricts the neighborhood of ξ in the attractive domain where the change of variables can be computed accurately.

Another approach in Case 1 or Case 2 is to compute as many terms of the change of variables as are necessary. That this can be done rapidly, given an equal number of terms in G's power series, has been

shown by Brent, Traub, and Kung ([BK 76],[KT 78], and especially [TB 78]). These researchers' papers give a number of useful, fast algorithms for the manipulation of power series -- and [TB 78] actually concerns itself with the solution of the Schröder and (pseudo-)Böttcher equations. We give an independently-developed method for solving the inverse Böttcher equation below, in the solution of one of the examples, which is similar to the Traub-Brent algorithm in its operation.

In the case where $|G'(0)| = 1$ there is no known general method for deriving a change of variables, as mentioned in Case 3 of the theorem above. However Traub and Brent show that in this case the power series for $G^{[n]}(x)$ can be derived rapidly from the power series for G for any fixed value of n (cf. [Kuc 68,ch.IX§6]). This may not be useful, depending on the requirements of the user, but if the number N of iterates required by the user is small, then it may be feasible to simply derive series for each function $F^{[n]}$; and if N is large, it is conceivable that these series could be derived at run time in execution on a parallel processor.

Step 5: Derivation of the inverse changes of variables

Once the power series for the change of variables functions are known, finding the series for the inverse functions is easily accomplished through the process of series reversion (discussed in Case 1 of the theorem above). Exact reversion formulas exist for low-order series; an algorithm based on Newton's method is given in [BK 76] which works for series of any order; and a quadratic "divide and conquer" algorithm is produced as a corollary of the methods in [TB 78].

Step 6: Construction of main procedure

This step will probably be trivial for all practical applications since iterations usually take place in a single attractive domain, with a starting point within that domain. However it should be pointed out that a general main procedure could be implemented precisely as a Hu-Tucker search tree having "keys" equal to the sorted lower and upper bounds of attractive domains. If the probabilities that the iteration would take place in each domain are known or can be estimated, then the search tree can be optimized using the Hu-Tucker adaptation of the Huffman algorithm cited in Chapter 2 above.

We give an example of how the entire process above might proceed for an algorithm designer seeking to produce a parallel algorithm for the Arithmetic-Geometric Mean (AGM) procedure. The AGM has been shown by Brent [Bre 76] to have numerous useful computational properties besides its original value as a rapid method for computing the elliptic integral

$$K(m) = \int_0^{\frac{\pi}{2}} \frac{d\theta}{\sqrt{1 - m \sin^2\theta}}$$

and other special functions. An excellent survey giving the background of the AGM and related iterations can be found in [Car 71]. The AGM iteration consists of compounding the named means in tandem:

$$a_0 = 1 \qquad b_0 = \sqrt{1 - m}$$

$$a_{k+1} = \frac{1}{2}(a_k + b_k) \qquad b_{k+1} = \sqrt{a_k\, b_k}$$

with quadratic convergence to the fixed point $a_\infty = b_\infty = \pi/(2K(m))$. We fit the AGM into a first-order nonlinear recurrence in the following

way. Let

$$F(x) = 2\sqrt{x}/(1+x) = \operatorname{sech} \ln \sqrt{x}$$

and define $x_0 = \sqrt{1 - m}$ and $x_{k+1} = F(x_k)$ for $k > 0$. We claim that

$$\frac{\pi}{2K(m)} \simeq \prod_{k=0}^{n} ((1+x_k)/2)$$

with the quadratic convergence of the AGM, proof lying in the facts that $x_k = b_k/a_k$ and $((1+x_k)/2) = a_{k+1}/a_k$.

It is easy to verify that $F(1) = 1$ is the fixed point we are interested in, where the starting point $x_0 = \sqrt{1 - m}$ is always contained in the interval $(0,1)$. So, following Step 2 above we derive the power series

$$G(x) = 1 - F(1 - x) = 1 - \sqrt{1-x} \,/\, (1-x/2)$$
$$= 1/8\, x^2 + 1/8\, x^3 + 13/128\, x^4 + \ldots$$

easily since the series for $\sqrt{1-x}$ and $1/(1-x/2)$ are well known. Note that $G(x) = O(x^2)$, reflecting that the AGM is a quadratic method, and since $G'(0) = F'(1) = 0$ we know that 1 is indeed an attractive fixed point. (We have used $(1-x)$ instead of $(x-1)$ in G merely for simplicity, since $(1-x)$ is always positive on the domain of interest.)

Following Step 3, we note that G' has no zeroes on $(0,1)$ so the attractive domain on which it is invertible extends all the way from the attractive fixed point 0 to the repulsive fixed point 1. Hence, our change of variables will be good for all iterates, providing we can derive a convergent form for it.

We now turn to the problem of deriving the change of variables function for G. Since $G'(0) = 0$, section II indicates that we must find a Böttcher solution $\beta(x)$ to $\beta(F(x)) = \beta(x)^2$, which can then be used to

evaluate the iterates x_k as $x_k = \beta^{-1}(\beta(x_0)^{2^k})$. A power series for $\beta^{-1}(x)$ can be derived to any desired order using the following algorithm:

Solution procedure for inverse Böttcher equation $F(\psi(x)) = \psi(x^2)$

The idea here is to take a power series which solves the inverse Böttcher equation up to some order n, and improve this series to be a solution up to order n+1. More precisely, suppose

$$\psi_n(x) = a_1x + a_2x^2 + \ldots + a_nx^n$$

is a polynomial satisfying

$$F(\psi_n(x)) = \psi_n(x^2) + O(x^{n+2}).$$

We contend we can find such a polynomial for all n, and we prove this inductively (and constructively). As a basis note that our contention is true for n = 1,2,3 using the Levy and Lessman formulas [LL 59]

$$a_1 = 1/b_0$$

$$a_2 = \frac{-1}{2} b_1 / b_0^3$$

$$a_3 = \frac{5}{8} b_1^2 / b_0^5 - \frac{1}{2} b_2 / b_0 - \frac{1}{4} b_1 / b_0^3 ,$$

where $F(x) = x^2(b_0 + b_1x + b_2x^2 + \ldots)$. Assuming the statement is true for n, we extend it to n+1 as follows. By Taylor's theorem we have

$$F(\psi_n(x) + cx^{n+1}) = F(\psi_n(x)) + F'(\psi_n(x))\, cx^{n+1} + O(x^{2n+2}).$$

Now if we take the induction hypothesis

$$F(\psi_n(x)) = \psi_n(x^2) + x^{n+2}R(x)$$

where $R(x) = r_0 + r_1x + r_2x^2 + \ldots$, then we find

$$F(\psi_n(x) + cx^{n+1}) - [\psi_n(x^2) + cx^{2(n+1)}]$$

$$= F(\psi_n(x)) - \psi_n(x^2) + F'(\psi_n(x))\, cx^{n+1} + O(x^{2n+2})$$

$$= x^{n+2}R(x) + F'(\psi_n(x))\, cx^{n+1} + O(x^{2n+2})$$

$$= x^{n+2}(r_0+2b_0a_1c) + O(x^{n+3})$$

$$= x^{n+2}(r_0+2c) + O(x^{n+3}).$$

Therefore selecting $c = -r_0/2$ and setting $\psi_{n+1}(x) = \psi_n(x) + cx^{n+1}$ establishes the statement inductively.

Concisely then, an algorithm for constructing $\psi(x)$ to order N may be written as follows:

$a_1 = 1/b_0$

$a_2 = -b_1/(2b_0^3)$

$\psi_2(x) = a_1x + a_2x^2$

for n=3 to N do begin

$r_0 = ((F(\psi_{n-1}(x)) - \psi_{n-1}(x^2)) / x^{n+2})\Big|_{x=0}$

$c = -r_0 / 2$

$\psi_n(x) = \psi_{n-1}(x) + cx^n$

end

This algorithm may be obviously extended to solve the general inverse Böttcher equation $F(\psi(x)) = \psi(x^\mu)$ for any integer $\mu > 2$ as well. A faster (quadratic) version of this process may also be implemented by using more information at each step. Instead of updating $\psi_n(x)$ by cx^{n+1} we can update it by $x^{n+1}P(x)$ where $P(x)$ is the n^{th}-degree polynomial

satisfying $x^{n+2}R(x) + F'(\psi_n(x))\ P(x)\ x^{n+1} = O(x^{2n+2})$, i.e.,

$$P(x) = xR(x) \ / \ F'(\psi_n(x)) \pmod{x^n}.$$

Doing so annihilates the first n terms of R(x), and if we set $\psi_{n+1}(x) = \psi_n(x) + x^{n+1}P(x)$ we discover

$$F(\psi_{n+1}(x)) = \psi_{n+1}(x) + O(x^{2n+2}).$$

Therefore if this method of updating is always used we get quadratic increases in accuracy, as stated. The techniques for manipulating power series in the way required here is described nicely in [BK 76] and in [TB 78]. The latter paper in particular gives quadratic algorithms for solving the Schröder equation when $0 < |F'(0)| < 1$, and the pseudo-Böttcher equation

$$\beta(F(x)) = b_0(\beta(x))^{\mu}$$

when $F(x) = b_0x^{\mu} + O(x^{\mu+1})$.

Employing the above procedure we can derive $\beta^{-1}(x) = \psi(x)$ for the AGM algorithm to any desired accuracy. A list of the first 25 coefficients appears in Table 1; the fact that they increase like 2^n suggests that we will get convergence of the series only for x less than 1/2. Convergence is perhaps the biggest problem confronting the automated use of the linearization methodology discussed here. If one can find closed form for these series then the problem disappears, but otherwise it is impossible to bypass -- and the attractive domains on which linearization is being applied must be cut down to the neighborhoods of the attractive fixed points where the series converge.

n	Coefficient of x^n in $\beta^{-1}(x)$	Coefficient of x^n in $\beta(x)$	
0	0	0	
1	8	.125000000000000000	= 1 / 8
2	-32	.062500000000000000	= 1 / 16
3	96	.039062500000000000	= 5 / 128
4	-256	.027343750000000000	= 7 / 256
5	624	.020568847656250000000	= 337 / 16384
6	-1408	.016265869140625000000	= 533 / 32768
7	3008	.013339996337890625000	= 3497 / 262144
8	-6144	.011243820190429687500	= 5895 / 524288
9	12072	.009678952395915985107	
10	-22976	.008471515029668807983	
11	42528	.007514530327171087265	
12	-76800	.006739115575328469276	
13	135728	.006099163420003606010	
14	-235264	.005562764881688053722	
15	400704	.005107191080924167180	
16	-671744	.004715841674283183240	
17	1109904	.004376324743778692291	
18	-1809568	.004079211253631198634	
19	2914272	.003817204763050890960	
20	-4640256	.003584574260314624121	
21	7310592	.003376758139825465760	
22	-11404416	.003190082169031995660	
23	17626944	.003021555030966448130	
24	-27009024	.002868717703624139870	

Table 1. Power series expansion for AGM change of variables $\beta(x)$

Regrettably, closed form for $\beta^{-1}(x)$ here could not be found, so the changes of variables cannot be applied for all x in (0,1). However we can show that

$$\beta^{-1}(x) = 8x / (1+2x)^2 + O(x^5)$$
$$= 8x\,(1+2x-2x^5+4x^6) / (1+2x)^3 + O(x^9)$$

which is remarkable but not good enough to guarantee convergence.

Finally, following Step 5, we derive the reverted series for $\beta(x)$ in Table 1, using the Brent-Kung method [BK 76]. Recalling that

$$G(x) = 1 - F(1-x) = \beta^{-1}(\,\beta(x)^2\,),$$

so $F(x) = 1 - \beta^{-1}(\,\beta(1-x)^2\,)$, our parallel algorithm for solving the AGM iteration is therefore:

1. Set $x_0 = \sqrt{1 - m}$

2. Set $y_0 = \beta(1-x_0)$

3. Compute $x_k = 1 - \beta^{-1}(\,{y_0}^{2^k}\,)$ in parallel, for $1 \le k \le n$

4. Compute $K(m) = \dfrac{\pi}{2 \prod_{k=1}^{n} ((1+x_k)/2)}$.

Taking convergence into account, we must ensure that ${y_0}^2 < 1/2$ for step three of this process to operate correctly, which requires that $\beta(1-x_0) < 1/\sqrt{2}$. Alternatively we can let the iteration run the usual way $x_{k+1} = F(x_k)$ until $\beta(1-x_k) < 1/\sqrt{2}$, and then finish the iteration via linearization; since the iterates x_k approach 1 very quickly we would not have to wait long for this change in strategies. This observation about x_k leads to a point about accuracy that should be made: The above algorithm is extremely accurate if x_0 is very close to 1, unlike

the ordinary AGM procedure. Thus the above approach might actually be useful in some environments.

Frequently the story has a much happier ending. This would certainly be the case for any of the trigonometric-related iterations of section III, and in fact the arc-hyperbolic-cotangent change of variables for the Newton-Raphson square root iteration was derived in precisely the manner outlined in this section. Consider now the iteration

$$x_{k+1} = x_k^2 + x_k - 1/4$$

mentioned by Kung as being a maximal efficiency iteration with regard to the efficiency measure $E = (\log_2 p)/M$ where p is the order of convergence of the iteration and M is the number of multiplications or divisions required per iterate [Kung 73]. Letting $F(x) = x^2 + x - 1/4$ we find F has an attractive fixed point at -1/2 and $F'(-1/2) = 0$, $F''(-1/2) \neq 0$, so convergence of the iteration at -1/2 is quadratic. (So E = 1 for this iteration.) Proceeding as for the AGM we find

$$G(x) = F(x - (-\tfrac{1}{2})) + (-\tfrac{1}{2}) = x^2 + 2x.$$

Surprisingly we have already found the Böttcher function for G in section III.1 and know that $G^{[n]}(x) = (x+1)^{2^n} - 1$, so Kung's iterates satisfy

$$x_k = (x_0 + 1/2)^{2^k} - 1/2$$

providing, of course, x_0 is within the attractive domain [-1/2,1/2). Thus Kung's iteration is essentially just $x_k = x_{k-1}^2$.

Lastly we consider a linearization of the arithmetic-harmonic mean iteration (14) mentioned in section III.3. Put

$$F(x) = 4x/(1+x)^2$$

and define $x_0 = A$, $x_{k+1} = F(x_k)$ for $k \geq 0$. Then in a manner exactly like that for the AGM iteration we claim

$$\sqrt{A} \simeq \prod_{k=0}^{n} ((1+x_k)/2) .$$

This can be proved by establishing $x_k = a_k/h_k$ and $(1+x_k)/2 = h_{k+1}/h_k$. For example, in computing $\sqrt{5}$ we get the table

k	x_k	$(1+x_k)/2$	$\Pi(1+x_k)/2$	Error
0	5	3	3	+.76393202
1	.55555555	.77777778	2.3333333	+.09726536
2	.91836735	.95918367	2.2380953	+.00202729
3	.99818923	.99909461	2.23606896	+.00000098

Now the observation to make is that

$$\sqrt{F(x^2)} = 2x/(1+x^2)$$

which is a form listed in section III.2, involving the hyperbolic tangent. So explicitly we get

$$x_k = (\tanh(2^k \operatorname{arctanh} \sqrt{x_0}))^2,$$

which recalls the coth result in III.2, but is different. As stated in III.3, finding a Newton-Raphson linearization that does not use square roots appears to be a very difficult problem.

V. Conclusion

We have shown that there is a methodology for the automated linearization of first-order, constant-coefficient recurrences, and that many nonlinear recurrences may be linearizable in general. The usefulness of the first-order result in a parallel processing environment is still subject to debate. However, if nothing more we have demonstrated how difficult it is to obtain realistic theoretical bounds on the power of parallel computation, like the iteration complexity bounds of Kung [Kung76]; and it is now simple enough to compute linearizing changes of variables that the parallel algorithm designer faced with an iteration might profitably consider doing so. In these two respects the results outlined in this paper are conclusive.

Open problems lie in extending the work of section III.3 and in the integration of a nonlinear recurrence resolver into a parallel compiling system like the PARAFRASE compiler [Kuck76]. Interestingly, in a large selection of FORTRAN programs being analyzed by PARAFRASE, the only nonlinear iterations to emerge were Gauss-Jordan elimination and a recurrence of form (10) with $\phi(x) = \log(x)$. Thus it seems unlikely that a compiler system would make much use of a general automated linearizer. Instead, it would seem more cost-effective to equip the compiler with a program capable of recognizing many simple nonlinear recurrence forms, like (11) in section III and whatever other recurrences seem popular for the class of programs being compiled. When unknown recurrences (like Gauss-Jordan elimination) were detected, the reasonable action for the compiler to take would be to recommend examination of the program by an algorithm designer for possible recoding.

VI. References

[Acz 66] Aczél, J. Lectures on Functional Equations and Their Applications. NY: Academic Press, 1966.

[Boo 70] Boole, G. Calculus of Finite Differences (fifth edition; ed. by J.F. Moulton). NY: Chelsea Publishing Co., 1970.

[Bre 76] Brent, R.P. "Fast Multiple-Precision Evaluation of Elementary Functions". JACM 23, 2, 242-251 (April 1976).

[BK 76] _____ & H.T. Kung. "Fast Algorithms for Manipulating Formal Power Series". Technical report, Carnegie-Mellon University, January 1976.

[deB 61] deBruijn, N.G. Asymptotic Methods in Analysis. (esp. Ch. 8) Amsterdam: North-Holland Publishing Co., 1961.

[Car 71] Carlson, B.C. "Algorithms involving Arithmetic and Geometric Means". AMM 78, 5, 496-505(May 1971).

[CKS 76] Chen, S.C., D.J. Kuck, & A.H. Sameh. "Practical Parallel Triangular System Solvers". University of Illinois at Urbana-Champaign preprint, Sept. 1976. Submitted.

[For 18] Forsyth, A.R. Theory of Functions of a Complex Variable. London: Cambridge University Press, 1918.

[Koe 1884] Koenigs, G. "Récherches sur les intégrales de certaines équations fonctionnelles". Ann. Sci. École Norm. Sup. 3, 1, Supplément, pp.3-41 (1884).

[Kuck76] Kuck, D.J. "Parallel Processing of Ordinary Programs", in Advances in Computers vol. 15, pp. 119-179. NY: Academic Press, 1976.

[Kuck74] _____. Private communication.

[Kuc 68] Kuczma, M. Functional Equations in a Single Variable. Warszawa: PWN - Polish Scientific Publishers, 1968.

[Kung73] Kung, H.T. "A Bound on the Multiplicative Efficiency of Iteration". J. Comp. & Syst. Sci. 7, 334-342 (1973).

[Kung76] _____. "New Algorithms and Lower Bounds for the Parallel Evaluation of Certain Rational Expressions and Recurrences". JACM 23, 2, 252-261 (April 1976).

[KT 78] _____ & J.F. Traub. "All Algebraic Functions Can Be Computed Fast". JACM 25, 2, 245-260 (April 1978).

[LL 59] Levy, H. & F. Lessman. Finite Difference Equations. London: Sir Isaac Pitman & Sons, Ltd., 1959.

[Mel 73] Melzak, Z.A. Companion to Concrete Mathematics. NY: John Wiley & Sons, 1973.

[M-T 51] Milne-Thomson, L.M. The Calculus of Finite Differences. London: Macmilland Co., Ltd., 1951.

[Par 77] Parker, D.S. "Nonlinear Recurrences and Parallel Computation", in High-Speed Computer and Algorithm Organization, ed. by D.J. Kuck, D.H. Lawrie, and A.H. Sameh. NY: Academic, 1977.

[SK 75] Sameh, A.H. & D.J. Kuck. "Linear System Solvers for Parallel Computers". Report UIUCDCS-R-75-701, University of Illinois at Urbana-Champaign, February 1975.

[SK 77] _____. "A Parallel QR Algorithm for Symmetric Tridiagonal Matrices". IEEE Trans Comp C-26, 2, 147-53(Feb.77).

[Sch 1871] Schröder, E. "Ueber iterirte Functionen". Mathematische Annalen III, 296-322 (1871).

[Sto 73] Stone, H.S. "An Efficient Parallel Algorithm for the Solution of a Tridiagonal Linear System of Equations". JACM 20, 1, 27-38 (January 1973).

[Sze 58] Szekeres, G. "Regular Iteration of real and complex functions". Acta. Math. 100, 3-4, 203-258 (1958).

[TB 78] Traub, J.F. & R.P. Brent. "On the Complexity of Composition and Generalized Composition of Power Series". Technical Report, Carnegie-Mellon University, April 1978.

Erstens, vergeßt nicht, kommt das Fressen
Zweitens kommt der Liebesakt.
Drittens das Boxen nicht vergessen
Viertens Saufen, laut Kontrakt.
Vor allem aber achtet scharf
Daß man hier alles dürfen darf.

-- B. Brecht, Mahagonny

4
Design and Analysis of Permutation Networks

I. Introduction

This chapter begins by investigating the tradeoffs involved in the implementation of the three-stage rearrangeable switching networks (RSNs) studied by Clos, Benes, and Waksman--taking into special consideration the time and gate complexity of the network's controller. These networks have a variety of applications, from connection of telephone terminals (their original intended use) to connecting parallel processors with memory modules in computers. Figure 9 illustrates the general structure of these networks; d can be any divisor of N, the number of input or output terminals, and the small boxes indicate smaller switches which may, in turn, be decomposed as three-stage RSNs or simply as crossbar switches. The reason for making "conjugated" permuting networks of this type is that they require fewer hardware elements (crosspoints) than the $O(N^2)$ needed by a full crossbar. However, they clearly require more time for execution of a permutation than a crossbar--both for the data to flow through the switch from input to output (the data time), and for the determination of the proper settings of the switch subnetworks necessary to realize the desired permutation (the control time).

A survey of the basic properties of these networks can be found in Chapter 3 of Benes's book [Ben 65]; notably, the RSN is capable of realizing <u>any</u> permutation of its inputs, and for essentially this reason is called <u>rearrangeable</u>. Waksman [Wak 68] and Joel [Joe 68] noticed that any single outer-stage subswitch may be eliminated as redundant--and a switch that is still rearrangeable may be obtained by setting this subswitch permanently to the identity permutation, as in Figure 10. Waksman went on to show that when d = 2 the switch in Figure 10 (with the center switches

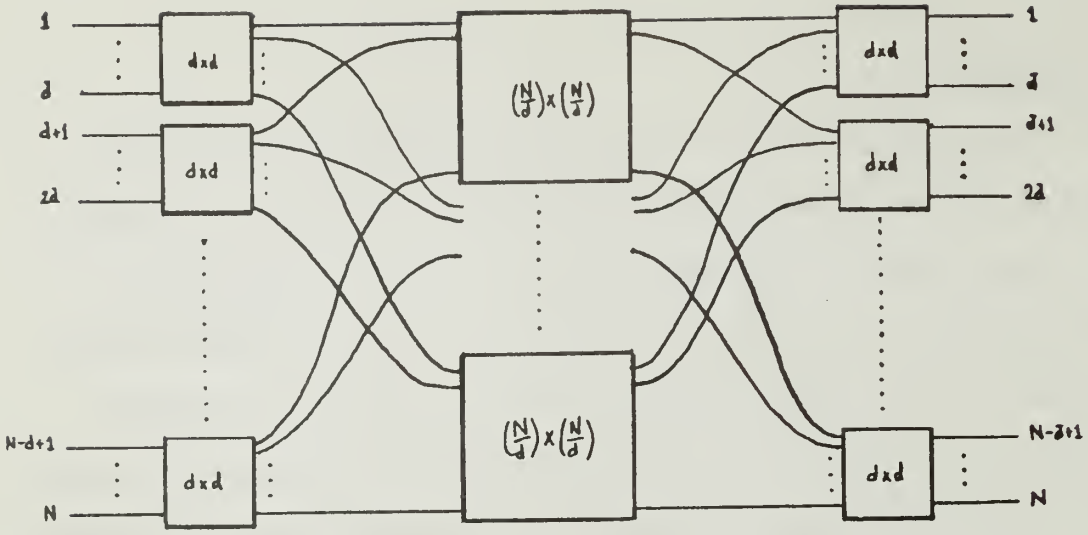


Figure 9. Three-stage RSN of base-d structure

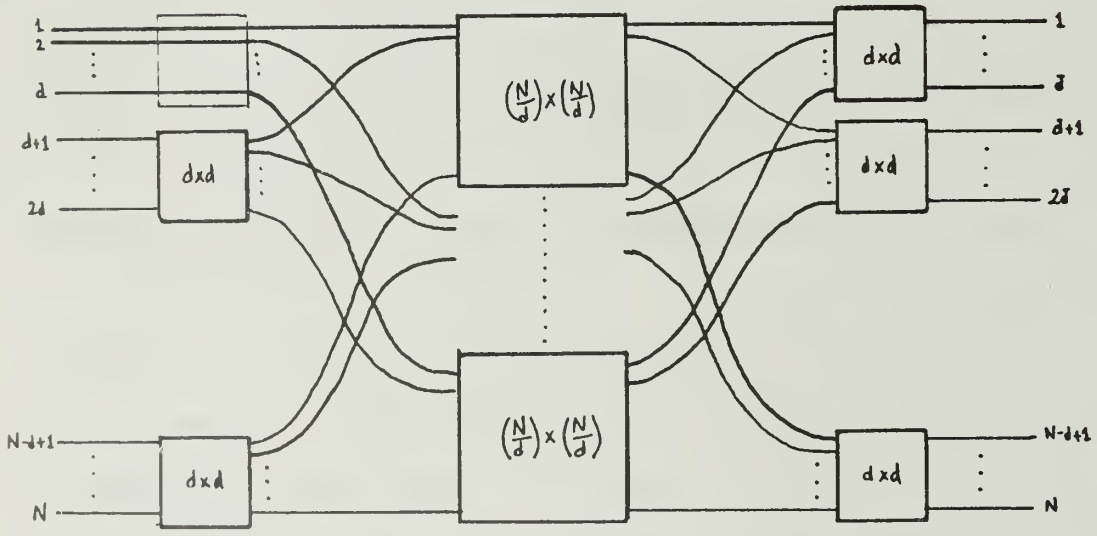


Figure 10. Base-d RSN with redundant switch removed

recursively decomposed as base-2 RSN's) is asymptotically optimal with respect to the number of 2×2 elements, since it contains $F(N) = N \lg N - N + 1$ of them* and

$$\lim_{N\to\infty} \left(\frac{F(N)}{\lg(N!)}\right) = 1 .$$

As we shall see in section II this elimination of redundant switches produces a significant savings when d gets large.

Proofs of the rearrangeability of the RSN, including the proof of Slepian and Duguid based on Philip Hall's theorem on "distinct representatives," are discussed in [Ben 65], [Ben 75b], [Ben 75c]. All of the proofs center on the fact that the switch will permute lines correctly if and only if the center stage subswitches can be set, which is guaranteed by Hall's theorem. This knowledge suggests the structure of a control algorithm for the three-stage RSN:

Step 1. Determine permutation settings needed for each of the d $(\frac{N}{d}) \times (\frac{N}{d})$ subswitches in the center stage (not necessarily unique settings).

Step 2. Determine the permutation settings for the $(\frac{N}{d})$ d×d subswitches in each of the outer two stages.

Step 3. Recursively (if necessary) apply this algorithm to the subswitches whose permutation settings were determined in Steps 1 and 2.

Ultimately this algorithm stops when permutation settings for all "small" subswitches (2×2 "crosspoint" elements or small crossbars, out of which

---

*$\lg(x) = \log_2(x)$

the switch is built) are determined. All existing control algorithms known to us are of the above form, including those presented in this paper. It would be interesting if an alternative approach were found.

Waksman [Wak 68] also suggested a practical method of performing Step 1 in the above algorithm. Given a permutation π on $\{1, \ldots, N\}$, he defined a partition matrix $M = (m_{ij})$ given by

$$m_{ij} = \sum_{k=(i-1)d+1}^{id} \sum_{\ell=(j-1)d+1}^{jd} \delta(\pi(k),\ell)$$

where $\delta(x,y) = \{1 \text{ if } x = y, 0 \text{ otherwise}\}$. If one thinks of π as being a permutation matrix, then one can view M as being a "collapsed" version of π, where each element in M corresponds to the sum of all the entries in a d×d partition of π. It is easy to show that since every row and column in π must sum to one, every row and column in M must sum to d. In addition, M contains all the information needed to perform Step 1 of the control algorithm; this will be discussed in greater depth in section III. Using partition matrices Neiman [Nei 69], and later Ramanujam [Ram 73], Gold and Kuck [GK 74], and Tsao-Wu [TW 74] presented backtracking control algorithms (Neiman's is based on the Hungarian method for solving matching systems) which work for any value of d. Because of the backtracking possibility and the way the algorithms roam over the partition matrix however, none of these algorithms have time complexity even approaching the O(N log N) of Opferman and Tsao-Wu's "looping algorithm" [OTW 71], wnich works for the case d = 2 and does not use partition matrices. There is thus no good existing RSN control algorithm for most computer applications, either for the case d = 2 or for general d, since O(N log N) steps is a long time to wait for switching of data in most circumstances.

This paper, therefore, attacks the problem of finding new, general-d RSN control algorithms for the following reasons. First, a better understanding of the control problem could hardly be harmful, especially because there are so many potential applications of RSN's if the control time could only be reduced. Second, we are interested in the case d > 2 because increased gate densities on modern chips has made mass production of small-to-moderate size crossbar packages feasible [Thu 71]: today RSN's could be constructed of 9×9crossbar chips instead of the 2×2 crosspoint elements originally considered by Clos [Clos 53]. For this reason we set up the following terminology which will be used for the rest of the paper:

Definition A (N,d,k×k) - RSN is a 3-stage, base-d RSN whose subswitches are (recursively) built out of k×k crossbars (and possibly also 2×2 crossbars depending on divisibility of d,k, and N). Note that d is taken to be a function of N; thus d = 2, d = N/2, d = N^{α} with $1/\log N \leq \alpha \leq (\lg N - 1)/\log N$ are all acceptable, and that it makes sense to say that if Figure 10 comprises an (N,d,k×k) - RSN, then its center switches are (N/d(N), d, k×k) - RSNs, and the outer switches are (d(N), d, k×k) - RSNs. Below we will write d for d(N) where no confusion should arise.

The third reason for attempting new control algorithms for values of d greater than 2, is that they can work faster. It will be shown in section II that as d grows the number of crosspoints in the (N,d,2×2) - RSN grows also, over the O(N log N) of the (N,2,2×2) - RSN demonstrated by Waksman. Thus there is some slack (over N!) in the number of possible

states of this RSN; intuitively we should be able to capitalize on this waste by getting a faster control algorithm for large d than we can get when d is small. We show this is essentially what happens, though unfortunately the gain realized by the algorithm here is not substantial.

Finally, one of the more interesting results from examining the RSN for various values of d is the study of a "hybrid" network recursively built out of $(N,N/2,k\times k)$ and $(k,2,2\times 2)$ - RSNs which exhibits interesting timing and gate properties. Analysis of this network leads to a number of conjectures which, on a prima facie basis, make the RSN seem uniformly less cost-effective than the competing switching networks of Batcher [Bat 68], and of Lawrie [Law 75], Lang [LS 76], [Lan 76], and Wen [Wen 76]. These negative results drive us, in sections IX-XI, to analyze possible alternatives to the RSN which we classify loosely as Shuffle/Exchange-type networks. Much less is quantitatively known about the permuting power of these networks, so we concentrate on this problem and derive some (surprising) results which reflect favorably on the potential of the Shuffle/Exchange for use as a computer switching network.

II. Structure of the (N,d,k×k) - RSN

In this section we simply tabulate the data times (number of delays through k×k switches experienced by the data in flowing from input to output) and the number of gates required by (N,d,k×k) - RSNs. We consider N and k to be powers of two for simplicity in solving the recurrences given below, although they need not be in general. Our motivation is to get a feel for the cost and speed of the RSN as d and k vary, so that cases other than the d = k = 2 switch (studied by Benes, Waksman, Opferman and Tsao-Wu, and others) can be evaluated quantitatively.

Three recurrences concern us: for T, the data time for the RSN; for G, the number of "gates" in the RSN measured in 2×2 and/or k×k crossbars; and for G', the number of "gates" for the RSN when redundant subswitches are removed as suggested in [Wak 68]. Note T, G, and G' do not take control overhead into account--control will be discussed in later sections. For the moment we simply want a general idea of RSN cost. It is easy to verify using Figures 9 and 10 that for the (N,d,k×k) - RSN we have

$$T(N) = 2T(d) + T(\tfrac{N}{d}), \qquad T(k) = 1;$$

$$G(N) = 2\,\tfrac{N}{d}\,G(d) + d\,G(\tfrac{N}{d}), \qquad G(k) = 1;$$

$$G'(N) = (2\,\tfrac{N}{d} - 1)\,G'(d) + d\,G'(\tfrac{N}{d}), \qquad G'(k) = 1.$$

These recurrences are not simple to solve in closed form. Note that if d is a small constant and $d \geq k$ then

$$T(N) \approx (2\log_d N - 1)\,T(d)$$

$$G(N) \approx \tfrac{N}{d}\,(2\log_d N - 1)\,G(d)$$

$$G'(N) \approx (\tfrac{N}{d}\,(2\log_d N - 1) - [\tfrac{N - d}{d(d - 1)}])\,G'(d)$$

but no general solution for broad classes of functions d(N) and varying ranges of k is known. However, we can easily obtain solutions for d = 1/2, d = $\sqrt{N}$, and d = N/2. These results are tabulated below, first for k = 2, and then for k an arbitrary power of 2. Note lg 3 ≈ 1.585.

Looking at these tables we notice several interesting trends. First, as d increases for any fixed values of k and N, T(N) and G'(N) also increase. However, for fixed d and N, as k increases T(N) and G'(N) both decrease. Thus if we increase d (for the sake of a faster control algorithm--see below) we can compensate for the increase in data time and crossbar packages by using suitably large k×k crossbars.

To give a concrete feeling for the magnitudes of T, G, and G' for various N, we lastly tabulate them for several values of N, where k = 2 and k = 16.

	T (switch delays)	G (switches)	G' (switches)
$d = 2$	$T(N) = 2 + T(N/2)$ $= 2\lg N - 1$	$G(N) = N + 2G(\frac{N}{2})$ $= N(\lg N - 1/2)$	$G'N = N - 1 + 2G'(\frac{N}{2})$ $= N(\lg N - 1) + 1$
$d = \sqrt{N}$	$T(N) = 3T(\sqrt{N})$ $= (\lg N)^{\lg 3}$	$G(N) = 3\sqrt{N}\, G(\sqrt{N})$ $= \frac{1}{2} N(\lg N)^{\lg 3}$	$G'(N) = (3\sqrt{N} - 1)\, G'(\sqrt{N})$ $\approx (.74)\, G(N)$
$d = \frac{N}{2}$	$T(N) = 2T(\frac{N}{2}) + 1$ $= N - 1$	$G(N) = 4G(\frac{N}{2}) + \frac{N}{2}$ $= \frac{1}{2}(N^2 - N)$	$G'(N) = 3G'(N/2) + N/2$ $= N^{\lg 3} - N$

Table 2a. Time and "Gate" Requirements for (N,d,2×2)-RSN

	T (switch delays)	G (switches)	G' (switches)
d = 2	$T(N) = 2\lg(\frac{N}{k})_{2\times 2s} + 1_{k\times k}$	$G(N) = N\lg(\frac{N}{k})_{2\times 2s} + (\frac{N}{k})_{k\times ks}$	$G'(N) = N\lg(\frac{N}{k}) - (\frac{N}{k} - 1)_{2\times 2s} + (\frac{N}{k})_{k\times ks}$
d = k	$T(N) = 2(\log_k N) - 1_{k\times ks}$	$G(N) = (\frac{N}{k})(2\log_k N - 1)_{k\times ks}$	$G'(N) = G(N) - (\frac{N-k}{k(k-1)})_{k\times ks}$
d = $\sqrt{N}$	$T(N) = (\log_k N)^{\lg 3}_{k\times ks}$	$G(N) = (\frac{N}{k})(\log_k N)^{\lg 3}_{k\times ks}$	$G'(N) = (1 - \varepsilon(k,N))\, G(N)$ [$\varepsilon(k,N) < .1$ for $k \geq 4$]
d = N/2	$T(N) = (\frac{N}{k}) - 1_{2\times 2s} + (\frac{N}{k})_{k\times ks}$	$G(N) = \frac{N}{2}(\frac{N}{k} - 1)_{2\times 2s} + (\frac{N}{k})^2_{k\times ks}$	$G'(N) = k((\frac{N}{k})^{\lg 3} - (\frac{N}{k}))_{2\times 2s} + (\frac{N}{k})^{\lg 3}_{k\times ks}$

Table 2b. Time and "Gate" (crossbar) Requirements for (N,d,k×k)-RSN

N	T(N)			G(N)			G'(N)		
	d = 2	d = $\sqrt{N}$	d = N/2	d = 2	d = $\sqrt{N}$	d = N/2	d = 2	d = $\sqrt{N}$	d = N/2
16	7	9	15	56	72	120	49	55	65
64	11		63	352		2,016	321		665
256	15	27	255	1,920	3,456	32,640	1,793	2,585	6,305
1,024	19		1,023	9,728		523,776	9,217		58,025
65,536	31	81	65,535	1,015,808	2,654,208	2,147,450,880	983,041	1,982,695	42,981,185

Table 3a. Tabulations of T, G, and G' for (N,d,2×2)-RSNs

N	T(N)			G(N)			G'(N)		
	d = 2	d = $\overline{N}$	d = N/2	d = 2	d = $\overline{N}$	d = N/2	d = 2	d = $\overline{N}$	d = N/2
16	0/1	1	0/1	0/1	1	0/1	0/1	1	0/1
64	4/1		3/4	8/4		96/16	5/4		80/9
256	8/1	3	15/16	64/16	48	1,920/256	49/16	47	1,040/81
1,024	12/1		63/64	384/64		32,256/4,096	321/64		10,640/729
65,536	24/1	9	4,095/ 4,096	49,152/ 4,096	36,864	134,184,960/ 16,777,216	45,057/ 4,096	36,049	8,437,520/ 531,441

Table 3b. Tabulations of T, G, and G' for (N,d,16×16)-RSNs

(Numbers above slashes pertain to 2×2 switches; below, to 16×16 crossbars.)

III. Theoretical Background for the Control Algorithm

In this section we briefly present some theoretical results that underly the correctness of the control algorithms for base-d RSNs presented in this paper. The material here is not really new, but is included for completeness. We define some notation first, then cite a result of Birkhoff on doubly stochastic matrices which turns out to be an incarnation of Hall's matching theorem that is useful in this context. With Birkhoff's result we easily prove the Slepian-Duguid theorem for $(N,d,k\times k)$ - RSNs (namely, that they can realize any permutation), and show exactly how the partition matrix mentioned in section 1 can be used to set the RSN to realize any permutation.

Given a permutation map π: $\{1, \ldots, N\} \to \{1, \ldots, N\}$ we define a corresponding permutation matrix $\pi = (\pi_{ij})$ where

$$\pi_{ij} = \begin{cases} 1 & \text{if } \pi(i) = j \\ 0 & \text{otherwise.} \end{cases}$$

Given any divisor d of N, then it is natural to define the partition matrix $M = (m_{ij})$ as above in section one by setting

$$m_{ij} = \sum_{k=(i-1)d+1}^{id} \sum_{\ell=(j-1)d+1}^{jd} \pi_{k\ell} .$$

Thus M is the matrix obtained by partitioning the matrix π into d×d submatrices and then collapsing each of these submatrices into one element by summing all the ones and zeroes they contain. It is important to note that π is a doubly-stochastic matrix, i.e., all of its elements are nonnegative and each of its rows and columns sum to 1. Likewise, the matrix $(\frac{1}{d})$ M is doubly stochastic since each of M's rows and columns must sum to d. We say that M is an unnormalized doubly-stochastic matrix. We can now state Hall's Theorem and a resulting theorem (originally proved by

G. Birkhoff and independently J. von Neumann using different methods) on doubly-stochastic matrices.

Theorem 1 (P. Hall--"Systems of Distinct Representatives")

Given a set A and any r subsets $A_1, \ldots, A_r$, there exists a set of "distinct representatives" $\{a_1, \ldots, a_r\}$ [i.e., $a_i \in A_i$, $a_i \neq a_j$ if $i \neq j$] if and only if the union of any k of the sets $A_1, \ldots, A_r$ contains at least k elements, for each k less than r. [Proofs appear in [Ben 65], [Ber 62]. In this application we consider the case where $r = N/d$ and the sets $\{A_i \; i = 1, \ldots, r\}$ represent the rows of M.]

Theorem 2 (Birkhoff--von Neumann)

Every doubly stochastic matrix is a convex combination of permutation matrices, i.e., if B is a doubly-stochastic matrix then

$$B = c_1P_1 + c_2P_2 + \ldots + c_mP_m$$

where $P_1, \ldots, P_m$ are permutation matrices and $\sum_1^m c_i = 1$

For a proof see Berge [Ber 62, pp. 105-106]. We adapt this theorem as follows:

Theorem 3 (Decomposition of Partition Matrix)

The partition matrix M of order (N/d) can be expressed as the sum of d permutation matrices of order (N/d), i.e.,

$$M = P_1 + P_2 + \ldots + P_d \; .$$

Proof Define sets A_i (i = 1, ..., N/d) as follows:

$$A_i = \{j \mid m_{ij} > 0\} .$$

Since M is unnormalized doubly stochastic, we know these sets satisfy the Hall condition (namely, any union of k of them contains at least k elements) for consider what happens if it does not hold. That would mean that there were some set of k rows of M (without loss of generality the first k) which contained nonzero values in at most k - 1 columns altogether (without loss of generality the first k - 1). But then we know the sum of the entries in the first k rows of M is

$$\sum_{i=1}^{k} \sum_{j=1}^{N/d} m_{ij} = \sum_{i=1}^{k} \sum_{j=1}^{k-1} m_{ij} = kd$$

whereas the sum of the first (k - 1) columns is

$$\sum_{i=1}^{N/d} \sum_{j=1}^{k-1} m_{ij} \geq \sum_{i=1}^{k} \sum_{j=1}^{k-1} m_{ij} = kd > (k - 1)\, d$$

which we know is false for the partition matrix M. Thus we can apply Hall's theorem and extract a set of representative columns j from each of the row sets A_i . If we let a_i be the column selected from set A_i , then the matrix $P_1 = (P^1_{ij})$ defined by

$$P^1_{ij} = \begin{cases} 1 & \text{if } j = a_i \\ 0 & \text{otherwise} \end{cases}$$

is a permutation matrix. We can apply the theorem inductively to $M - P_1$ since $M - P_1$ is still unnormalized doubly stochastic (n.b., we must replace d by (d - 1) everywhere above in this process), and after d such applications find a decomposition

$$M = P_1 + P_2 + \ldots + P_d .$$

The next result is a simple derivative of Theorem 3.

Theorem 4 (Slepian-Duguid, restricted to (N,d,k×k) - RSNs.)

An (N,d,k×k) - RSN can be set to realize any permutation π on N letters.

Proof (Constructive) Compute the partition matrix M corresponding to π and, using Theorem 3, find a decomposition $M = P_1 + \ldots + P_d$. Set the i^{th} center switch (inductively) to realize P_i, for $1 \leq i \leq d$ (cf., Figure 9). It is then straightforward to find permutation settings for each of the d×d outer switches that map their inputs to the now-determined outputs correctly, and set each of these switches inductively. It is clear, once the center switches have been set, that the entire RSN can realize π as requested. Consult Algorithm 2 of section VI for the details.

To close this section we give a simple example. Suppose we have a (9,3,3×3) - RSN which is to be set to realize the permutation

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ 5 & 1 & 3 & 6 & 2 & 9 & 7 & 8 & 4 \end{pmatrix}.$$

Then M is the $(\frac{N}{d})\times(\frac{N}{d}) = 3\times3$ partition matrix

$$\begin{pmatrix} 2 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 2 \end{pmatrix}.$$

Note that

$$M = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} + \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix} + \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

Therefore a valid setting of the RSN is as shown in Figure 11.

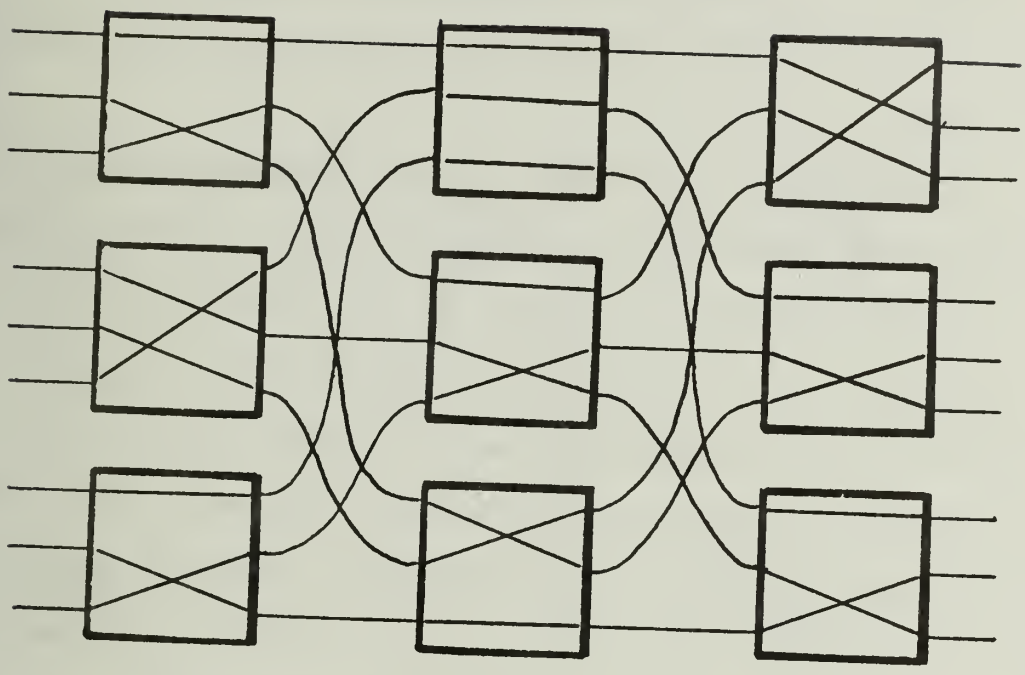


Figure 11. Setting of (9,3,3×3)-RSN for example π

IV. The Special Case d = 2

For d = 2 the RSN has the configuration in Figure 12. For simplicity here and hereafter, we assume N is a power of 2 unless otherwise indicated.

In this case it is not really practical to control the switch using the partition matrix approach outlined in sections I and III, since this matrix alone would have to comprise $(\frac{N}{2}) \times (\frac{N}{2})$ words of lg N bits each: thus either we must buy $O(N^2 \lg N)$ gates to hold the partition matrix--in which case we might as well build a crossbar since it is faster and cheapter--or else we must buy a large random-access memory and pay $O(N^2)$ clocks to initialize the matrix each time we seek to set the switch.

Fortunately for d = 2 there is an alternative, the "Looping algorithm" of Opferman and Tsao-Wu [OTW 71]. The Looping algorithm is based on the simple observation that, if lines number i and i + 1 are inputs to a 2×2 switch in the left outer stage of the RSN, then they must be gated to opposite center switches. Any algorithm which gates <u>all</u> of the inputs in a consistent manner to these center switches will (recursively) define an (N,2,2×2) - RSN control algorithm. The Looping algorithm does just this: it begins with an arbitrary assignment of one of the inputs to one of the center switches and proceeds to make all the assignments required by the first one. For any input i, let $\hat{i}$ be the other input entering into the same 2×2 switch as i (Opferman and Tsao-Wu call $\hat{i}$ the <u>dual</u> of i). To start the looping procedure we gate input i arbitrarily to center switch 1 and $\hat{i}$ to center switch 2. We must then gate output $\pi(\hat{i})$

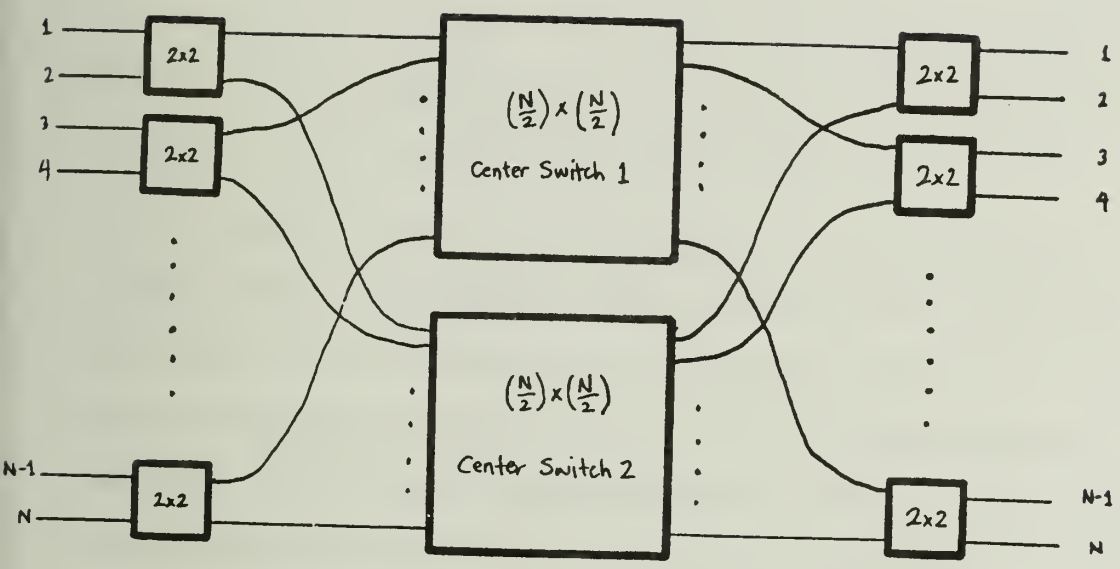
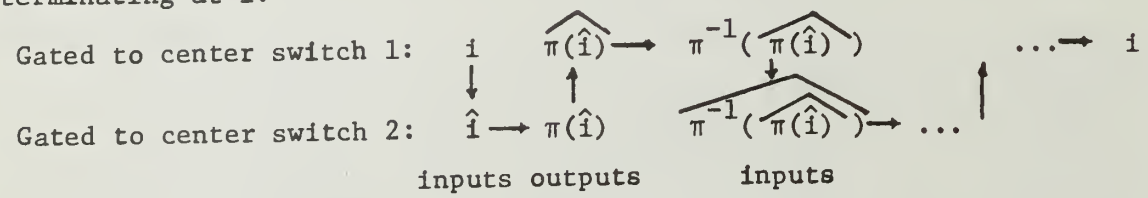


Figure 12. Base-2 RSN

to center switch 2 if the RSN is to work. But this gates output $\widehat{\pi(\hat{i})}$ to center switch 1, so input $\pi^{-1}(\widehat{\pi(\hat{i})})$ must be gated to center switch 1. Continuing this process leads to a "loop" of assignments eventually terminating at i:



This process may not assign all inputs from one loop, in which case another initial arbitrary assignment and loop must be made. But if $\pi(x)$ and $\pi^{-1}(x)$ are available (Opferman and Tsao-Wu point out that either a content-addressable memory or two memories can be used to store π) and new unassigned inputs are immediately available without search (in case the Looping process terminates "prematurely"), then the Looping algorithm will take N steps to set all of the outer stage 2×2 switches, of which there are $2(\frac{N}{2}) = N$. Ignoring the time needed to rearrange the memory contents to represent the permutation settings for the two center switches, since the Looping algorithm must be applied recursively lg N times to set the entire switch, the Looping algorithm takes N lg N steps to control the (N,2,2×2)-RSN, at a cost of O(N lg N) gates for the memory and control hardware. At a gate level the Looping algorithm takes at least $O(N \lg^2 N)$ gate delays since among other things the memory address decoding time is O(lg N) delays.

It has apparently never been noticed before that the above timings can be improved by a factor of $\frac{1}{2}$ lg N through the use of parallelism in the control.† By using separate memories and separate control hardware for each of the subswitches set in recursive applications of the Looping algorithm, the control time can be cut to

† Late note: Clark Thompson has detected this fact in [Tho 77].

$N + N/2 + N/4 + \ldots + 1 = (2 - 1/N)\ N = \underline{2N - 1 \text{ steps}}$.

The use of parallelism requires a factor of 1/2 lg N more gates, however, Thus we have the gate-time tradeoff summarized in Table 4. This speed improvement is appreciable but is not as good as we would like (namely, a control algorithm that runs in O(lg N), or at the very least o(N), steps), and unfortunately for the same order of gates one can buy a Batcher switch which has a control time of only $\frac{1}{2}\ \lg^2 N$ steps [Bat 68].

Note that the Looping algorithm and any algorithm like it seems to inherently require $\Omega(N)$ steps (i.e., requires at least O(N) steps) for two reasons. First, the algorithm requires the availability of π^{-1} values; these values must either come from a CAM or an auxiliary memory, each of which have to be filled at some point--which requires N steps. Second, the Looping process by itself, even if it could make multiple parallel memory accesses, cannot be parallelized to a significant degree because of the cycle structure of permutations. We cannot have multiple processors working on different loops since there is no way to tell a priori in time o(N) that the loops are different (i.e., two processors could find that they were both working on the same loop and that their (arbitrary) assignments of switch settings conflicted). Moreover, Opferman and Tsao-Wu have shown [OTW 71,p.1606] that most permutations have only one loop (out of N! permutations, $(N/2)!(N/2-1)!2^{N-1}$ have this property) and derive a formula for the number of permutations having m loops. In addition to this, the work of Shepp and Lloyd [SL 66] may be applied to show that the length of the longest loop generated by the Looping algorithm for a random input permutation on N letters (equivalent to 2 times the length of the longest cycle in a random permutation on N/2 letters) has an expected value asymptotic to $E\left[\begin{array}{c}\text{maximum loop length} \\ \text{for random } \pi\end{array}\right] \sim 2\lambda\ (\frac{N}{2} + \frac{1}{2}) = \lambda(N + 1)$

	Time (Program Steps)	Time (Gate Delays)	Gates Required
Original Looping Algorithm	N lg N	$O(N \lg^2 N)$	O(N lg N)
Parallel Looping Algorithm	2N - 1	O(N lg N)	$O(N \lg^2 N)$

Table 4. Control Time and Gate Requirements for (N,2,2×2)-RSN

where $\lambda = .62432965...$ is a constant. Thus not only is it possible that two processors will be working on the same loop, it is likely. Clearly, the Looping algorithm or any variant of it is not going to lead to sublinear control time with a reasonable (i.e., $O(N \lg^2 N)$, to be competitive with Batcher) number of gates.

It is interesting to note that the (N,2,2×2)-RSN can be used as a rapid one-bit sorter. This is a new result and will be used below. Muller and Preparata [MP 75] point out that Batcher's network, with time $O(\lg^2 N)$ and gates $O(N \lg^2 N)$ is the best known 1-bit sorter, then construct a sorter that works in time $O(\lg N)$ using $O(N^2)$ gates. We show here that the (N,2,2×2)-RSN can be used to sort bits in time $O(\lg^2 N)$ using $O(N \lg N)$ gates, thus making an improvement over Batcher's switch in gates. It would be extremely interesting to find a one-bit sorter working in time $O(\lg N)$ which required much less than $O(N^2)$ gates.

Note that the (N,2,2×2)-RSN in Figure 12 will sort bits if:

1) the left stage of 2×2 switches gates equal numbers of zeroes and ones to each of the center switches (with a difference of at most one, should there be an unequal number of zeroes and ones).
2) the center stage switches recursively sort their N/2 input bits.
3) the right stage of 2×2 switches "merges" the sorted lists from the center stage in the obvious way: zeroes are gated up and ones are gated down (but note that all but at most one of the switches in this stage will receive identical inputs).

The only nontrivial step in the control is Step (1), but a small amount of thought reveals that setting the left stage of 2×2 switches can be completed in $O(\lg \frac{N}{2})$ gate delays simply by fanning in, using a binary tree, lines from these switches which indicate whether the switch inputs disagree or not (i.e., whether the inputs are 0 and 1, instead of 0 and 0 or 1 and 1). When two such lines are fanned together and both indicate disagreement, the corresponding 2×2 switches are signalled to send their 1's to opposite center switches and no further disagreement is indicated from them. When a line indicating disagreement is merged with a line indicating no disagreement, a line indicating disagreement results (together with the latching of a path back down the tree so that the 2×2 switch with disagreement may be accessed later, when it is known where its inputs should be sent. Thus Steps (1) and (3) together take $O(\lg N)$ steps, and by applying this process recursively to the whole switch the input bits can be sorted in $O(\lg^2 N)$ gate delays using $O(N \lg N)$ gates for the RSN and control. Notice, interestingly, that this switch is very similar in spirit to the odd-even merge network of Batcher [Bat 68] but incorporates a number of simplifications justifiable for the purpose of sorting bits.

V. The Special Case d = N/2

For d = N/2 the RSN has the configuration in Figure 13 (Waksman reduced). Using recurrences it is easy to show that the $(N,\frac{N}{2},2\times2)$-RSN contains $O(3^{\lg N}) = O(N^{\lg 3}) \approx O(N^{1.58})$ of the 2×2 switches. This is more switches than required by the (N,2,2×2)-RSN, which may be why not much attention has been paid to this configuration. However, here the control algorithm is <u>extremely</u> simple and fast--we show that the $(N,\frac{N}{2},k\times k)$-RSN can be controlled in time $O(\lg^2 N \lg(N/k))$ delays using $O(N \lg N (\frac{N}{k})^{\lg 3 - 1}) + O((\frac{N}{k})^{\lg 3} G(k))$ gates, where G(k) is the number of gates in the k×k switch. Unfortunately also the time for the data to flow through the (N,N/2,2×2)-RSN is N - 1 2×2 switch delays, as shown in section II. Thus the d = N/2 switch has the opposite problems of the d = 2 switch: whereas the latter is cheap, fast, and hard to control, the former is expensive, slow, and easy to control. We will exploit this strange reversal in section VII (as much as we can).

The control for the d = N/2 RSN is simple because in this case there is effectively no time needed to set the center stage subswitches. In this case the partition matrix M is of dimension 2×2, so it is trivial to decompose it into a sum of permutation matrices. However, we do not even have to evaluate M here because removal of the redundant upper-left-hand $\frac{N}{2} \times \frac{N}{2}$ switch forces the settings of the center 2×2 subswitches, which can be set in parallel in one step. Thus all we have to do is find permutation settings for the two outside stages, and then set the 3 resulting $\frac{N}{2} \times \frac{N}{2}$ subswitches recursively in parallel.

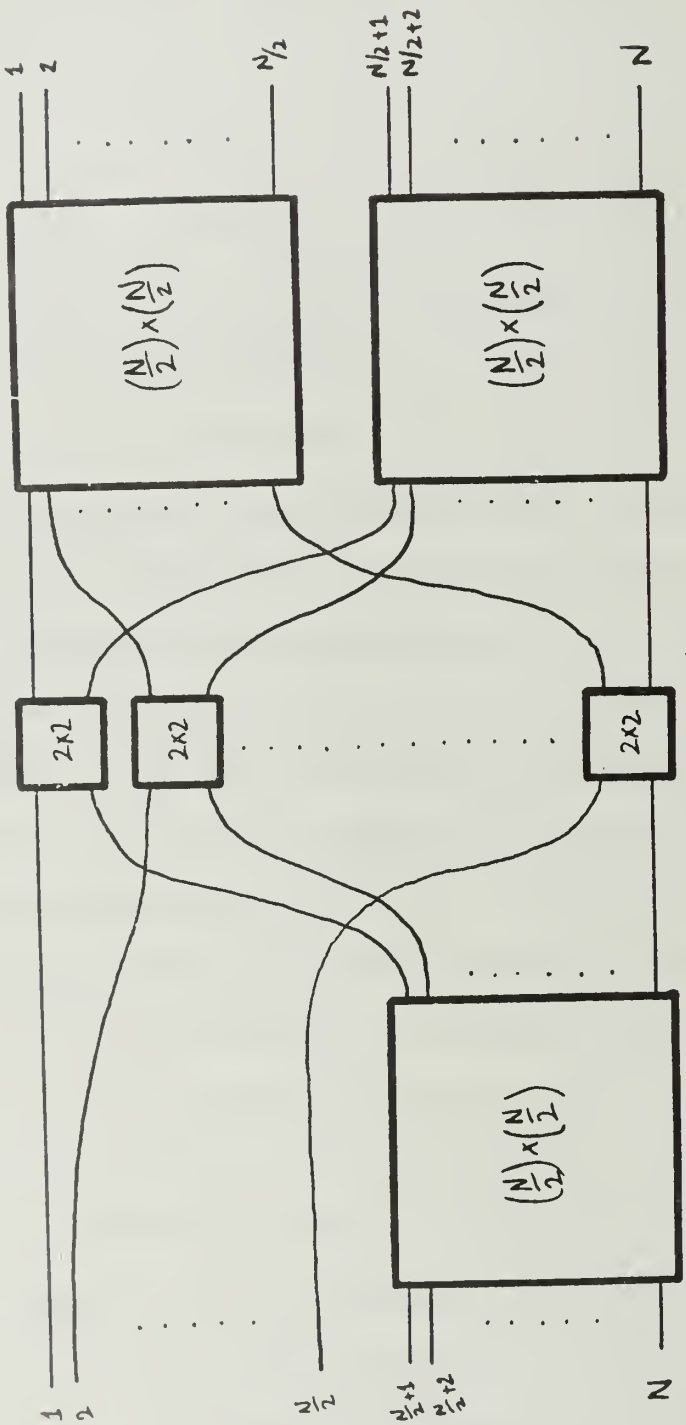


Figure 13. Base-N/2 RSN

This is just as easy as it sounds. After we have set the center 2×2 switches, the outputs of the lower left switch in Figure 13 are connected with the 2 switches in the right stage. We assign a "parity" value 0 or 1 to these outputs according to whether they are connected to the first or second switch in the right stage. Simultaneously, we assign parity values to the inputs i of the lower left switch according to which of the two switches $\pi(i)$ is in. The entire RSN will work if we can get the lower left switch to connect its inputs to its outputs in such a way that these parity values match. Once this has been done the setting of the right stage switches is trivial.

The method used here to achieve this matching is the use of two one-bit sorters, though there may be some better method. We use the sorter described in section IV--which requires $O(\lg^2 N)$ delays and $O(N \lg N)$ gates. These sorters are connected in a chain as shown in Figure 14, which forms a data path through which the values of i and $\pi(i)$ can flow to be used in setting the left and right stages, respectively. Note that the input permutation values π are assumed to be available in <u>registers</u> and that no use of π^{-1} has been made, so we have not forced ourselves to the $\Omega(N)$ time bound required by the Looping algorithm.

If we analyze carefully the requirements of the above algorithm we find that, since it requires $\lg(N/k)$ recursive steps to complete, the control time for the whole RSN is $O(\lg^2 N \lg(N/k))$ delays. Also, suppose that two M-input 1-bit sorters of the type discussed in section IV require $cM \lg M$ gates, for some constant c. Then this control algorithm for the Base-N/2 RSN requires

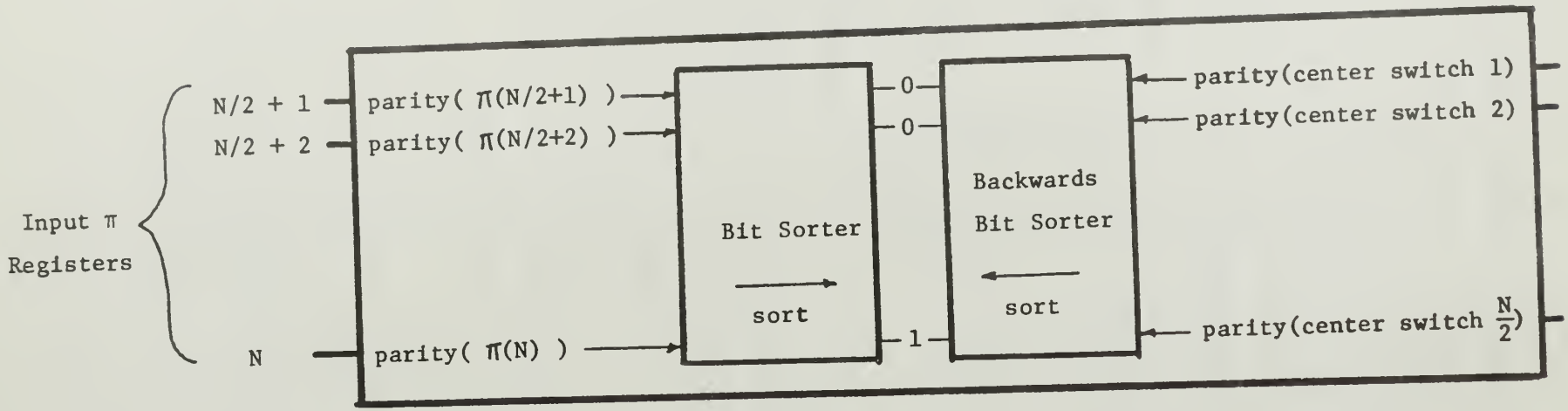


Figure 14. Method for finding settings for lower left switch in Figure 13.

$$c\frac{N}{2}\lg\frac{N}{2} + 3c\frac{N}{4}\lg\frac{N}{4} + 3^2 c\frac{N}{8}\lg\frac{N}{8} + \ldots + 3^{\lg(N/k)-1}\, ck\lg k + 3^{\lg(N/k)-1}\, G(k)$$

$$< c\frac{N}{2}\lg\frac{N}{2}\left(1 + 3/2 + (3/2)^2 + \ldots + (3/2)^{\lg(N/k)-1}\right) + 3^{\lg(N/k)-1}\, G(k)$$

$$\approx 2c\frac{N}{2}\lg\frac{N}{2}\left((N/k)^{3/2}\right) + \frac{1}{3}\left(\frac{N}{k}\right)^{\lg 3} G(k)$$

gates, where G(k) is the number of gates required by a k×k switch. [This need not be k^2, i.e., we need not necessarily use crossbars for these subswitches.]

It is interesting to see how the Base-N/2 RSN could be used as a 1-bit sorter, for comparison with the previous section. There are two modes of operation we consider: first, when the switch is in the Waksman-reduced configuration of Figure 13, and second, when no reduction has been made.

Note that when the (N,N/2,2×2)-RSN of Figure 13 is used to sort bits, the center stage switches are trivial to set as usual (if the immediately incident input bit is zero, the switch is set to gate it to the upper right stage switch, otherwise to the lower one). However, after this there is no nice recursive decomposition--it appears that the lower left stage switch must again do 2 chained 1-bit sorts as it did for the (N,N/2,k×k)-RSN control algorithm above. Thus although the Waksman reduction saves us a great deal of gates, it only increases the complexity of controlling the switch for sorting.

If the Waksman-Joel reduction is not used, though, control of the switch is very straightforward. It is clear that the switch will sort if:

1) The output of the first stage and input to the second stage forms a bitonic sequence of zeroes and ones (i.e., a string of N bits of the patterns $0^*1^*0^*$ or $1^*0^*1^*$ in Kleene-notation; cf.,[Bat 68] for more about bitonic sequences).
2) The second stage works in the obvious way: it gates 0 inputs up and 1 inputs down.
3) The third stage switches sort their inputs recursively.

This sorting scheme is most easily executed by making the top left switch a sorter and the bottom left switch an "upside-down" sorter, so the input to the second stage is always of the form $0^*1^*0^*$. Unfortunately, although the switch takes time $O(1)$ to set (all the 2×2 switches must do ultimately is behave like the Batcher elements described in requirement (2) with the polarity (whether sorting is being done "upside-down" or not) taken into account), the data time is $O(N)$ for flow through the switch and the switch requires $O(N^2)$ gates. Thus the switch is not interesting in its own right as a sorting device the way the (N,2,2×2)-RSN was.

It is extremely interesting to see how the d = 2 and d = N/2 sorting networks behave like Batcher's even-odd merge and bitonic sorting networks, respectively. Among other things it brings certain philosophical points to the foreground (even-odd merging strives for a simple starting and ending procedure, whereas bitonic sorting simply wants a simple step in the middle) and highlights the fact that bitonic sorting requires more hardware than even-odd merging. It would be interesting if some theory about sorting networks could be developed from this similarity.

VI. A Non-backtracking Control Algorithm for General d

When we are not given that d is some special value near 1 or near N (e.g., d = 2 or d = N/2 as just discussed, or for example, d = $(\frac{N}{3})$ or $(\frac{N}{4})$ so that a decomposition of the partition matrix M by exhaustively running through all 3! or 4! permutation matrices is feasible), then the only known methods for controlling the (N,d,k×k)-RSN are the backtracking algorithms of Neiman [Nei 69], [TW 74], Ramanujam [Ram 73], and Gold and Kuck [GK 74]. This section gives a new algorithm, based on the partition matrix attack described in sections I and III and outlined below in Figure 15. This algorithm runs in $O((\frac{N}{d})^2)$ steps or $O((\frac{N}{d})^2 \lg(\frac{N}{d}))$ delays, and requires $O(p(\frac{N}{d})^2 \lg d)$ gates where $p \geq 1$ is a parameter; it relies on various lookahead indicators to eliminate the need for backtracking. The lookahead heavily relies on parallelism, and special-purpose hardware is of course required to set the RSN in the stated time bounds (i.e., unlike the Looping algorithm this method may not be encoded as a program to run on an external processor connected to the RSN).

Generally speaking, it seems that the algorithm used to control an (N,d,k×k)-RSN will always depend on the relative size of d with respect to N. We saw that special techniques were applicable for very small or very large values of d in sections IV and V. The same is true here: in all that follows in this section we assume d is not small compared to N (say, $d = \Omega(\sqrt{N})$) because the partition matrix itself requires $O((\frac{N}{d})^2 \lg d)$ bits of storage, which is exorbitant if d is not very large. Note that the algorithm of Figure 15, or any algorithm using partition matrices, is *a priori* good only for large d. Opferman and Tsao-Wu's "Looping Algorithm"

Step 0. Compute Partition Matrix for Permutation to be Realized

Input Permutation π, stored as an N-word vector of registers.

Output Partition Matrix M, dimension $(\frac{N}{d}) \times (\frac{N}{d})$ array. [d = d(N)]

Step 1. Get Center Subswitch Permutation Settings

Input Partition Matrix M.

Output A set $\{P_1, \ldots, P_d\}$ of permutations on $\{1, \ldots, \frac{N}{d}\}$ (stored as vectors), such that when expressed in matrix form

$$P_1 + P_2 + \ldots + P_d = M .$$

Step 2. Get Outer Subswitch Permutation Settings

Input Original permutation π and center switch permutations $\{P_1, \ldots, P_d\}$.

Output A set $\{Q_2, \ldots, Q_{2\frac{N}{d}}\}$ of permutations on $\{1, \ldots, d\}$ (stored as vectors) giving settings for all of the outer subswitches.

Step 3. Recurse

Invoke Steps 0-2 for all permutations in $\{P_1, \ldots, P_d\} \cup \{Q_2, \ldots, Q_{2\frac{N}{d}}\}$ which cannot be directly applied to k×k (or 2×2) switches.

Figure 15. (N,d,k×k)-RSN control algorithm

[OTW 71] for d = 2 takes advantage of special properties of the matching problem $M = P_1 + P_2$ to eliminate the construction of the partition matrix M altogether. This is another example of a combinatorial problem that has an elegant solution for the parameter = 2 case, but gets complicated for the case parameter > 2.

Below we give several algorithms to solve Steps 0-2 of the process in Figure 15. Algorithm 0 solves Step 0, algorithm 1 Step 1, and algorithms 2.1 and 2.2 Step 2. Input and output specifications are as in Figure 15.

Algorithm 0 Compute Partition Matrix M

Timing $O((\frac{d}{p} + \lg p)\lg d)$ gate delays; p is a parameter described below, with $1 < p \leq d/2$

Gates $O(p(\frac{N}{d})^2 \lg d)$

Method

Each row of M is filled in independently in parallel; p processors work on each row, with every processor having its own copy of the current value of entries for that row. Thus essentially p copies of M are kept. There are d entries in π that must be tabulated in each row of M, so each processor tabulates d/p entries; when this is done the p copies of M are added up (in time O(log p log d)).

Note that p should be chosen so that (1) not too many gates are used by this algorithm, but (2) the algorithm is not too slow. Thus for $d = \sqrt{N}$, p = 1 might be appropriate, but for d = N/2, $p = (\sqrt{N}/2)$ is needed to achieve roughly the same bounds.

(Note that in the case d = N/2 M is only a 2×2 matrix, so many copies may be kept at little cost.)

Algorithm 1 Simple Serial Extraction of Permutations from M

Timing $O(d(\frac{N}{d})\ \lg(\frac{N}{d})) = O(N \lg(\frac{N}{d}))$ gate delays (ignoring fanout--see below)

Gates $O((\frac{N}{d})^2 \lg d)$ [adders, etc.] + $O(d(\frac{N}{d})\ \lg(\frac{N}{d}))$ [for $\{P_1, \ldots, P_d\}$]

Method

The permutations P_i (i = 1, ..., d) are peeled off entry by entry, one by one, using sufficient lookahead to prevent bad selections as shown in Figure 16. We show the lookahead vector can be defined to make this algorithm work. Consider the algorithm beginning the j-loop at some point. We know that the partition matrix M is now the sum of (d - i + 1) permutation matrices. After the j-loop has executed several times, we are effectively working on a submatrix M' of M, of dimension $(\frac{N}{d} - j + 1) \times (\frac{N}{d} - j + 1)$. M' is precisely that submatrix with rows 1 to j-1 and those columns marked AVAILABLE=FALSE being deleted from M. We know inductively that M' contains a permutation matrix of dimension $(\frac{N}{d} - j + 1) \times (\frac{N}{d} - j + 1)$, and we try to extend the permutation P_i being extracted by absorbing this permutation. We delete the first row and some column k' in M' (equivalently, row j and some column k from M, where k corresponds to k') after choosing $P_i(j) = k$ as being a good extension. If we define LOOKAHEAD_OK(ℓ), for $\ell = 1, \ldots, (\frac{N}{d})$ by

```
do  i = 1 to d;              /*   Once for each permutation P_i     */
AVAILABLE(1),...,AVAILABLE(N/d)=TRUE;     /*   Indicate all images of P_i available*/
    do  j = 1 to N/d;                     /*   Once for each entry of P_i    */
    Compute  LOOKAHEAD_OK(1),...,LOOKAHEAD_OK(N/d)  (Boolean vector)
        using  AVAILABLE  and rows j through (N/d) of M, as described
        in the text.  Using parallelism this takes time  O(log(N/d)).
    Select k such that LOOKAHEAD_OK(k)=TRUE  in time O(log(N/d)).
    Set    P_i(j) ← k.  (Actually set a switch if required.)
    Set    AVAILABLE(k) ← FALSE.
    Set    M(j,k) ← M(j,k) - 1.
    end;
end;
```

Figure 16. Algorithm 1

$$\text{LOOKAHEAD\_OK}(\ell) = \begin{cases} \text{FALSE} & \text{if column } \ell \text{ of M is not in M' (i.e., AVAILABLE}(\ell)\text{=FALSE)} \\ & \underline{\text{or}} \text{ if } M(j,\ell) = 0 \text{ (i.e., } M'(1,\ell') = 0 \text{ where } \ell' \text{ corresponds to } \ell) \\ & \underline{\text{or}} \text{ if the minimum column or row sum of M' with row 1 and column } \ell' \text{ deleted is zero } (\ell' \text{ corresponds to } \ell) \\ \text{TRUE} & \text{otherwise} \end{cases}$$

then we contend choosing $P_i(j) = k$ is a good extension iff LOOKAHEAD_OK(k)=TRUE, i.e., the algorithm will not get stuck and will peel off all the permutations P_i without backtracking.

The proof is simple: it is clear first of all that this is a necessary condition for choosing $P_i(j) = k$, since we must have $M(j,k) > 0$ [$P_i(j) = k$ is possible] and AVAILABLE(k)=TRUE [$P_i(j') = k$ cannot already have been assigned]. We must show sufficiency. We have inductively that M' contains at least one permutation matrix. By selecting only columns at each step having true LOOKAHEAD_Ok's, we guarantee that the submatrix handled by the next step also contains at least one permutation matrix (since, by the third condition in V's definition, each row and column in it has sum at least 1). Thus the j-loop works inductively, so we can successfully remove at least one permutation P_1 from M. But then Birkhoff's theorem applies, and since $M - P_1$ is (unnormalized) doubly-stochastic, we can apply the j-loop successfully to it again. Thus the i-loop must work inductively as well, so we can extract $\{P_1, \ldots, P_d\}$ in sequence from M.

To see how LOOKAHEAD_OK can be computed in time $O(\log(\frac{N}{d}))$ using only $O((\frac{N}{d})^2 \log(\frac{N}{d}))$ gates, consider the following. First, it is clear that the only difficult computation is that of evaluating whether or not the row sums are zero when various columns are deleted. Some simple algorithms for computing these row sums include (1) summing all the rows and then subtracting the elements in columns being deleted, taking time $O(\log(\frac{N}{d}) \log d)$ and $O((\frac{N}{d})^2 \log d)$ gates, and (2) setting a bit for each entry in M indicating whether that entry is zero/unavailable or not, and then fanning all these bits in with OR-trees in time $O(\log(\frac{N}{d} - 1))$ using $O((\frac{N}{d})^3)$ gates. A more efficient algorithm might work as follows: suppose for any given row we compute a bit-vector $\vec{S}$ of dimension $(\frac{N}{d})$ such that

$$S(k) = \begin{cases} 0 & \text{if the row's sum is zero when column k is deleted} \\ 1 & \text{otherwise.} \end{cases}$$

We can recursively construct $\vec{S}$ in time $O(\frac{N}{d} \lg(\frac{N}{d}))$ by dividing it into subproblems of half the size. In Figure 17, the outputs out of the top of the box represent S(k) for $1 \leq k \leq n$; the bottom output is a line indicating whether <u>any</u> of the row entries considered are zero, or not. Of course, the inputs are entered at the lowest level of recursion, in boxes $S^{(2)}$ as in Figure 18. It is easy to verify that $\vec{S}$ is the output of box $S^{(N/d)}$ (assuming N/d is a power of 2). Moreover, since $S^{(n)}$ runs in time $O(\log n)$ and requires gates

$$G(n) = 2G(n/2) + n + 1, \qquad G(2) = 1$$

$$\Longrightarrow \qquad G(n) = n \lg n - 1 \qquad \text{(for n a power of 2)}$$

We can thus compute $\vec{S}$ in time $O(\lg \frac{N}{d})$ and gates of $O(\frac{N}{d} \lg \frac{N}{d})$.

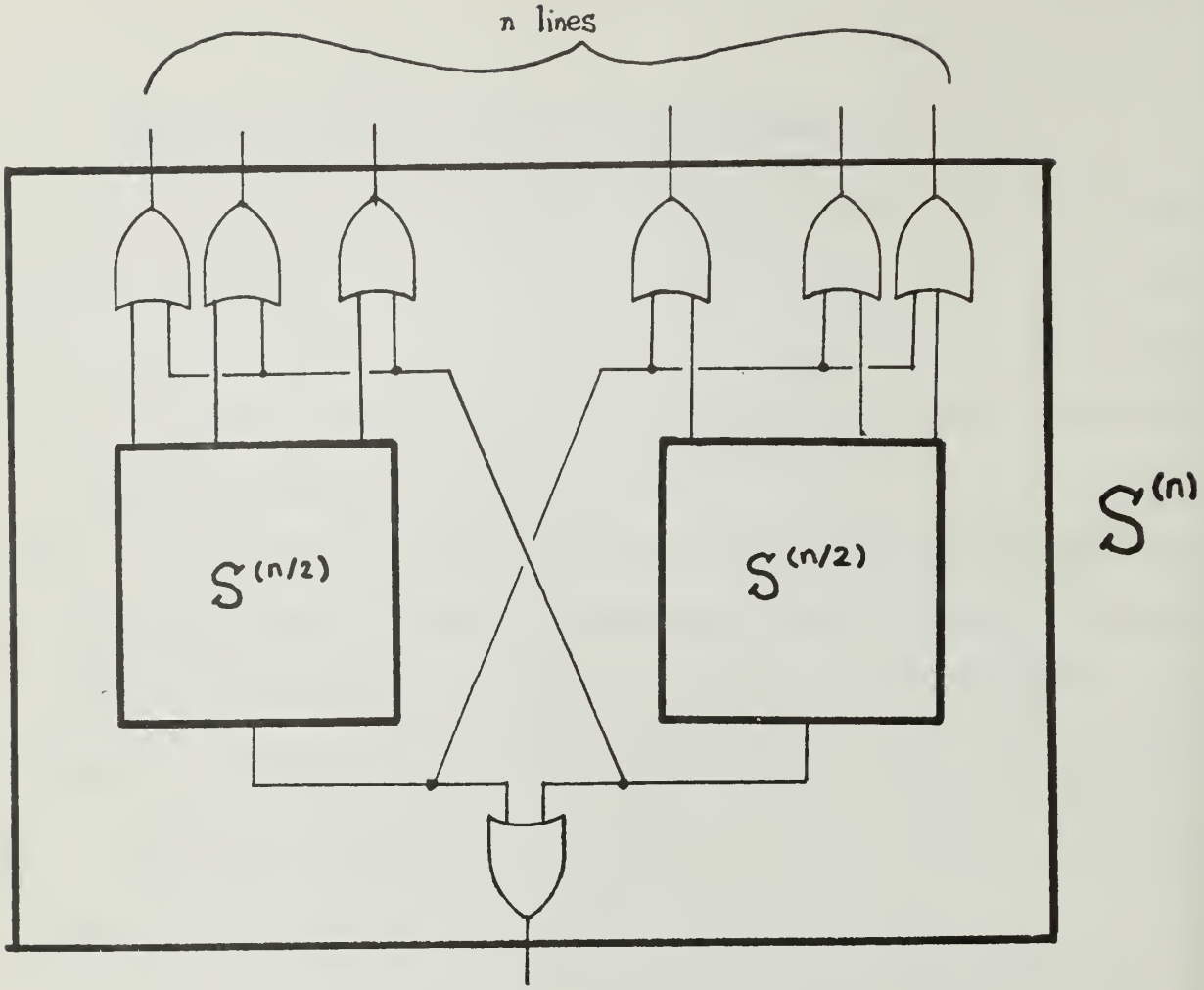


Figure 17. Nonzero-row-sum-detector construction

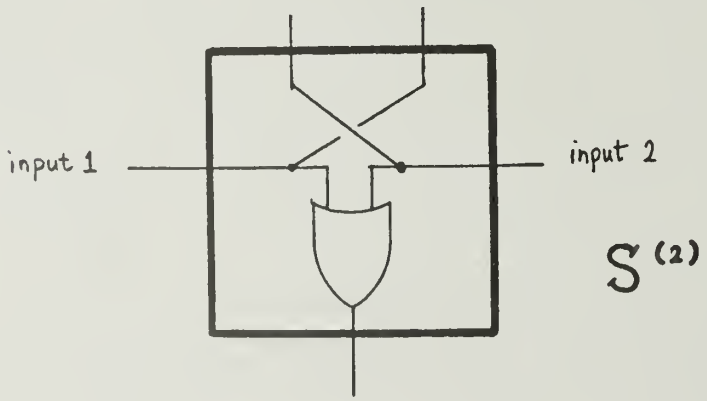


Figure 18. Nonzero-row-sum-detector initialization

Since there are $(\frac{N}{d})$ rows and computation of the other factors of LOOKAHEAD_OK is easier, we can get everything in time $O(\lg(\frac{N}{d}))$ using $O((\frac{N}{d})^2 \lg \frac{N}{d})$ gates. This construction <u>does</u> ignore gate fanout; more complicated schemes for computing $\vec{S}$ could avoid this problem, or simply accept a time bound of $O(\lg(\frac{N}{d}) \log_f (\frac{N}{d}))$ gate delays, where f is the fan-out limit of the gates used.

To clarify the execution of Algorithm 1, we give a short example. Consider the problem of setting a (12,4,4×4)-RSN to realize the permutation

$$\pi = \begin{pmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\ 6 & 2 & 1 & 12 & 10 & 5 & 9 & 11 & 3 & 4 & 7 & 8 \end{pmatrix}$$

We find $M = \begin{pmatrix} 2 & 1 & 1 \\ 0 & 1 & 3 \\ 2 & 2 & 0 \end{pmatrix}$, and a program trace of Algorithm 1 would look as follows:

```
i = 1;                                          /* Select P_1 from M */
      AVAILABLE = (TRUE,TRUE,TRUE);             /* All 3 columns available */
      j = 1;
            LOOKAHEAD_OK = (TRUE,TRUE,FALSE);   /* note M(1,3) = 0 */
            Select k = 1
            Set  P_1(1) = 1, AVAILABLE(1) = FALSE, M(1,1) = 1;
      j = 2;
            LOOKAHEAD_OK = (FALSE,FALSE,TRUE);
            Select k = 3;     /* Note avoidance of M(2,2) */
            Set  P_1(2) = 3, AVAILABLE(3) = FALSE, M(2,3) = 2;
```

j = 3;

LOOKAHEAD_OK = (FALSE,TRUE,FALSE);
/* Only AVAILABLE(2) = TRUE */
Select k = 2

Set $P_1(3) = 2$, AVAILABLE(2) = FALSE, M(3,2) = 1;

At this point we have

$$P_1 = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{pmatrix}$$

and

$$M = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 2 \\ 2 & 1 & 0 \end{pmatrix} .$$

i = 2; AVAILABLE = (TRUE,TRUE,TRUE)

j = 1; LOOKAHEAD_OK = (TRUE,TRUE,FALSE)

Select k = 1; Set $P_2(1) = 1$, M(1,1) = 0.

j = 2; LOOKAHEAD_OK = (FALSE,FALSE,TRUE);

Select k = 3; Set $P_2(2) = 3$, M(2,3) = 1.

j = 3; LOOKAHEAD_OK = (FALSE,TRUE,FALSE)

Select k = 2; Set $P_2(3) = 2$, M(3,2) = 0.

Again we get

$$P_2 = \begin{pmatrix} 1 & 2 & 3 \\ 1 & 3 & 2 \end{pmatrix} \text{ , and now } M = \begin{pmatrix} 0 & 1 & 1 \\ 0 & 1 & 1 \\ 2 & 0 & 0 \end{pmatrix} .$$

i = 3; AVAILABLE = (TRUE,TRUE,TRUE);

j = 1; LOOKAHEAD_OK = (FALSE,TRUE,TRUE);

Select k = 2; Set $P_3(1) = 2$, M(1,2) = 0;

j = 2; LOOKAHEAD_OK = (FALSE,FALSE,TRUE);

Select k = 3; Set $P_3(2) = 3$; $M(2,3) = 0$;

j = 3; LOOKAHEAD_OK = (TRUE,FALSE,FALSE);

Select k = 1; Set $P_3(3) = 1$; $M(3,1) = 1$;

This leaves $P_3 = \begin{pmatrix} 1 & 2 & 3 \\ 2 & 3 & 1 \end{pmatrix}$ and $M = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} = P_4 = \begin{pmatrix} 1 & 2 & 3 \\ 3 & 2 & 1 \end{pmatrix}$

which the algorithm will extract on the pass through the i = 4 loop.

Notes on Algorithm 1

1) Currently the algorithm produces a set of permutations $\{P_1, \ldots, P_d\}$ in random order. If one wanted to use them on the center switches of an RSN with the redundant upper-left-hand subswitch set permanently to the identity (the Waksman-Joel simplification discussed in section 1, cf., Figure 10) then they would have to be reorganized. Fortunately, a minor modification of the algorithm leads to production of permutations $\{P_1, \ldots, P_d\}$ in an order that can be directly applied to the center switches when this redundant switch has been removed. Note that in any decomposition the entries taken from the first row of M require no lookahead (since they must wind up in some permutation matrix, it makes no difference which one). Thus we simply set $P_i(1) = k$ for all i where $k = \lfloor \pi(i)/d \rfloor$ instead of randomly selecting $P_i(1)$. [Notice that the first input to center switch i is from input i which flows through the redundant first switch, now set to the identity.]

2) Note that Algorithm 1 has inherent time complexity $\Omega(N)$ because it repeats a process requiring $O(N/d)$ steps d times. Since there is no obvious (correct) way of using any more parallelism in its implementation than has already been done, the time bound $\Omega(N)$ seems inescapable. It seems the only way (using partition matrices) to get a sub-linear time complexity algorithm is to find some fast method of decomposing a partition matrix M, whose columns and rows sum to d, into two partition matrices of the same order M_1 and M_2, whose columns and rows sum to d/2, i.e.,

$$M = M_1 + M_2 .$$

This "divide-and-conquer" approach could be applied repeatedly in parallel until permutation matrices appeared at the bottom of a computation tree of height $\lceil \lg d \rceil$. Thus if the decomposition $M = M_1 + M_2$ took time τ, then the whole process (assuming everything could be done in parallel) would take time $\tau\lceil \lg d \rceil$. Unfortunately, we can find no good decomposition algorithm.

Algorithms 2.1 -- 2.2 Setting the Outer Switch Stages

As indicated above, the input here is the original permutation π and the center switch settings $\{P_i | i = 1, \ldots, d\}$, and the desired output is switch settings $\{Q_2, \ldots, Q_{2\frac{N}{d}}\}$ for the outer switches. All algorithms considered here work as follows, in a two-step process:

Step 2.a Determine left-side switch settings $\{Q_2, \ldots, Q_{N/d}\}$.

Note that after the center switch settings $\{P_i\}$ have been set, then

Right-side-destination-switch # (left-side-switch i, output j) $= P_j(i)$,

i.e., the outputs of the left-side switches are directly connected with the inputs of the right-side switches in a precise way. Therefore, for $\{Q_2, \ldots, Q_{N/d}\}$ to constitute a valid setting for the left side we must have

$$Q_i(k) = \ell \Longrightarrow \lfloor \pi((i-1)d + k)/d \rfloor = P_\ell(i)$$

since the inputs to switch i are exactly $\{(i-1)d + k | k = 1, \ldots, d\}$. In other words, the destinations forced by the center switches must be correct. Thus our algorithm for this step will take the inputs k of the outer switches and match them (serially) to an output ℓ satisfying the above requirement.

Step 2.b Determine right-side switch settings $\{Q_{N/d+1}, \ldots, Q_{2N/d}\}$

This is trivial once the left side has been set, for all inputs and outputs have been determined. Since the complexity is

smaller than that of Step 2.a, we omit discussion of the implementation of this step altogether.

Algorithm 2.1 Determination of $Q_2, \ldots, Q_{N/d}$

Timing $O(d^2 \lg d)$ gate delays

Gates $O(\frac{N}{d} \lg d)$ gates

Method All the d×d settings $Q_2, \ldots, Q_{N/d}$ are determined in parallel, each using the brute force $O(d^2)$ matching technique:

```
AVAILABLE(1), ..., AVAILABLE(d) ← TRUE;
do  k = 1 to d;
      do  ℓ = 1 to d;
      if      ⌊π((i-1)d + k)/d⌋ = P_ℓ(i)      & AVAILABLE(ℓ)
            then do;
                 Q_i(k) ← ℓ
                 AVAILABLE(ℓ) ← FALSE
                 end;
      end;
end;
```

Figure 19. Algorithm 2.1

Note that there are no "access conflicts" for the values $P_\ell(i)$ between processors setting Q_i and Q_j in parallel, since the processor setting Q_x references only $P_\ell(x)$ for $\ell = 1, \ldots, d$.

Algorithm 2.2 Determination of $Q_2, \ldots, Q_{N/d}$

Timing $O(\lg d \lg N/d)$ gate delays

Gates $O(d \lg d \lg N/d)$ gates

Method The matching is achieved by sorting the values $\{\lfloor \pi((i-1)d + k)/d \rfloor | k = 1, \ldots, d\}$ (keeping their original order as tag information) and then using binary search to find matches with $P_\ell(i)$. The sorting, to achieve the above parallel time bounds, would require a small ($\lceil \lg N/d \rceil$-bit, d-input) Batcher network [Bat 68]. Note that if $d = O(N)$ then this algorithm is useless to us, since we are trying to design a switch that competes with Batcher's, not that subsumes it.

VII. Hybrid Switches and Conjectures on Bounds

Until now we have restricted our attention to rearrangeable switching networks whose structure is recursively defined by a single function d = d(N). In this section we consider what power is gained when we permit multiple functions d, calling any RSN which uses two or more d's a Hybrid RSN. An example of a Hybrid RSN is given in Figure 20, utilizing d = N/2 and d = 2 RSN structure. In fact, the only Hybrid RSNs we will consider here will use a judicious mixture of the d = 2 and d = N/2 switches.

We noticed in section V that the (N,2,2×2)-RSN is cheap, fast, and hard to set while the $(N,\frac{N}{2},2\times2)$-RSN is expensive, slow and easy to set. This suggests the following approach: we use a d = N/2 structure at first to break the switch setting problem down rapidly into a set of smaller switch setting problems, then before we use too many gates or make the switch too slow we change to the d = 2 structure and finish the problem off. Thus, formally, we initially set an (N,N/2,k×k)-RSN and then set a number of (k,2,2×2)-RSNs, for some intelligent value of k. With the formulas derived in sections II, IV, and V, the right value of k is not difficult to obtain.

Theorem 5 The Hybrid (N,N/2,k×k)/(k,2,2×2)-RSN

with $k \approx N/_{(\lg N)^{(\frac{1}{\lg 3-1})}} \approx N/_{(\lg n)^{1.7095113}}$

can be set in time $O(N\ (\lg N)^{0.2905})$ gate delays using $O(N \lg^2 N)$ gates.

Proof From section II, we have that the time for the data to flow through the Hybrid switch in 2×2-switch delays is

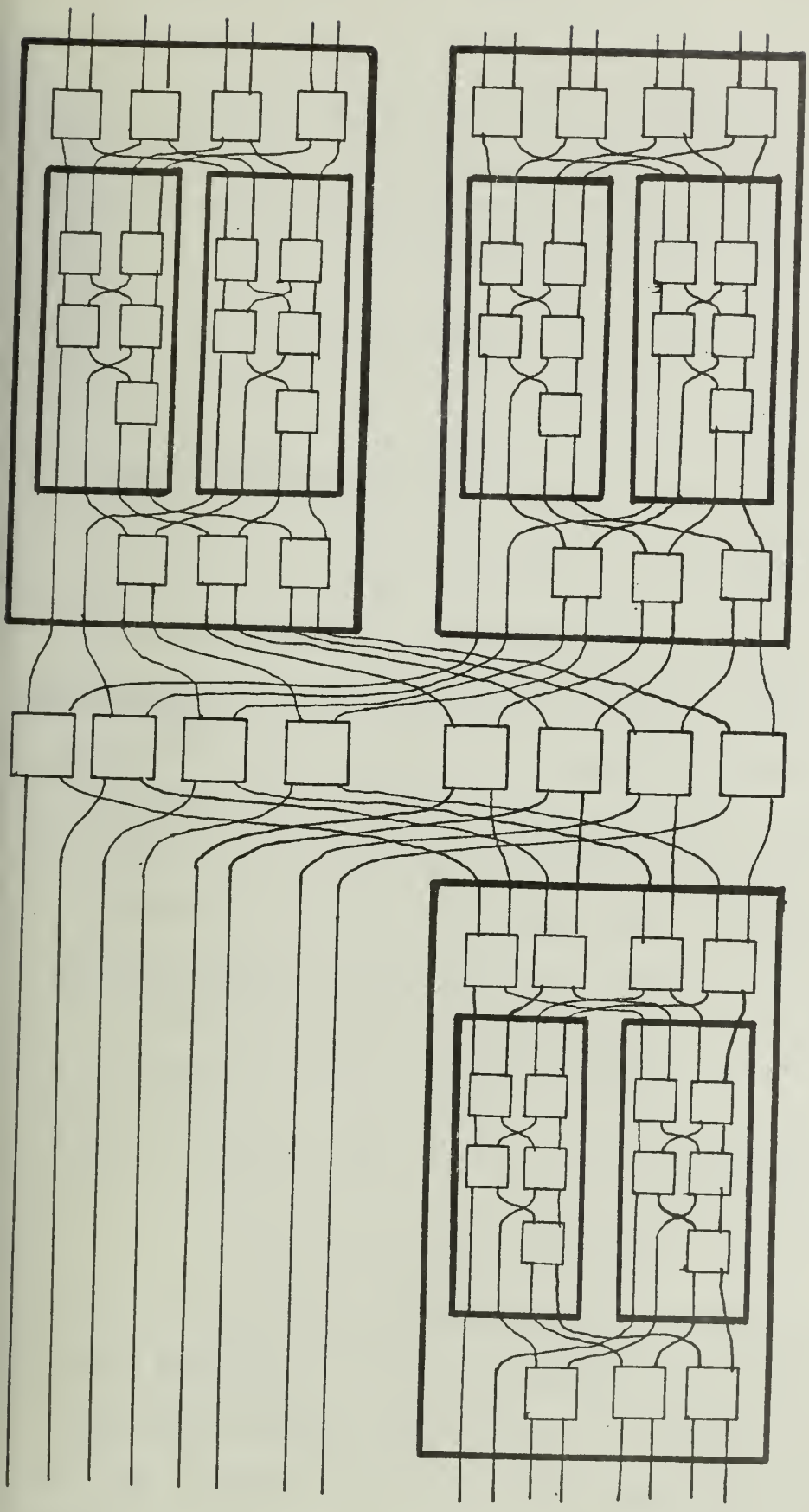


Figure 20. 16-line Hybrid RSN using d = N/2 and d = 2 structure
(Boxes are 2×2 switches).

$$T_D(N,k) = (\frac{N}{k}) - 1 + [(\frac{N}{k})]\ [2\lg k - 1]$$

$$= 2(\frac{N}{k})\ \lg k - 1$$

and the total number of 2×2-switches required is

$$G_D(N,k) = k((\frac{N}{k})^{\lg 3} - (\frac{N}{k})) + (\frac{N}{k})^{\lg 3}\ [k(\lg k - 1) + 1]$$

$$= (\frac{N}{k})^{\lg 3}\ k \lg k + (\frac{N}{k})^{\lg 3} - N$$

Write $k = N^{\alpha}$, for some variable α to be specified momentarily. Then

$$G_D(N,N^{\alpha}) = \alpha\ N^{(1-\alpha)\lg 3+\alpha}\ \lg N + N^{(1-\alpha)\lg 3} - N\ .$$

We are assuming here that $G_D(N,k)$ is $O(N\ \lg^2 N)$ for the switch to be competitive with Batcher's networks. (Below in Theorem 6 we consider using more gates.) If this is true, then

$$\alpha\ N^{(1-\alpha)\lg 3+\alpha}\ \lg N = o(N\ \lg^2 N)$$

so

$$\alpha\ N^{\lg 3+\alpha(1-\lg 3)} = o(N\ \lg N)\ .$$

Since $\lg N = N^{\lg \lg N/\lg N}$ we find the above statement is true if

$$\lg 3 + \alpha(1-\lg 3) < 1 + \lg \lg N/\lg N$$

or equivalently

$$\alpha > 1 + \frac{\lg \lg N}{(1-\lg 3)\lg N}$$

implying that to use less than $O(N\ \lg^2 N)$ gates we must have

$$k > N/_{(\lg N)^{(\frac{1}{\lg 3-1})}} = N/_{(\lg N)^{1.7095113}}$$

Table 5 gives a feel for this function of N for the switch sizes that could interest us. Although the function seems asymptotically not that much smaller than N, it is fairly small for all N of interest. What this says is that we can use the fast d = N/2 control algorithm without using

N	lg N	$\lceil N/(\lg N)^{1.7095} \rceil$	$\lceil \lg(\frac{N}{(\lg N)^{1.7095}}) \rceil$
16	4	2	1
32	5	3	2
64	6	3	2
128	7	5	3
256	8	8	3
512	9	12	4
1,024	10	20	5
2,048	11	34	6
4,096	12	59	6
8,192	13	103	7
16,384	14	180	8
32,768	15	320	9
65,536	16	573	10
131,072	17	1,033	11
262,144	18	1,874	11
524,288	19	3,417	12
1,048,576	20	6,259	13

Table 5. Lower bounds for k in Hybrid d = N/2, d = 2 RSN

too many gates until we reach fairly small subproblems, of size k. Additionally, the time for the data to flow through a network with k near $N(\lg N)^{-1.7}$ is

$$T_D(N,N(\lg N)^{-1.7}) = 2(\lg N)^{1.7} \lg(\frac{N}{(\lg N)^{1.7}}) - 1$$

$$\approx 2(\lg N)^{2.7}$$

over our range of interest. We must also check the amount of time and gates used by the control of our Hybrid RSN. From sections IV and V we have the formulas

$$T_c(N,k) = O(\lg^2 N \lg(N/k)) + O(k \lg^2 k) \text{ delays}$$

$$G_c(N,k) = O(N \lg N(\frac{N}{k})^{\lg 3-1}) + O((\frac{N}{k})^{\lg 3}[k \lg k]) \text{ gates}$$

assuming we are using the original Looping algorithm and not the parallelized one. Again, letting k approach the limit $N(\lg N)^{-1.7095}$ we discover

$$T_c(N,N(\lg N)^{-1.7095}) = O(\lg^2 N \lg\lg N) + O(N (\lg N)^{-1.7095} \lg^2 (N(\lg N)^{-1.7095}))$$

$$= o(N(\lg N)^{0.2905}) \text{ delays.}$$

$$G_c(N,N(\lg N)^{-1.7095}) = o(N \lg^2 N) \text{ gates.}$$

Therefore, Theorem 5 follows, since we have $o(N \lg^2 N)$ gates and achieved the stated time bound.

In the above proof, it is shown that the bottleneck of the Hybrid RSN is still in the time needed to control it, due to the restriction on the number of gates. We can remove this bottleneck and match the control time with the data time if we lift the gate restriction. Theorem 6 states the result.

Theorem 6 The Hybrid $(N,N/2,k\times k)/(k,2,2\times 2)$-RSN with $k = \sqrt{N}$ can be set in time $O(\sqrt{N} \lg N)$ gate delays using $O(N^{(1+\lg 3)/2} \lg^2 N) \simeq O(N^{1.3} \lg^2 N)$ gates.

Proof Since we are concerned with minimizing time here instead of gates, we use the parallel Looping algorithm of section IV. We then find the formulas for data and control time and gates as in Theorem 5

$$T_D(N,k) = 2(N/k) \lg k - 1$$

$$G_D(N,k) = (N/k)^{\lg 3} k \lg k + (N/k)^{\lg 3} - N$$

$$T_c(N,k) = O(\lg^2 N \lg(N/k)) + O(k \lg k)$$

$$G_c(N,k) = O(N \lg N (N/k)^{\lg 3 - 1}) + O((N/k)^{\lg 3} [k \lg^2 k])$$

Note that $T_c(N,\sqrt{N})$ and $T_D(N,\sqrt{N})$ are of the same order as functions of N, so in that sense if we recursed down to problems of size $k = \sqrt{N}$ before changing from a $d = N/2$ structure to a $d = 2$ structure, we would be "balancing" the control time against the data time. (Actually a slightly smaller or slightly larger value of k might get a better balance; to say anything definite we need to know the constants involved in T_c and T_D, but we have avoided doing this since there are so many considerations to be taken into account--for example, we should really be discussing time in clocks and not gate delays if this switch is really to be built. These problems will be discussed in the next section.)

If we choose $k = \sqrt{N}$, then we find

$$T_D(N,\sqrt{N}) = \sqrt{N} \lg N - 1$$

$$G_D(N,\sqrt{N}) = O(N^{\frac{1+\lg 3}{2}} \lg N)$$

$$T_C(N,\sqrt{N}) = O(\lg^3 N) + O(\sqrt{N} \lg N)$$

$$G_C(N,\sqrt{N}) = O(N^{\frac{1+\lg 3}{2}} \lg^2 N)$$

Thus the theorem follows. We should point out that for $N \leq 1024$ it seems likely that the $O(\lg^3 N)$ term will dominate the control time, so for small N in this area the balancing value of k will have to be more carefully determined.

The above two theorems and the experience accumulated in the development of this paper lead to two conjectures on lower bounds for the amount of time needed to control the switch:

Conjecture 1: Three-stage RSNs cannot be set in time $o(N)$ steps using $o(N \lg^2 N)$ gates.

Conjecture 2: Three-stage RSNs cannot be set in time $o(\sqrt{N})$ steps using $o(N^2)$ gates.

VIII. RSN Conclusions

The evidence above suggests (particularly if the two conjectures in section VII can be proved) that the three-stage RSN must be rejected as impractical at computer interconnection speeds, since it is uniformly slower and more expensive than the Shuffle-Exchange Networks of Lawrie [Law 75], Lang [Lan 76], [LT 76], and Wen [Wen 76], and if not requiring more gates than Batcher's networks [Bat 68] it certainly requires more time. From a theoretical standpoing the RSN seems very unappealing as a switching device where switching speed is important.

From a computer designer's standpoint, it should be emphasized however, the RSN may not be so unappealing when there is a moderate number of input lines. To truly measure the effectiveness of the RSN one needs real constants on the gate counts (with packaging considerations taken into account) and time estimates in clocks and not gate delays since there are always factors like wire length and design latching requirements which affect the total timing of the hardware and are not captured by "gate delays." A designer should not necessarily be deterred by the asymptotic pessimism of the above conjectures unless he wishes to build an enormous switch.

The RSN may not be such an unattractive switch also if one considers programming it to realize the most frequently-encountered permutations quickly. This is the approach that has been taken in [FS 77a] and [FS 77b], for example, where k-shifts, shuffles, broadcasts, and other useful configurations have been preprogrammed so that the entire RSN may be set in a few clocks for this restricted class. Clearly, the library of programs can grow to fit any application in an easy way. This approach has the disadvantage that the processors will idle for an intolerably long time

if the switch is forced to realize some permutation it has no program for, but if it can be guaranteed that this eventuality will arise only very rarely, if at all, then the RSN will be a good alternative.

Thus, one cannot conclude immediately from the results here that RSNs are useless for computer interconnection. It would be great if the above conjectures could be disproved by, for example, finding a good "divide and conquer" algorithm as discussed in section VI; however, we are skeptical that a good RSN control algorithm exists. It seems more likely that by studying other switching networks, like the Shuffle/Exchange or multi-stage RSN, that more cost-effective networks for computers will be found. We therefore now turn our attention to Shuffle/Exchange networks.

IX. Introduction to Shuffle/Exchange Networks

For some time it has been known that Shuffle/Exchange networks provide an effective interconnection scheme for parallel computation on many problems -- see, for example, [Sto 71] and [Law 76]. These networks are constructed of repeated copies (or a cyclically reused single copy) of a "perfect shuffle" connection followed by a column of 2x2 crosspoint elements which can exchange adjacent line values independently. See Figure 21. Stone points out in [Sto 71] that this network can be used for sorting à la Batcher [Bat 68], evaluating polynomials, transposing matrices, and computing fast Fourier transforms (as Pease showed [Pea 68]); Lawrie, in [Law 76], shows that many useful routing permutations (in particular those arising in matrix computations) can be realized using these networks; and others (e.g., [Lang76], [LS 76]) have shown that networks based on the simple Shuffle/Exchange have other interesting properties.

The standard model for a multi-stage (= multi-copy) Shuffle/Exchange network is the Omega network of Lawrie, which consists of $n = \lg(N)$ Shuffle/Exchanges, where N is the number of input lines (which we will assume is a power of two). The Omega network for N=16 is illustrated in Figure 22. The purpose of this paper is to compare properties of this network with those of two other networks: first, Pease's indirect binary n-cube array [Pea 77] (Figure 23), and second, a network obtained by appending a bit-reversal connection --clarified below-- to the first half of the base-2 RSN discussed in section IV, which we will call an

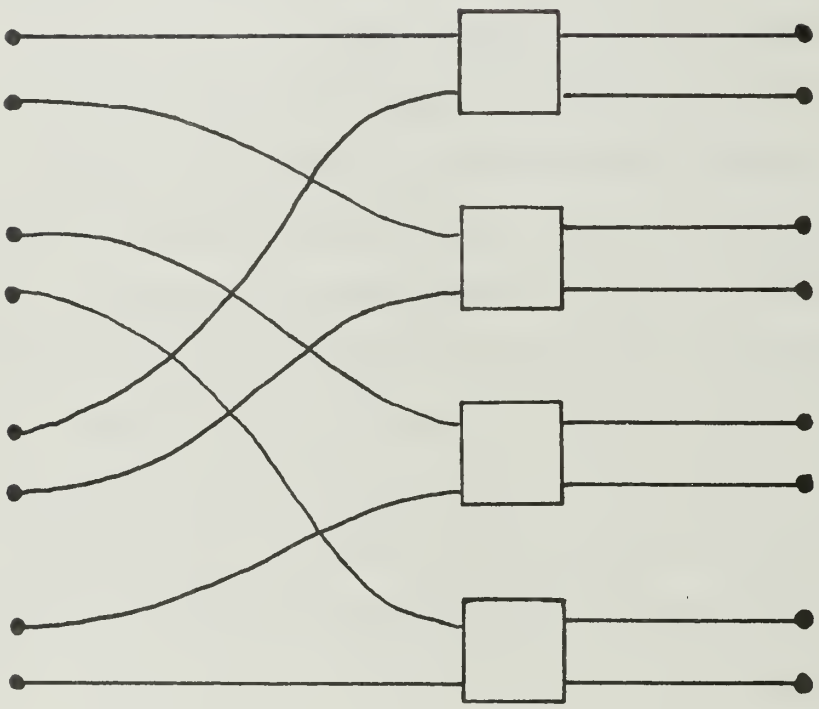


Fig. 21. One stage of an 8-input Shuffle/Exchange Network

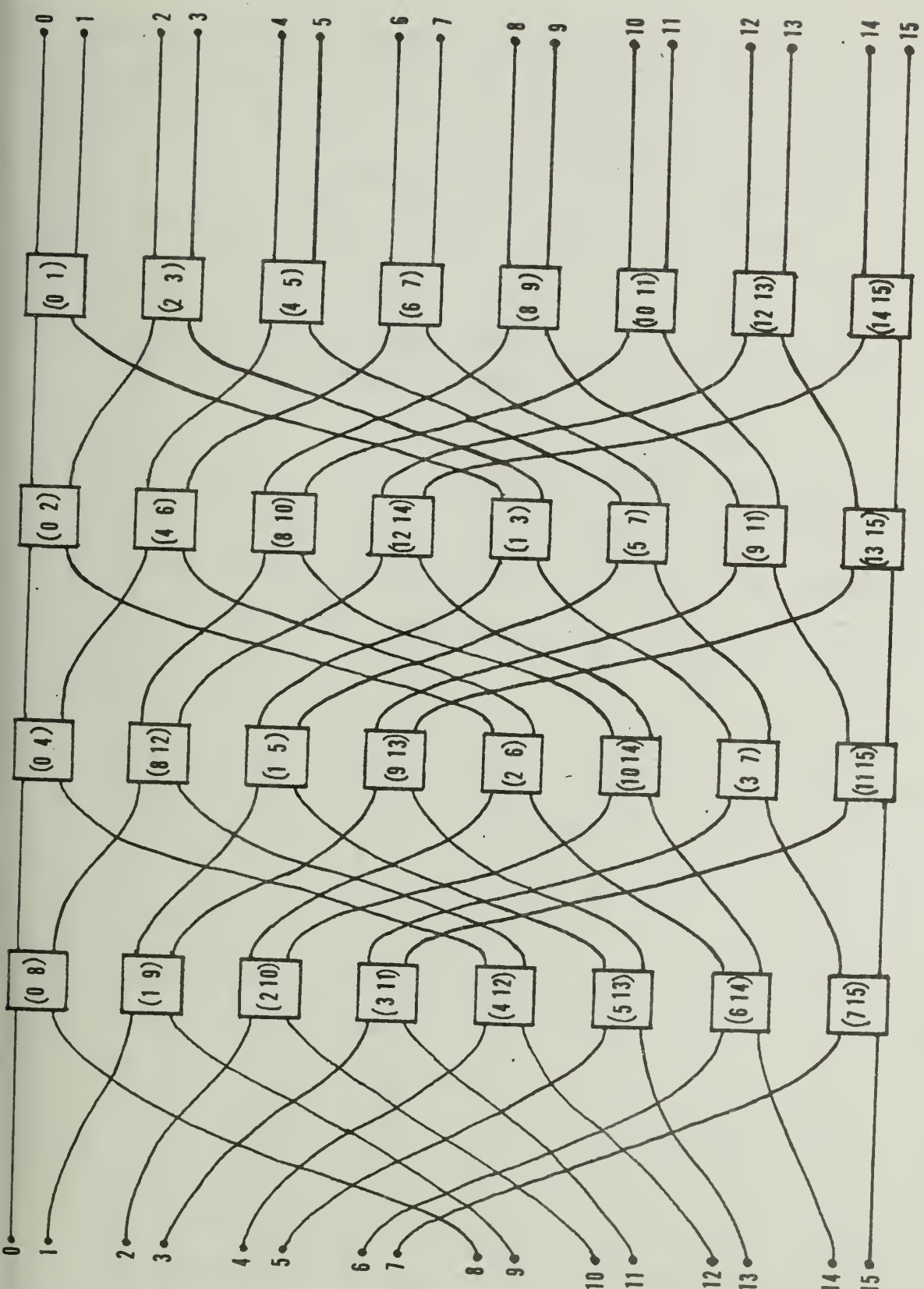


Figure 22. A 16-input Omega Network

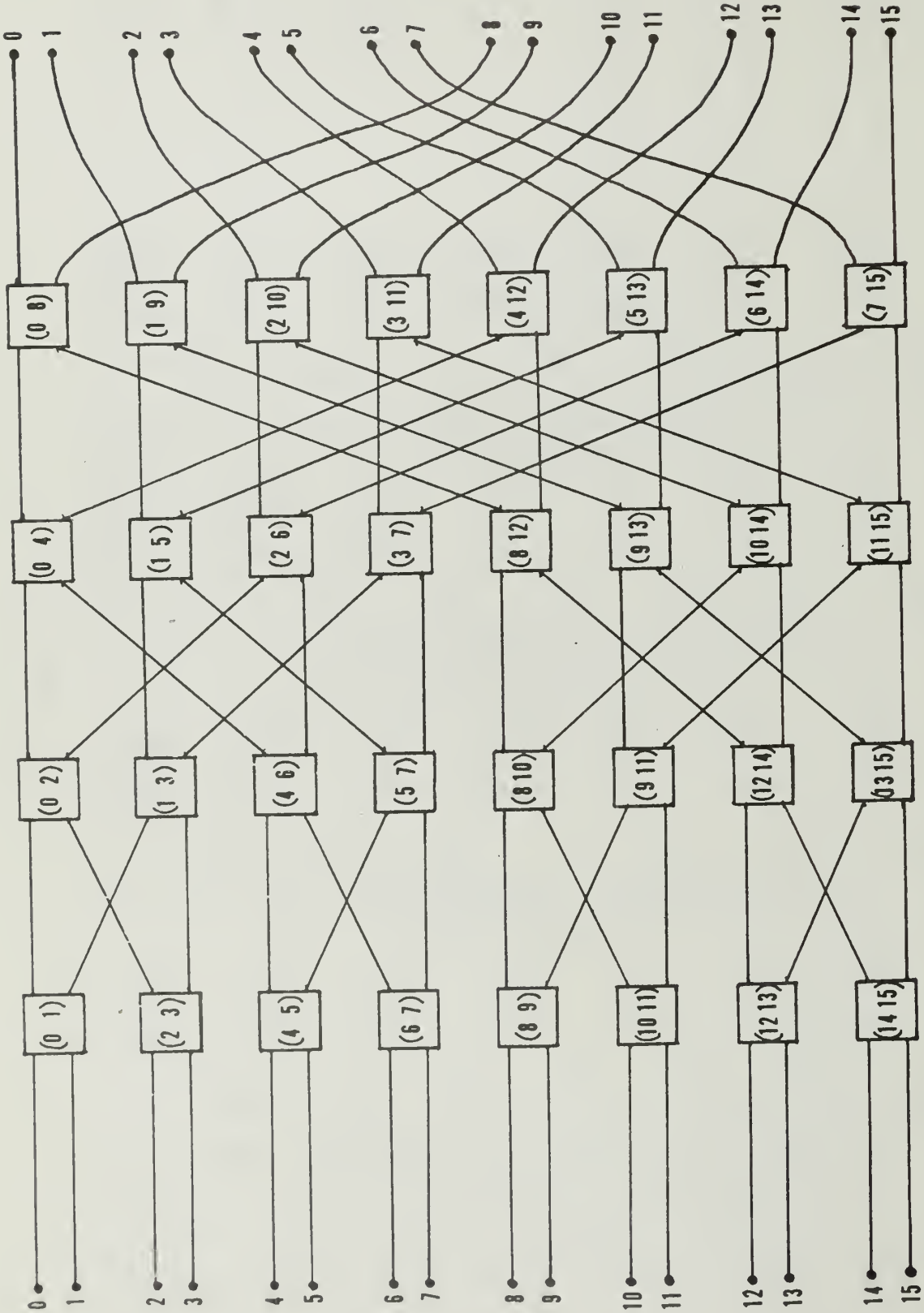


Figure 23. An Indirect Binary 4-Cube Network

R-network (Figure 24). We define the inverse Omega network, as suggested by the N=16 case in Figure 25, to be just lg(N) Exchange/Unshuffles. This is exactly what we would get if we ran the Omega network "backwards", i.e., let data flow from right to left in Figure 22, instead of left to right. Having done this we can make the following statement:

> The inverse Omega network, the indirect binary n-cube, and the R-network are all equivalent.

The word "equivalent" may be interpreted in at least two different ways (topological equivalence or functional equivalence), and in fact results about the equivalence of algorithms may be derived from the statement if one views the networks as operating on the input data, rather than just permuting it.

This claim is probably not obvious, and will be proved in the next section after the necessary tools are developed. Once proved, however, this result is useful since it lets us apply what we know about any one of the networks to the others. One application of this understanding is in showing how the standard FFT algorithm, with a "butterfly" algorithm graph related to the indirect binary n-cube network, can be transformed into Pease's shuffle-based algorithm [Pea 68], or can be transformed into an algorithm based on the R-network with no bit-reversal stage (the transform outputs are produced in correct order).

Having established this network equivalence, we address the topic of the "universality" of these and other networks (their ability to realize arbitrary permutations if multiple passes through them are

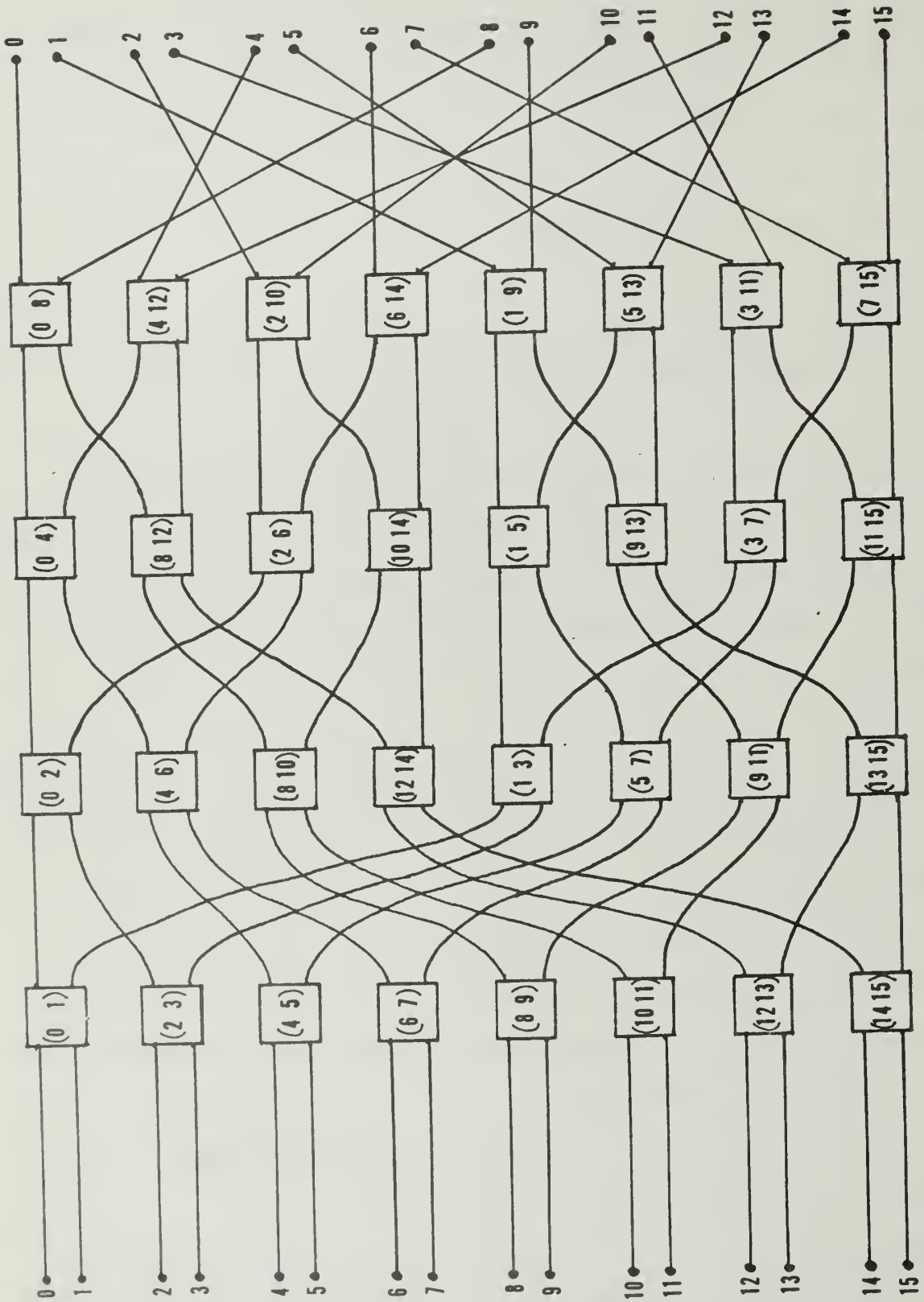


Figure 24. A 16-input R-Network

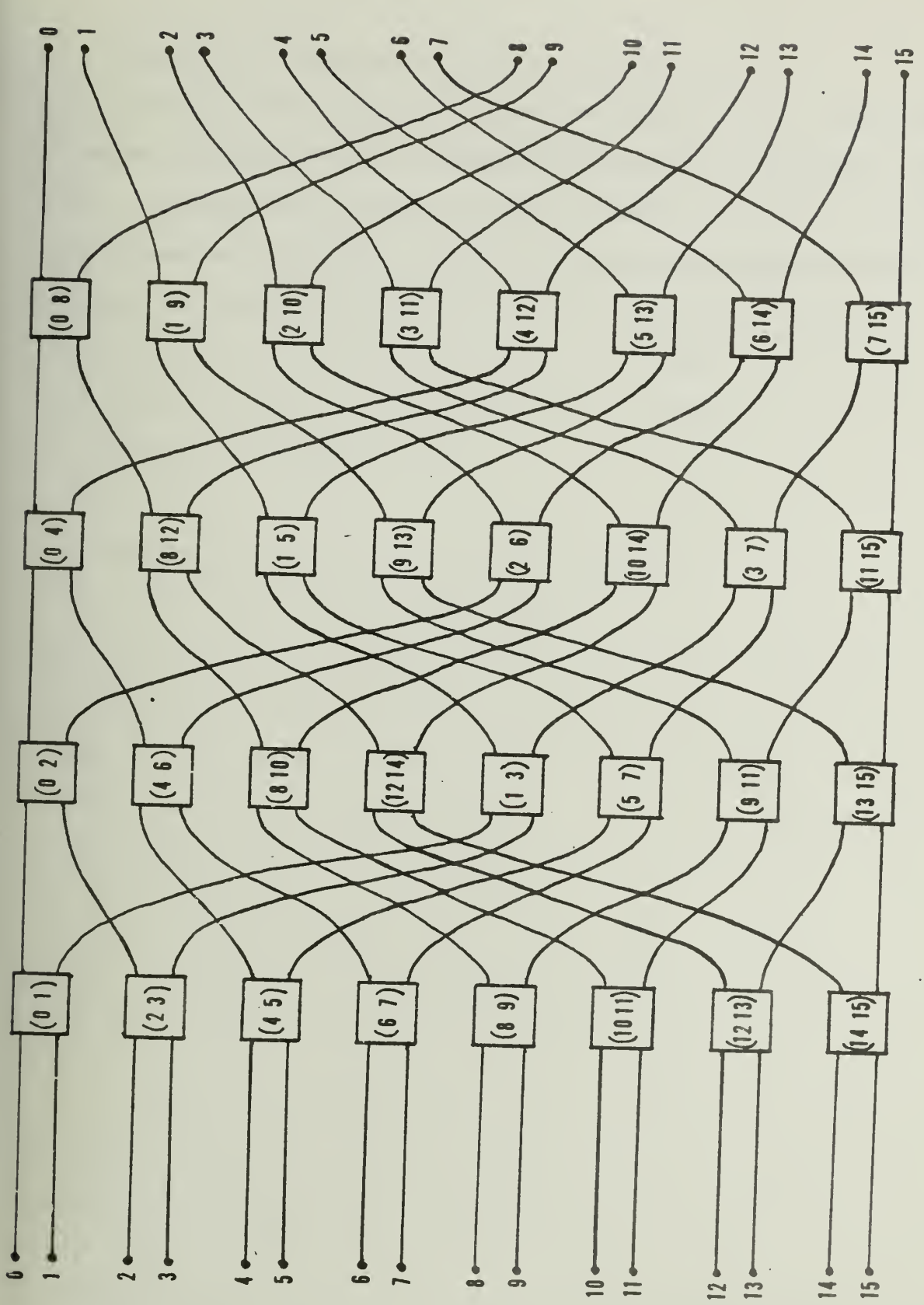


Figure 25. A 16-input Inverse Omega Network

permitted). This question has been touched upon in [Sie 77a] and [Sie 77b]; here it is more fully developed, and the results are shown to have interesting implications on the problem of controlling Shuffle/Exchange-type networks to realize arbitrary permutations or to sort data in parallel.

X. Fundamental Connections and Formal Switch Definitions

In this section we will be concerned only with networks having $N = 2^n$ inputs and outputs, where n is an integer. Although the results presented here generalize for the case where N is not a power of two, we content ourselves for the time being with this restriction. We now define 4 basic permutations which suffice to generate the Omega, indirect binary n-cube, and R-networks -- the shuffle (σ), butterfly (β), bit reversal (ρ), and exchange (E) permutations.

The perfect shuffle permutation σ is defined by

$$\sigma(x) = (2x + \lfloor 2x/N \rfloor) \bmod N$$

where x, the index of some input line (i.e., its order from the top of all N lines), lies between 0 and N-1. See Figure 22. Perhaps a more cogent way of describing this is to say that if $x = [x_n\ x_{n-1} \cdots x_1]$ when x is described in binary notation (so $x = (2^{n-1})x_n + \ldots + (2)x_2 + x_1$) then shuffling corresponds to a circular left shift of the index bits:

$$(1) \qquad \sigma(x) = \sigma([x_n\ x_{n-1} \cdots x_1]) = [x_{n-1}\ x_{n-2} \cdots x_1\ x_n].$$

Thus it is clear that σ^{-1}, the unshuffle, corresponds to a circular right shift. We also define the k^{th} subshuffle $\sigma_{(k)}$, for $1 \leq k \leq n$, by

$$\sigma_{(k)}(x) = \sigma_{(k)}([x_n \ldots x_1]) = [x_n \cdots x_{k+1}\ x_{k-1} \cdots x_1\ x_k].$$

So $\sigma_{(k)}(x)$ is a shuffle on the k least significant bits in x's binary representation, $\sigma_{(1)}(x) = x$, and clearly $\sigma_{(n)} = \sigma$.

Using this notation we define the bit reversal permutation ρ by

$$(2) \qquad \rho([x_n\ x_{n-1} \cdots x_2\ x_1]) = [x_1\ x_2 \cdots x_{n-1}\ x_n]$$

and the <u>k^{th} butterfly permutation $\beta_{(k)}$</u>, for $1 \le k \le n$, by

(3) $$\beta_{(k)}([x_n \ldots x_1]) = [x_n \ldots x_{k+1}\ x_1\ x_{k-1} \ldots x_2\ x_k]$$

-- i.e., the butterfly permutation interchanges the first and k^{th} bits of the index. Also, if we define $\hat{x}$ for all $0 \le x \le N-1$ with $x = [x_n \ldots x_1]$ by

$$\hat{x} = [x_n \ldots x_2\ \bar{x}_1]$$

where the bar indicates Boolean complementation, then we can say that the <u>set E of exchange permutations</u> is

(4) $$E = \{\text{permutations } e \mid \text{for every } 0 \le x \le N-1 \text{ we have either } e(x)=x \text{ and } e(\hat{x}) = \hat{x} \text{ or } e(x)=\hat{x} \text{ and } e(\hat{x}) = x\ \}\ .$$

Note: E is a set with $2^{N/2}$ elements.

The immediate application of these definitions is that they describe the permutations that can be realized using the networks we are concerned with. For example, the set Ω_N of permutations that can be realized by an Omega network is

(5) $$\Omega_N = \sigma E\ \sigma E \ldots \sigma E = (\sigma E)^n \qquad (n = \lg(N)).$$

In writing this expression for repeated Shuffle/Exchanges we imply a left-to-right composition of permutations, so that $\pi_1\ \pi_2(x) = \pi_2(\pi_1(x))$. This possibly confusing convention is chosen to make our permutations conform to the traditional left-to-right flow of data through network drawings; so $\Omega_{16} = \sigma E\ \sigma E\ \sigma E\ \sigma E$ corresponds directly to Figure 22.

This formalism is useful in that it gives us an algebraic grasp on things. Notice that we can derive the expression for those permutations realized by the inverse Omega network:

$$\Omega_N^{-1} = (\Omega_N)^{-1} = ((\sigma E)^n)^{-1} = ((\sigma E)^{-1})^n = (E^{-1}\sigma^{-1})^n$$

and since $E^{-1} = E$,

$$\Omega_N^{-1} = (E\sigma^{-1})^n \quad . \tag{6}$$

Let C_N be the set of permutations realizable by Pease's indirect binary n-cube, and R_N be those realizable by the R-network. Then

$$C_N = E\,\beta_{(2)}\,E\,\beta_{(3)}\,E \cdots E\,\beta_{(n)}\,E\,\sigma^{-1} \tag{7}$$

and

$$R_N = E\,\sigma_{(n)}^{-1}\,E\,\sigma_{(n-1)}^{-1}\,E \cdots E\,\sigma_{(2)}^{-1}\,E\,\rho \quad . \tag{8}$$

Formalizing the claims of section IX, we argue that the following statement is true, establishing the functional equivalence of the networks.

Theorem 7 $\quad \Omega_N^{-1} = C_N = R_N$.

Proof The proof is strictly manipulative. To facilitate its execution we note the following identities, which we state without proof:

$$\rho^{-1} = \rho \tag{10.1}$$

$$\sigma_{(k)}^{-1} = \sigma_{(k)}^{k-1} \qquad k = 1,\ldots,n \tag{10.2}$$

$$\beta_{(k)}^{-1} = \beta_{(k)} \qquad k = 1,\ldots,n \tag{10.3}$$

$$\sigma\rho = \rho\sigma^{-1} \tag{10.4}$$

$$\beta_{(1)} \cdots \beta_{(k)} = \sigma_{(k)} \qquad k = 1,\ldots,n \tag{10.5}$$

(10.6) $$\sigma_{(1)} \cdots \sigma_{(k)} = \rho_{(k)} \qquad k = 1,\ldots,n$$

where $\rho_{(k)}([x_n \cdots x_1]) = [x_n \cdots x_{k+1}\ x_1 x_2 \cdots x_{k-1} x_k]$.

To prove $\Omega_N^{-1} = C_N$, we must show expressions (6) and (7) are equivalent. We start this by noting that

$$\sigma^{-1} E = \beta_{(2)} E \beta_{(2)} \sigma^{-1},$$

an easily verified identity. Thus we have

$$\Omega_N^{-1} = (E\sigma^{-1})^n = E (\beta_{(2)} E \beta_{(2)} \sigma^{-1}) \sigma^{-1} (E \sigma^{-1})^{n-2}$$

$$= E \beta_{(2)} E \beta_{(2)} \sigma^{-2} (E \sigma^{-1})^{n-2}.$$

Now it is also true that

$$\beta_{(2)} \sigma^{-2} E = \beta_{(3)} E \beta_{(3)} \beta_{(2)} \sigma^{-2}$$

so, by substituting again,

$$\Omega_N^{-1} = E \beta_{(2)} E \beta_{(3)} E \beta_{(3)} \beta_{(2)} \sigma^{-3} (E \sigma^{-1})^{n-3}.$$

Since by induction we can show

$$\beta_{(k)} \beta_{(k-1)} \cdots \beta_{(2)} \sigma^{-k} E = \beta_{(k+1)} E \beta_{(k+1)} \beta_{(k)} \cdots \beta_{(2)} \sigma^{-k}$$

we obtain

$$\Omega_N^{-1} = E \beta_{(2)} E \beta_{(3)} E \cdots E \beta_{(n)} E (\beta_{(n)} \cdots \beta_{(2)} \sigma^{-(n-1)}) \sigma^{-1}.$$

Inverting (10.5) and using (10.3) and (10.2) we get

$$\beta_{(n)} \cdots \beta_{(2)} \sigma^{-(n-1)} \sigma^{-1} = \sigma^{-1} \sigma^{-(n-1)} \sigma^{-1} = \sigma^{-1},$$

and by comparison with (7) we conclude $\Omega_N^{-1} = C_N$.

The proof that $\Omega_N^{-1} = R_N$ is similar, except that we make use of

$$\sigma^{-1} E = \sigma_{(n-1)}^{-1} E \sigma_{(n-1)} \sigma^{-1}$$

$$\sigma_{(n-1)} \sigma^{-2} E = \sigma_{(n-2)}^{-1} E \sigma_{(n-2)} \sigma_{(n-1)} \sigma^{-2}$$

$$\sigma_{(2)} \cdots \sigma_{(n-1)} \sigma^{-(n-1)} E = \sigma_{(2)}^{-1} E \sigma_{(2)} \cdots \sigma_{(n-1)} \sigma^{-(n-1)}$$

and apply (10.2) with (10.6), using k = n in both cases. Equivalence of Ω_N^{-1} and R_N follows exactly as above.

We have established now that the three networks are functionally equivalent, i.e., that they all realize the same set of permutations. Actually one can go further and show that the networks are isomorphic, or topologically equivalent, in the sense that all three are only different drawings of the same network. This can be shown directly by a simple adaption of the proof of Theorem 7 which is more careful with what goes on in an Exchange stage -- an isomorphism between control settings for each of the networks may be shown to exist, where the relationship between control settings for any given exchange stage, say between the inverse Omega and Cube networks, is given by a permutation involving shuffles and butterflies. (The permutation can be directly constructed from the proof of Theorem 7.)

What is important, however, is that we have established a convenient formalism for working on networks which is useful for analyzing things besides "permuting power". In the proof of Theorem 7 we were quite vague about the precise function of "E" -- it was simply a set which made identities true. If we generalize the formalism and think of E as being a class of data manipulation operations (instead of just permutations), and think of

strings like (5),(6),(7),(8) as being "programs" (like APL programs, but read left to right of course), then we can still find identities like those in Theorem 7 and prove the equivalence of various programs.

Thus the above analysis can be applied to the study of algorithm structures, provided we can somehow relate a known structure to one of the three networks above or something like them. A case in point is the fast Fourier transform (FFT) algorithm, a widely used algorithm for computing the discrete Fourier transform of a set of N data points. Introductory material may be found in [Coc 67]; we concern ourselves here with the radix-2 form of the algorithm and assume N is a power of 2, but of course the results generalize for more general conditions. When viewed as a network the traditional algorithm may be written as

$$\text{(12)} \qquad \begin{aligned} \text{FFT} &= \beta_{(n)} W \beta_{(n)}^{-1} \beta_{(n-1)} W \beta_{(n-1)}^{-1} \cdots \beta_{(1)} W \beta_{(1)}^{-1} \rho \\ &= \beta_{(n)} W \beta_{(n)} \beta_{(n-1)} W \beta_{(n-1)} \cdots \beta_{(1)} W \beta_{(1)} \rho \end{aligned}$$

where the W operators are columns of N/2 two-input/two-output "multiply-add" units, very much like the exchange operators discussed above. See Figure 26. Because we are concerned with the gross structure of the algorithm, we ignore for the moment the fact that the multiply-add units in each W involve varying powers of a complex root of unity used in the transform; as long as we preserve the topological properties of the network which evaluates the FFT, then these powers still exist and can be determined.

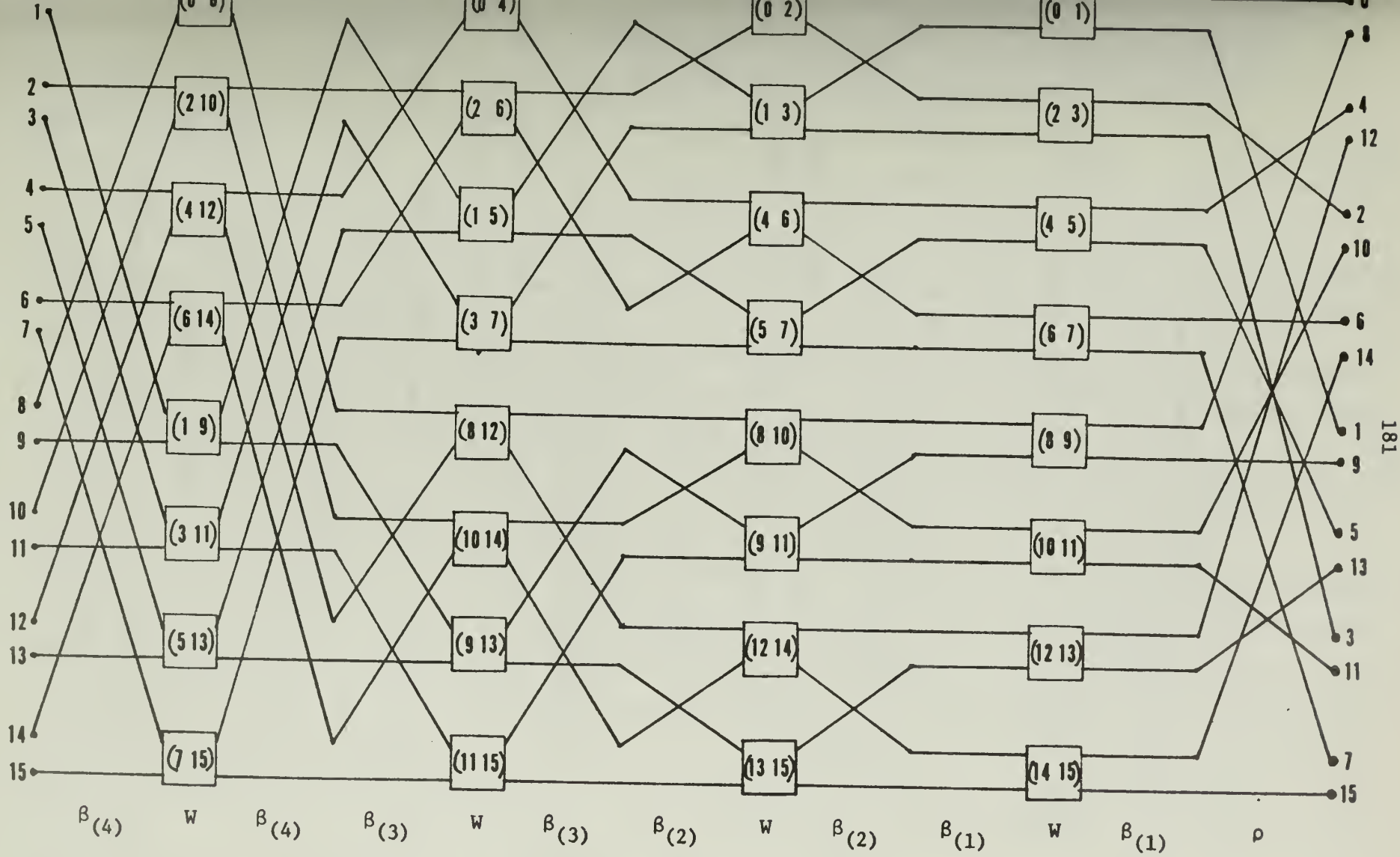


Figure 26. Traditional FFT Algorithm structure

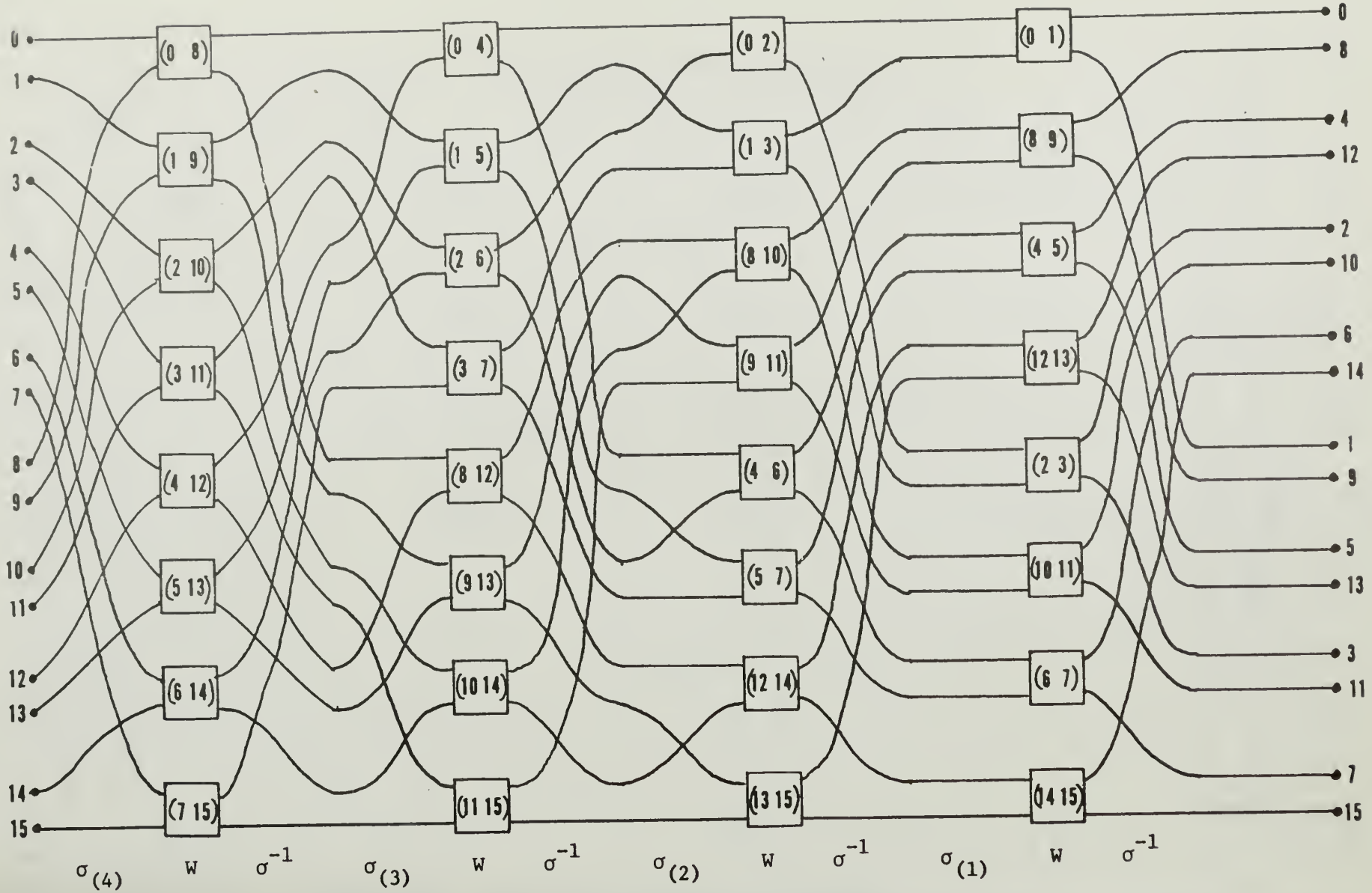


Figure 27. FFT Algorithm without Bit Reversal

If we invert (7), apply (10.3) and replace all occurrences of E with its topological equivalent, W, we obtain

(13) $$C_N^{-1} = \sigma_{(n)} \, W \, \beta_{(n)} \, W \cdots W \, \beta_{(2)} \, W \, .$$

So, if we propagate the easily-verified identity

$$\sigma_{(k)} \, W \, \beta_{(k)} = \beta_{(k)} \, W \, \beta_{(k)} \, \sigma_{(k-1)}$$

from left to right in expression (13), we find

(14) $$C_N^{-1} = \beta_{(n)} \, W \, \beta_{(n)} \, \beta_{(n-1)} \, W \, \beta_{(n-1)} \cdots \beta_{(2)} \, W \, \beta_{(2)} \, W.$$

Direct comparison with (12) yields

(15) $$FFT = C_N^{-1} \, \rho \, .$$

Now we can apply Theorem 7. We discover immediately that

(16) $$FFT = \Omega_N \, \rho = (\sigma W)^n \, \rho \, ,$$

which indicates that the FFT algorithm can be implemented on a network which uses only shuffles and a bit reversal at the end. This is Pease's result [Pea 68]. We also find

(17) $$FFT = R_N^{-1} \, \rho = (\rho \, W \, \sigma_{(2)} \, W \, \sigma_{(3)} \, W \cdots W \, \sigma_{(n)} \, W) \, \rho.$$

The importance of this result is that it leads to an algorithm that uses no bit reversal -- i.e., it leads to an FFT algorithm which produces its outputs in the correct order. If we rewrite (17) using (10.4) as

(18) $$FFT = \sigma \, (\rho\sigma) \, \sigma_{(1)} \, W \, \sigma_{(2)} \, W \cdots W \, \sigma_{(n)} \, W \, \rho$$

and then propagate the $(\rho\sigma)$ in (18) to the right using the relationship

$$(\rho\sigma) \, \sigma_{(k)} \, W = \sigma^{-1} \, \sigma_{(n-k+1)} \, W \, (\rho\sigma)$$

we obtain

$$\text{(19)} \qquad \text{FFT} = \sigma_{(n)} W \sigma^{-1} \; \sigma_{(n-1)} W \sigma^{-1} \cdots \sigma_{(2)} W \sigma^{-1} .$$

The algorithm structure corresponding to (19) is shown in Figure 27. That the FFT could be computed without bit reversal has been known for some time (cf. [Coc 67,p.1671] where Stockham is credited with having developed a procedure for doing it), but the butterfly/bit reversal algorithm has become standard since fast methods for computing bit reversals are known [Pol 74], and since both the butterfly and bit reversal operations can be applied to the array in place -- i.e., no auxiliary storage is necessary to hold results from one stage of the FFT to the next, as would be required by a serial algorithm based on expression (19). The point is, however, that on some machine architectures the traditional algorithm may not work out to be the best. If a machine is equipped to implement $\sigma_{(k)}$ for all k but cannot handle ρ efficiently, then (19) is a better algorithm for that machine.

A surprising by-product of this analysis is the fact that the FFT algorithm (19) may be implemented extremely efficiently on a serial machine if one does not mind using twice as much storage as the traditional FFT. The Unshuffle/Shuffle operations do necessitate the existence of a work array in this algorithm if it is to be coded reasonably; but great savings come in the following observation: in the usual FFT algorithm, at stage k (k = 1,...,n) one "butterflies" the values x_i and $x_{i+2^{n-k}}$ by forming

$$\begin{bmatrix} x_i \\ x_{i+2^{n-k}} \end{bmatrix} \leftarrow \begin{bmatrix} x_i + W^p\, x_{i+2^{n-k}} \\ x_i - W^p\, x_{i+2^{n-k}} \end{bmatrix}$$

where $W = \exp(-2\pi j/N)$, a primitive root of unity, and, if $i = [i_{n-1} \cdots i_0]$ in bit notation, then p is the integer with bit notation

$$p = [i_{n-k}\ i_{n-k+1} \cdots i_{n-1}\ 0 \ldots 0]$$

so at the final stage p is the reverse of i (i.e., $p = \rho(i)$). The computation of W^p is a main part of the inner loop of most FFT routines since it involves evaluating many sines and cosines, as well as bit reversals in evaluating p if the program logic is not rearranged to avoid this. The beautiful thing about (19) is that it reverses the <u>data</u> at each stage just enough that the powers p come out in increasing, non-bit-reversed order as i increases; in fact one can show that, for (19),

$$p = [i_{n-1}\ i_{n-2} \cdots i_{n-k}\ 0 \ldots 0] = i - (i \bmod 2^{n-k})$$

when the algorithm is implemented properly. A simple-minded encoding is shown in Figure 28. Obvious economies can be made on the way the way the values W^p are computed, and the Shuffles and Unshuffles can be moved inside the loop by using more complicated subscript expressions.

This section then has indicated not only that the three networks under consideration are equivalent, but also has shown generally how such networks may be analyzed for equivalence (topological equivalence, or equivalence of the algorithm identified with the network flow graph). Hopefully the reader has obtained some feeling for altering networks

```
W = exp(-2πj/N)
do  k = 1  to  n    /*  lg(N) stages  */
   begin
   execute  σ(n-k+1)   on data
   do  i = 0  to  (N/2)-1   /* (N/2) 2×2 elements/stage  */
      begin
      p = i - (i mod 2^(n-k))
      [x2i  ]  ←  [x2i + W^p x2i+1]
      [x2i+1]     [x2i - W^p x2i+1]
      end
   execute  σ^-1   on data
   end
```

Figure 28. Simple FFT Algorithm without bit reversal

into equivalent ones by manipulating equations involving the permutations ρ, $\beta_{(k)}$, and $\sigma_{(k)}$. There is still a great deal of room left for exploration here: among other things we can show that, like equation (19) above we have the structure

$$\text{FFT} = \sigma W \sigma_{(1)} \sigma W \sigma_{(2)} \cdots \sigma W \sigma_{(n-1)} \sigma W \sigma^{-1} \tag{20}$$

and there are certainly other formulations. Also, although the connections ρ, $\beta_{(k)}$, and $\sigma_{(k)}$ seem very "natural" and, when composed, are capable of generating any permutation of the index bits, there may be other connections which perform "better" for a particular algorithm than these, in the sense that fewer of them or fewer compositions of them are needed to construct a network which executes the algorithm. For example, if there were a connection π such that $\text{FFT} = (\pi W)^n \pi$, then π would be an extremely good connection to have in a Shuffle/Exchange-type network on a multiprocessor which was intended to evaluate FFT's (unfortunately, if π is to be a permutation on the bits of the indices (as are the shuffle and bit reversal, for example) then it is not hard to show that no such π exists for $N > 4$. Any permutation of this type that can handle the FFT must be a cycle of length n, so $\pi^n = 1$). Therefore, the architect of a new multiprocessor with a Shuffle/Exchange-type network should consider which connections will provide him with the most cost-effective switching capability for the programs to be run on the machine, and include all of them -- so the Shuffle/Exchange will comprise more connections than just a shuffle. An FFT processor will probably want to include both σ and ρ; Gajski has

shown [Gaj 77] that a processor which solves linear recurrences would work best if $\sigma_{(k)}$ for k = 2,...,n were all available. Note that the bit reversal, butterfly, and shuffle permutations can all be generated using only σ and $\sigma_{(2)}$, but they are not <u>easily</u> generated. The problem of generating arbitrary index-bit permutations (therefore <u>a fortiori</u> bit reversal, shuffles, etc.) has been studied in the context of bubble memories by Wong and Coppersmith [WC 76].

XI. Universality of Shuffle/Exchange-type Networks

In the previous section we concerned ourselves with topological properties of Shuffle/Exchange networks, and how the same network could be represented in a number of different ways. We now address the problem of determining their inherent "permuting power", or "universality" in the terminology of Siegel [Sie 77b]. That is, we want to know whether we can realize arbitrary permutations of the inputs with the Shuffle/Exchange networks if multiple passes through them are allowed. Note that this characterization of the networks has really nothing to do with the way the network is drawn -- by Theorem 7, we know that the inverse Omega, indirect binary n-cube, and RSN-derived networks have identical permuting capabilities although they look different.

Let S_N denote the group of all permutations on N lines. We now ask the question: what is the smallest value of k such that

(21) $$S_N \subset (\Omega_N)^k$$

--i.e., how many passes through the Omega network are necessary to ensure that any permutation can be generated? Clearly the value of k that works for Ω_N will also work for Ω_N^{-1}, R_N, and C_N, so we can restrict our attention to Ω_N here and answer the question for all these networks. One might also ask the smallest value of ℓ such that

(22) $$S_N \subset (\sigma E)^\ell$$

but clearly $(k - \ell/(\lg(N))) < 1$ so we have $\ell = k \lg(N)$ approximately, and since the behavior of the Omega networks is simpler to analyze than that of bare Shuffle/Exchanges, we will only derive bounds on k here. It would be interesting if we could show $\ell = k \lg(N)$ exactly.

A word about why we are interested in (21) is in order. We are concerned with the ability of our interconnection network (in this case, an Omega network used iteratively) to permute the input values to output lines, as would be necessitated by processor-memory interaction in a multiprocessor architecture. It is true, in a multiprocessor environment like this the network will be typically requested to implement connections between inputs and outputs that are not perfect permutations. For example, two processors might reference the same memory module, or one processor might broadcast data to several memory modules. Temporarily we restrict our attention to S_N and (21), and address the problem of generalized connections immediately afterwards (the extension is easy).

A simple cardinality argument shows k=1 is impossible in (21) for $N > 2$, since

$$|| S_N || = N! \sim (\sqrt{2\pi}) \; 2^{(N+1/2)\lg(N) - N\lg(e)}$$

and

$$|| \Omega_N || = 2^{(N/2)\lg(N)} .$$

However k=2 cannot be immediately rejected, since it is possible that $(\Omega_N)^2$ contains enough elements. In fact, it is easy to show that $S_4 \subset (\Omega_4)^2$, and a computer program run on the IBM 360/75 here -- requiring 15 minutes of CPU time to evaluate all 16 million elements in $(\Omega_8)^2$ -- demonstrated that $S_8 \subset (\Omega_8)^2$ (moreover, every permutation in S_8 could be realized in at least 288, and at most 776, ways). It is tempting to

conclude that k=2 makes (21) true in general, but this seems difficult to prove. Also, the author has not been able to find a two-pass solution for realizing the permutation (0 15) -- an interchange of lines 0 and 15 -- when N=16.

It is clear that since Batcher's bitonic sorting can be implemented on a Shuffle/Exchange architecture in $(\lg(N))^2$ stages [Sto 71],[Sie 77b], we have $k = \lg(N)$ as an upper bound for (21). Thus we have established the limits $2 \leq k \leq \lg(N)$. The following theorem improves the upper bound considerably.

Theorem 8 Any permutation can be realized with $\min(6,\lg(N))$ Ω_N-passes; i.e., $$S_N \subset (\Omega_N)^{\min(6,\lg(N))}.$$

Proof This non-intuitive result (which is not the best possible) is established in a sequence of lemmas. As always we assume N is a power of two.

Lemma 1 $$S_N \subset \Omega_N^{-1}\,\Omega_N$$

Proof Hall's theorem on systems of distinct representatives can be used to show that a rearrangeable switching network (RSN) can realize any permutation of its inputs [Ben 65]. Therefore, $S_N \subset R_N\,R_N^{-1}$. Now apply Theorem 7.

Lemma 2 $\qquad \Omega_N^{-1} = \rho\, \Omega_N\, \rho$

Proof This follows immediately from the fact that $\rho\sigma\, E = E\, \sigma^{-1}\rho$, so we can propagate the leftmost ρ above rightward using equation (5) and wind up with equation (6), which gives Ω_N^{-1}.

Lemma 3 The bit reversal permutation ρ can be realized in two Ω_N-passes, i.e., $\qquad \rho \in (\Omega_N)^2$.

Proof This is a corollary of the work of Pease [Pea 77]. Notice that

$$\rho([x_n \ldots x_1]) = P\,[x_n \ldots x_1]^t$$

where P is the matrix

$$P = \begin{pmatrix} & & & & 1 \\ & & & 1 & \\ & & \cdot & & \\ & \cdot & & & \\ & \cdot & & & \\ 1 & & & & \end{pmatrix}_{nxn}$$

Now P can be decomposed as $P = L\, U\, L$ where L and U are the matrices

$$L = \begin{pmatrix} 1 & & & & \\ & 1 & & & \\ & & \cdot & & \\ & & & \cdot & \\ & & \cdot & \cdot & \\ & 1 & & & 1 \\ 1 & & & & 1 \end{pmatrix}_{nxn} \qquad U = \begin{pmatrix} 1 & & & & 1 \\ & 1 & & 1 & \\ & & \cdot & \cdot & \\ & & & \cdot & \\ & & & & 1 \\ & & & & 1 \end{pmatrix}_{nxn}$$

Pease has shown that any permutation $y = \pi(x)$ such that

$$y = LU\, x$$

where L is a lower triangular and U a unit upper triangular nxn matrix can be realized with a single Ω_N pass. Therefore since

$$\rho(x) = P\, x = LU\, (L\, x)$$

ρ can be realized in two passes, or $\rho \in (\Omega_N)^2$. In fact we have

$\rho(x) = \omega_2(\omega_1(x))$ where

(23) $\omega_1([x_n \ldots x_1]) = [x_n\ x_{n-1} \cdots x_{n/2}\ (x_{n/2} \oplus x_{n/2+1}) \cdots (x_1 \oplus x_n)]$

and

(24) $\omega_2([x_n \ldots x_1]) = [(x_n \oplus x_1)\ (x_{n-1} \oplus x_2) \cdots (x_{n/2+1} \oplus x_{n/2})\ x_{n/2} \cdots x_n]$

(assuming that n is even; the case where n is odd is similar), and it is easy to verify that both ω_1 and ω_2 are in Ω_N. For an example with N=8 see Figure 29.

Theorem 8 now follows immediately. We know

(25) $S_N \subset \Omega_N^{-1}\ \Omega_N$ (Lemma 1)

(26) $= \rho\ \Omega_N\ \rho\ \Omega_N$ (Lemma 2)

(27) $\subset (\Omega_N)^2\ \Omega_N\ (\Omega_N)^2\ \Omega_N = \Omega_N^6.$ (Lemma 3)

Thus $S_N \subset (\Omega_N)^6$ holds for all N, but since we know $S_N \subset (\Omega_N)^{lg(N)}$ for all N as well, we obtain the result stated in the theorem. Note that this result is pretty crude, once the principles behind it have been grasped. We can refine it a bit :

Theorem 9 $S_N \subset (\Omega_N)^{\min(4,lg(N))}.$

Proof We obtain the value 4 by showing that $\rho\ \Omega_N \subset (\Omega_N)^2$, and use this result immediately in equation (26) above. We do this as follows. Pease [Pea 77] has characterized exactly which permutations π are in Ω_N^{-1}, showing that

(28) $\Omega_N^{-1} = \{\ \pi(x) = y \mid y_i = x_i \oplus f_i(y_1,\ldots,y_{i-1},x_{i+1},\ldots,x_n)\ \}$

$$P = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} = LUL \qquad LU = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \qquad L = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix}$$

$$\omega_1([x_3\ x_2\ x_1]) = L\,[x_3\ x_2\ x_1]^t = [x_3\ x_2\ (x_3 \oplus x_1)] = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 0 & 1 & 2 & 3 & 5 & 4 & 7 & 6 \end{pmatrix}$$

$$\omega_2([x_3\ x_2\ x_1]) = LU\,[x_3\ x_2\ x_1]^t = [(x_3 \oplus x_1)\ x_2\ x_3] = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 0 & 4 & 2 & 6 & 5 & 1 & 7 & 3 \end{pmatrix}$$

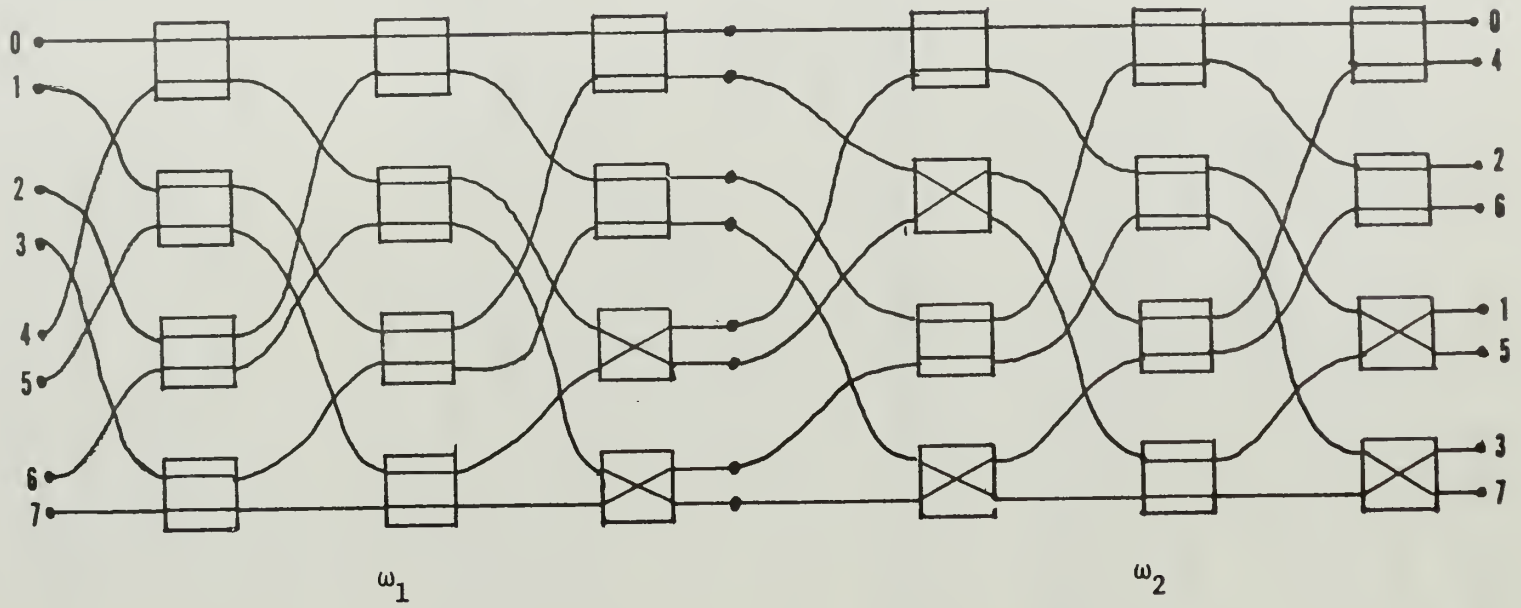


Fig. 29. Demonstration that ρ is in $(\Omega_8)^2$

where the f_i are arbitrary Boolean functions of n-1 variables. From this it follows immediately from Lemma 2 above that

$$(29)\quad \Omega_N = \{\, \pi(x) = y \mid y_{n-i+1} = x_{n-i+1} \oplus f_i(y_n,\ldots,y_{n-i+2},x_{n-i},\ldots,x_1) \,\}$$

and

$$(30)\quad \rho\,\Omega_N = \{\, \pi(x) = y \mid y_{n-i+1} = x_i \oplus f_i(y_n,\ldots,y_{n-i+2},x_{i+1},\ldots,x_n) \,\}.$$

We show that every permutation π in $\rho\,\Omega_N$ can be realized as a sequence of two permutations in Ω_N, where in fact the first permutation is just ω_2 defined in (24). Every permutation π in $\rho\,\Omega_N$ determines a unique set of Boolean functions $\{ f_i \mid i=1,\ldots,n \}$, as indicated by (30). Thus for every such π we can associate a permutation ω_π in Ω_N defined as follows: if $y = \omega_\pi(z)$, so $[y_n \ldots y_1] = \omega_\pi([z_n \ldots z_1])$, then

$$y_{n-i+1} = z_{n-i+1} \oplus g_i(y_n,\ldots,y_{n-i+2},z_{n-i},\ldots,z_1)$$

where the g_i are in turn defined by

$$g_i(y_n,\ldots,y_{n-i+2},z_{n-i},\ldots,z_1)$$

$$= \begin{cases} f_i(y_n,\ldots,y_{n-i+2},z_{n-i},\ldots,z_1) & \text{if } i > n/2 \\ z_i \oplus f_i(y_n,\ldots,y_{n-i+2},(z_{n-i}\oplus z_{i+1}),\ldots,(z_{n/2}\oplus z_{n/2+1}),z_{n/2},\ldots,z_1) & \text{if } i \le n/2 \text{ and } n \text{ is even} \\ z_i \oplus f_i(y_n,\ldots,y_{n-i+2},(z_{n-i}\oplus z_{i+1}),\ldots,(z_{\lceil n/2\rceil+1}\oplus z_{\lceil n/2\rceil-1}),z_{\lceil n/2\rceil},\ldots,z_1) & \text{if } i \le n/2 \text{ and } n \text{ is odd.} \end{cases}$$

With these definitions it is simple to verify $\pi(x) = \omega_\pi(\omega_2(x))$ is true for all x and π. Thus we have shown $\rho\,\Omega_N \subset (\Omega_N)^2$, and Theorem 9 has been proved.

In summary, then, we have shown that

$$S_N \subset (\Omega_N)^4 \tag{31}$$

is true for all N, giving the nice bounds $2 \leq k \leq 4$ on the least value of k that will satisfy (21). Again, the construction used for k=4 seems crude when one takes into consideration that two of the four Omega passes realize only the constant permutation ω_2 -- an apparently wasteful situation. However it is not clear that k=2 or even k=3 will suffice, and it remains an open question to determine precisely which k is minimal for any value N > 8.

Theorems 8 and 9 lead to some interesting derivatives. Define the F-network to be any network which realizes the set of permutations

$$F_N = \Omega_N \rho \ . \tag{32}$$

Then we can make three observations: first, (16) tells us that the F-network can be used to implement FFT's. Second, we have

$$S_N \subset (F_N)^2 \tag{33}$$

since $S_N = \rho\, S_N\, \rho \subset \rho\, (\rho\Omega_N\rho\Omega_N)\, \rho = (\Omega_N\rho)^2$. Thus the analogue of (21) for F-networks is always satisfied by k=2. Third, and most surprisingly, noting that $F_N^{-1} = (\rho\, \Omega_N)^{-1} = \Omega_N^{-1}\, \rho = (\rho\, \Omega_N\, \rho)\, \rho = \rho\, \Omega_N$, we find

$$F_N = F_N^{-1} \tag{34}$$

and the network turns out to be topologically equivalent to itself when run "backwards". It remains to be seen whether these properties have any useful implications; disappointingly, the F-network is incapable of realizing the identity permutation.

The result (31) is interesting for the following reason. When the author originally formulated the question of finding the least value of k satisfying (21) he was confident that the answer would be $k = \lg(N)$, which would have suggested that Batcher's algorithm is essentially optimal for sorting or realizing arbitrary permutations on a Shuffle/Exchange network. However, (31) implies that the optimality of Batcher's method is not so easy to demonstrate. In fact, Lemmas 1-3 and Theorem 8 say that if we can find a rapid control algorithm for the RSN, then we can realize arbitrary permutations in a small constant number of Ω_N-passes, significantly less than the $\lg(N)$ passes required by bitonic sorting. (Unfortunately, the best known RSN control algorithm at the moment is related to the "Looping algorithm" of Opferman and Tsao-Wu [OTW 71], and takes O(N) steps to execute.) This brings the complexity of controlling the network under focus as the main question concerning us -- notice that the Batcher algorithm uses a strictly local control process (pairwise comparisons), while the RSN control algorithm is strictly global. The Batcher algorithm thus sacrifices a number of Ω_N-passes for control simplicity, whereas the RSN algorithm takes the opposite tack. We ask: is there a semi-global control strategy lying between the strictly local Batcher and strictly global RSN algorithms which uses both a modest amount of control time and a limited number of Ω_N-passes? Wen [Wen 76] has done some preliminary work indicating that this is probably the case, at least for the average number of passes required by random permutations. Results here could have great impact on the design of processor-memory interconnection networks.

Finally we bring up, as promised, the problem of making arbitrary connections using Shuffle/Exchange-type networks. Note first of all that if m inputs all desire to come out on the same output line, then we have no choice but to delay (m-1) of them, and let these through one at a time after the first one has reached its destination. Thus at least m Ω_N-passes are needed to transfer all the data through the network, and there is little else to say. If, however, one input desires to be "broadcasted" (in the sense of [Law 76]) to m outputs, then it is still possible that a small constant number of Ω_N-passes could suffice. Let T_N be the set of arbitrary connections between all N inputs and outputs; then clearly $S_N \subset T_N$ and

$$|| T_N || = N^N. \tag{35}$$

Unfortunately the RSN is not powerful enough to cover all of T_N, or even arbitrary broadcast patterns, so we do not have

$$T_N \subset \Omega_N^{-1} \Omega_N$$

and Theorems 8 and 9 do not hold for generalized connections. Fortunately the problem has been studied and Thompson has derived several useful results in [Tho 77]. He defines a Generalized Connection Network (GCN) with the network structure/set of connections

$$\begin{aligned} G_N = {} & B\ \beta_{(2)}\ B\ \beta_{(3)}\ B \cdots B\ \beta_{(n)}\ B\ \beta_{(n-1)}\ B \cdots B\ \beta_{(2)}\ B \cdots B \\ & \beta_{(n-1)}\ B\ \beta_{(n)}\ B \cdots B\ \beta_{(2)}\ B \\ = {} & C_N\ \sigma\ (B\ \beta_{(n-1)}\ B \cdots B\ \beta_{(2)}\ B \cdots B\ \beta_{(n-1)}\ B)\ \sigma^{-1}\ C_N^{-1} \end{aligned} \tag{36}$$

where C_N denotes (7) with E's replaced by B's, and B is the set of all

connections realizable with a column of exchange elements when broadcasting is permitted. Following (4) we can formally define

(37) $B = \{$ connections b | for every $0 \le x \le N-1$ we have
either $b(x)=x$ and $b(\hat{x})=\hat{x}$
or $b(x)=\hat{x}$ and $b(\hat{x})=x$
or $b(x)=b(\hat{x})=x$
or $b(x)=b(\hat{x})=\hat{x}$ $\}$.

Thus B is a set with $4^{N/2} = 2^N$ elements. Thompson has shown that $T_N \subset G_N$; now, since it is easy to manipulate (36) to show

$$G_N \subset C_N C_N^{-1} C_N C_N^{-1}$$

we can apply the old results above. Using Theorem 7, Lemma 2, and Theorem 9 (where all E's are replaced by B's) we get

(38)
$$\begin{aligned} T_N &\subset G_N \\ &\subset C_N C_N^{-1} C_N C_N^{-1} \\ &= \Omega_N^{-1} \Omega_N \Omega_N^{-1} \Omega_N \\ &= (\rho\Omega_N\rho)\ \Omega_N\ (\rho\Omega_N\rho)\ \Omega_N \\ &= (\rho\Omega_N)^4 \\ &\subset (\Omega_N)^8. \end{aligned}$$

We have therefore shown that arbitrary broadcast patterns can be realized in 8 Ω_N-passes, given that the exchange elements are equipped to make upper or lower broadcasts [Law 76]. Alternately we can use the same argument to establish this result using 4 F-network passes. Once again, whether these pass counts can be further refined is an open question.

XII. References

[Bat 68] Batcher, K.E. "Sorting Networks and Their Applications". Proc. AFIPS SJCC 32, 307-314 (1968).

[Ben 65] Beneš, V.E. Mathematical Theory of Connecting Networks and Telephone Traffic. NY: Academic Press, 1965.

[Ben 75a] _____. "Applications of Group Theory to Connecting Networks". BSTJ 54, 2, 407-420 (Feb. 1975).

[Ben 75b] _____. "Proving the Rearrangeability of Connecting Networks by Group Calculations". BSTJ 54, 2, 421-434 (Feb. 1975).

[Ben 75c] _____. "Towards a Group-Theoretic Proof of the Rearrangeability Theorem for Clos' Network". BSTJ 55, 4, 797-805 (Apr. 1975).

[Ber 62] Berge, C. The Theory of Graphs and its Applications. NY: John Wiley & Sons, Inc., 1962.

[Coc 67] Cochran, W.T. et al. "What is the Fast Fourier Transform?" Proc. IEEE 55, 10, 1664-1674 (Oct. 1967).

[Clo 53] Clos, C. "A Study of Non-Blocking Switching Networks". BSTJ 32, 2, 406-424 (Mar. 1953).

[FS 77a] Feierbach, G. and D. Stevenson. "Processor Interconnection Networks, Some New Results". in High-Speed Computer and Algorithm Organization, ed. by D.J. Kuck, D.H. Lawrie, and A.H. Sameh. NY: Academic Press, 1977.

[FS 77b] _____. "A Feasibility Study of Programmable Switching Networks for Data Routing". Phoenix Project Memorandum #003, Inst. for Advanced Comp., Sunnyvale, CA, 1977.

[Gaj 77] Gajski, D.D. "Processor Arrays for Computing Linear Recurrence Systems". Proc. International Conf. on Parallel Processing, August 23-25, 1978.

[GK 74] Gold, D.E. & D.J. Kuck. "A Model for Masking Rotational Latency by Dynamic Disk Allocation". CACM 17, 5, 278-288(May 1974).

[Joe 68] Joel, A.E., Jr. "On Permutation Switching Networks". BSTJ 47, 5, 813-822 (May-June 1968).

[Lang76] Lang, T. "Interconnections between Processors and Memory Modules using the Shuffle-Exchange Network". IEEE Trans. Comput. C-25, 5, 496-503 (May 1976).

[LS 76] _____ & H.S. Stone. "A Shuffle-Exchange Network with Simplified Control". IEEE Trans. Comput. C-25, 1, 55-65 (Jan. 76).

[Law 76] Lawrie, D.H. "Access and Alignment of Data in an Array Processor". IEEE Trans. Comput. C-25, 12, 1145-1155 (Dec. 1976).

[MP 75] Muller, D. & F.P. Preparata. "Bounds to Complexities of Networks for Sorting and Switching". JACM 22, 2, 195-201 (Apr. 1975).

[Nei 69] Neiman, V.I. "Structure et Commande Optimales de Réseaux de Connexion Sans Blocage". Annales des Télécommunications 24, 7-8, 232-238 (July/Aug. 1969).

[OTW 71] Opferman, D.C. & N.T. Tsao-Wu. "On a Class of Rearrangeable Switching Networks," "Part I: Control Algorithm," "Part II: Enumeration Studies and Fault Diagnosis," both in BSTJ 50, 5, 1579-1600 and 1601-1618 (May-June 1971).

[Pea 68] Pease, M.C. "An Adaptation of the Fast Fourier Transform for Parallel Processing". JACM 15, 2, 252-264 (Apr. 1968).

[Pea 69] _____. "Organization of Large Scale Fourier Processors". JACM 16, 3, 474-482 (July 1969).

[Pea 77] _____. "The Indirect Binary n-Cube Microprocessor Array". IEEE Trans. Comput. C-26, 5, 458-473 (May 1977).

[Pol 74] Polge, R.J. et al. "Fast Computational Algorithms for Bit Reversal". IEEE Trans. Comput. C-23, 1, 1-9 (Jan. 1974).

[Ram 73] Ramanujam, H.R. "Decomposition of Permutation Networks". IEEE Trans. Comput. C-22, 7, 639-643 (July 1973).

[SL 66] Shepp, L.A. and S.P. Lloyd. "Ordered Cycle Lengths in a Random Permutation". Trans. AMS 121, 2, 340-357 (Feb. 1966).

[Sie 77a] Siegel, H.J. "Analysis Techniques for SIMD Machine Interconnection Networks and the Effects of Processor Address Masks". IEEE Trans. Comput. C-26, 2, 153-161 (Feb 1977).

[Sie 77b] _____. "The Universality of Various Types of SIMD Machine Interconnection Networks". Proc. 4th Annual Symposium on Computer Architecture, March 23-25, 1977.

[Sto 71] Stone, H.S. "Parallel Processing with the Perfect Shuffle". IEEE Trans. Comput. C-20, 2, 153-161 (Feb. 1971).

[Tho 77] Thompson, C.D. "Generalized Connection Networks for Parallel Processor Intercommunication". Technical report, Carnegie-Mellon University, May 1977.

[Thu 71] Thurber, K.J. "Permutation Switching Networks". Proceedings of the Technical Program, 1971 Computer Designer's Conference, Anaheim, CA, Jan. 1971, pp.7-24.

[TW 74] Tsao-Wu, N.T. "On Neiman's Control Algorithm for the Control of Rearrangeable Switching Networks". IEEE Trans. Commun. COM-22, 6, 737-742 (June 1974).

[Wak 68] Waksman, A. "A Permutation Network". JACM 15, 1, 159-163 (Jan. 1968).

[Wen 75] Wen, K-Y. "Interprocessor Connections -- Capabilities, Exploitation and Effectiveness". Rept. No. UIUCDCS-R-76-830, Dept. of Computer Science, Univ. of Illinois,at Urbana-Champaign, Oct. 1976.

[WC 76] Wong, C.K. & D. Coppersmith. "The Generation of Permutations in Magnetic Bubble Memories". IEEE Trans. Comput. C-25, 3, 254-262 (March 1976).

Vita

D. Stott Parker, Jr. was born in New Haven, Connecticut on December 31, 1952. After growing up in California and Texas he attended Princeton University, where he received an A.B. in Mathematics (cum laude) in 1974. He started graduate work at the University of Illinois that fall, receiving the M.S. in Computer Science in May 1976 and the Ph.D. in Computer Science in October 1978. Following a brief post-doctoral period of study at the Université de Grenoble, he plans to join the faculty at the University of California, Los Angeles, in January 1979. He is a member of the ACM, IEEE, AMS, and Sigma Xi.

BIBLIOGRAPHIC DATA SHEET	1. Report No. UIUCDCS-R-78-930	2.	3. Recipient's Accession No.
4. Title and Subtitle: STUDIES IN CONJUGATION: HUFFMAN TREE CONSTRUCTION, NONLINEAR RECURRENCES, AND PERMUTATION NETWORKS			5. Report Date: July, 1978 6.
7. Author(s): Douglass Stott Parker, Jr.			8. Performing Organization Rept. No. UIUCDCS-R-78-930
9. Performing Organization Name and Address: University of Illinois at Urbana-Champaign, Department of Computer Science, Urbana, Illinois 61801			10. Project/Task/Work Unit No. 11. Contract/Grant No. US NSF MCS73-07980
12. Sponsoring Organization Name and Address: National Science Foundation, Washington, D. C.			13. Type of Report & Period Covered: Doctoral Dissertation 14.

15. Supplementary Notes

16. Abstracts

This dissertation demonstrates that conjugation (the replacement, loosely speaking, of some function or computational structure F by $\phi \circ F \circ \phi^{-1}$, where $\circ$ denotes composition and ϕ is a "change of variables") is a technique that is useful in attacking the three problems listed in the title. Basically the results are, first, that the Huffman algorithm--a well-known algorithm for constructing weighted trees--produces optimal trees whenever the weight of a parent node in the tree is given by a linear, or conjugate-to-linear, function of its sons' weights, and a tree's cost is measured by a concave Schur function of the tree weights. Second, that many nonlinear recurrences (in particular, all first-order analytic iterations) may be conjugated into linear recurrences on certain domains, and hence solved quickly on a parallel machine. Third and last, that results concerning Shuffle/Exchange networks and Benes̆ networks (built from smaller networks using conjugating Shuffle/Unshuffle connections) can be derived from new algebraic perspectives on their structure. New control algorithms for the Benes̆-type network are exhibited which are asymptotically faster than previously known ones, but unfortunately are not good enough to be practical. Equivalence between three variants of the Shuffle/Exchange network is then proved, giving as a by-product an elegant Fast Fourier Transform algorithm which produces outputs in correct order without the normally required bit-reversal permutation of the data, and it is then shown that only $4 \log_2 N$ Shuffle/Exchange passes is sufficient to generate any permutation of N lines, which is surprising since implementation of a Batcher network requires $(\log_2 N)^2$ passes.

17. Key Words and Document Analysis. 17a. Descriptors

17b. ~~Identifiers/Open-Ended Terms~~ Key Words:

Huffman trees
Nonlinear recurrences
Optimal trees
Parallel computation
Switching networks

17c. COSATI Field/Group

18. Availability Statement: Release Unlimited	19. Security Class (This Report) UNCLASSIFIED	21. No. of Pages 209
	20. Security Class (This Page) UNCLASSIFIED	22. Price

FORM NTIS-35 (10-70) USCOMM-DC 40329-P71