DECOMPOSITION OF BOOLEAN FUNCTIONS ON A NETWORK OF POLYFUNCTIONAL NODES

Rik A. Verstraete
Jacques J. Vidal

February 1987
CSD-870008

DECOMPOSITION OF BOOLEAN FUNCTIONS ON A NETWORK OF POLYFUNCTIONAL NODES

Rik A. Verstraete
Jacques J. Vidal

Distributed Machine Intelligence Laboratory
Computer Science Department
University of California, Los Angeles

Networks of fixed topology but flexible functionality have received very little attention in the literature. Indeed, problems of combinational circuit design invariably deal with the question of how to interconnect a minimal number of Boolean gates (AND-gates or OR-gates) in order to implement a given Boolean function. Gate array technology is a typical embodiment of this work. In VLSI technology, however, the complexity of the interconnections is the most critical factor, while the complexity of the modules themselves is of lesser concern.

The work reported below adopts a point of view that is compatible with these VLSI requirements. The interconnection topology of the network is fixed and regular, and the interconnections are short and unidirectional. The Boolean functions of the constituent nodes are adjustable and not constrained to AND or OR functions. Refer to [1, 8] for a preliminary study of the functional characteristics of such networks. This class of systems includes threshold gate networks, for instance perceptrons, [7] since a threshold gate is indeed also an adjustable Boolean gate. The specifics of threshold gate logic, however, is not the subject of this paper. Rather, the treatment is at an implementation-independent level, using Boolean logic as the tool.

This paper considers the implementation of a completely specified Boolean function on a given network of polyfunctional nodes. Stated differently, it deals with the selection of the nodal functions in order to accomplish a given network function. (We assume that the network can indeed implement the given function.) This problem is called *decomposition* of a function onto the network. It is similar in purpose to programming a sequential computer, but here we view it as a decomposition of one Boolean function into a set of smaller Boolean functions.

The central topic in the decomposition problem is the *assignment of functional responsibility:* what parts of the given global function must be assigned to which nodes of the network? This problem addresses at the simplest possible level, namely that of Boolean operations, the most crucial question in the development of parallel systems. It can be viewed as a archetypical problem of parallel processing.

We apply the theory of decomposition of Boolean functions to this problem and show that in certain simple networks the solution to the decomposition problem can be obtained in a deterministic fashion. In more general networks, however, a search strategy is required to assign responsibility. We develop a heuristic criterion that reduces the amount of search necessary to find one or more correct decompositions.

Section 1 introduces the necessary definitions and a formal definition of the decomposition problem. Section 2 is a treatment of the preliminary concepts that will be used in the decomposition algorithm. The algorithm itself is developed in section 3, including a program that implements the method. Section 4 lists conclusions and future research questions.

1. DEFINITIONS

1.1. Polyfunctional Networks

Polyfunctional nets, or *P-nets*, consist of a number of polyfunctional combinational nodes. In this paper, the nodes of the P-nets are assumed to be all *identical* since it simplifies the description. However, the principles and methods outlined here are generalizable to heterogeneous networks.

Each node i has k inputs $\mathbf{x}_i=(x_{i,1},\ldots,x_{i,k})$ and its output z_i is a Boolean function of its inputs:

$$z_i = f_i(\mathbf{x}) = f_i(x_{i,1},\ldots,x_{i,k}) .$$

The nodal functions are adjustable independently, and input or output values do not influence these functions directly. In other words, the function f_i can be any one of a set of possible functions:

$$\phi = \{f_1,\ldots,f_p\} , \text{ with } |\phi|=p .$$

If $p=2^{2^k}$ then the node is *universal*: it can implement all possible functions of its inputs. A schematic representation of the node is shown in figure 1.

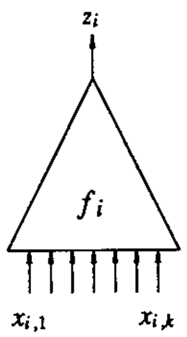


Figure 1: Schematic representation of a node (labeled i) of a polyfunctional network.

The node of a P-net is formally defined to be a pair:

$$\mathrm{N} \equiv (k,\phi) ,$$

where k is the *number of inputs* and ϕ is the *functional set.*

Each output of a node is connected to an input of another node, and the interconnections form a single-output loop-free network. The interconnections are defined by a set of *node interconnections:*

$$\mathsf{I} \equiv \{(i, z_i, \mathbf{x}_i) \mid i=1, \ldots, l\}$$

where:

i is a unique label for the node;

z_i is the output label of the node;

$\mathbf{x}_i=(x_{i,1}, \ldots, x_{i,k})$ are the input labels of the node, an ordered set of either output labels of other nodes or labels of the network inputs;

l is the number of nodes in the network.

Each node interconnection $(i, z_i, \mathbf{x}_i)$ specifies that the inputs of node i are connected to respectively $x_{i,1}, x_{i,2}, \ldots, x_{i,k}$, and that its output is labeled z_i. The latter can be used in other node interconnections. An example of a P-net is shown in figure 2.

The definition of an entire P-net is a 4-tuple:

$$\mathsf{P} \equiv [\mathsf{N}, z, \mathbf{x}, \mathsf{I}]$$

where:

N is the definition of the node;

z is the network output;

$\mathbf{x}=(x_1, \ldots, x_n)$ are the network inputs;

I is the set of node interconnections.

The resulting network implements a Boolean function of n inputs:

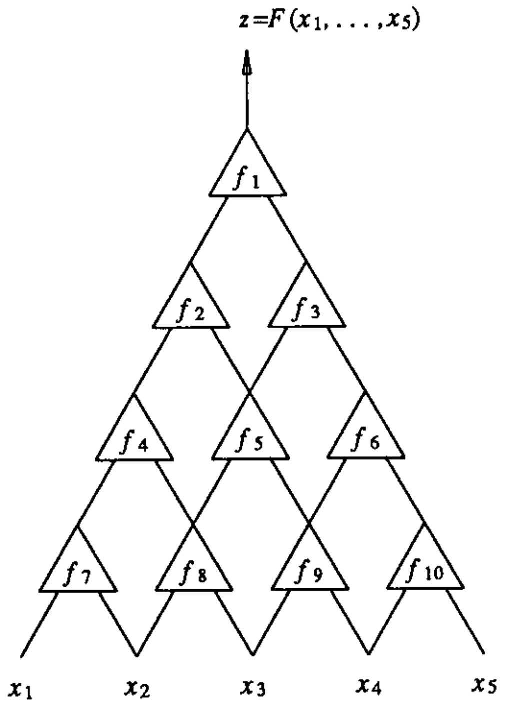


Figure 2: Example of a P-net. It consists of 10 nodes with 2 inputs each. The network has 5 inputs.

$$z = F(\mathbf{x}) = F(x_1, \ldots, x_n)$$

This network function F is derived from the choices f_i for the functions of the constituent nodes. By changing these functions the network implements a set of different functions:

$$\Phi = \{F_1, \ldots, F_q\}, \text{ where } |\phi| = Q .$$

Again, if $Q = 2^{2^n}$ then the network is *universal.*

Φ is not part of the definition of a P-net, but can be derived from it. For example, in figure 2 the network function is derived as follows:

$$F(x_1,x_2,x_3) = f_1(f_2(f_4(x_1,x_2),f_5(x_2,x_3)),f_3(f_5(x_2,x_3),f_6(x_3,x_1))) .$$

An *assignment* of node functions to a P-net is a mapping

$$\alpha : \{1, \ldots, l\} \rightarrow \phi : i \rightarrow f_i$$

meaning that the node labeled i is *assigned* the function $f_i \in \phi$. The corresponding network function is represented by F_α. A *partial assignment* (one in which not all elements of the domain are mapped) is a subset of a *complete* assignment.

An important question associated with P-nets relates to the size of the set of network functions Φ. We define the *universality index* γ as follows:

$$\gamma = \frac{Q}{2^{2^n}} .$$

Since $0 < Q \leq 2^{2^n}$ it follows that $0 < \gamma \leq 1$.

Another important concept is the total number of nodal assignments. If the network contains l nodes, then $P = p^l$ different combinations of node functions are possible. Typically this number will be larger than Q, meaning that not every different combination of node functions results in a different network function, and vice versa, a given network function can usually be obtained by different combinations of nodal functions. This phenomenon is called *functional redundancy*. If $P > Q$ then the P-net is said to be *redundant*; if $P = Q$ then it is *nonredundant*. We define the *redundancy factor* ρ as follows:

$$\rho = \frac{P}{Q} .$$

Q cannot exceed P and therefore $\rho \geq 1$. In P-nets, redundancy is the rule rather than the exception.

EXAMPLE 1: Consider the network of of figure 3. The global network function

$$F(x_1,x_2,x_3) = x_1 x_2 \bar{x}_3 + \bar{x}_1 x_2 x_3 .$$

can be achieved many different ways, for example

$f_1=z_2+z_3;\ f_2=z_4z_5;\ f_3=\bar{z}_5z_6;\ f_4=x_1x_2;\ f_5=x_2\oplus x_3;\ f_6=x_3\bar{x}_1$.

$f_1=z_2\bar{z}_3;\ f_2=z_4\oplus z_5;\ f_3=z_6;\ f_4=x_1x_2;\ f_5=x_2x_3;\ f_6=x_3x_1$.

$f_1=z_2+z_3;\ f_2=z_4\oplus z_5;\ f_3=z_5\bar{z}_6;\ f_4=x_1x_2;\ f_5=x_2x_3;\ f_6=x_3x_1$.

$f_1=z_3;\ f_2=0;\ f_3=z_5z_6;\ f_4=0;\ f_5=x_2;\ f_6=x_3\oplus x_1$.

etc.

If each of the structure's nodes can implement all 16 possible Boolean functions of 2 inputs, then $p=16$ and $P=16^6$. For this network $Q=256$ and the redundancy factor is

$$\rho=\frac{P}{Q}=\frac{16^6}{2^{2^3}}=\frac{2^{24}}{2^8}=2^{16}\ .\ \square$$

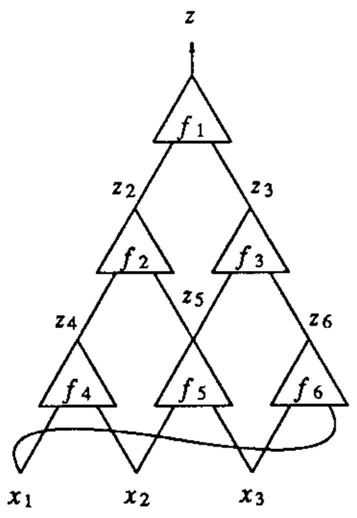


Figure 3: A 3-input polyfunctional network.

Two different types of redundancy can be considered.

a. Trivial redundancy.

Consider an interconnection of the output of one node to the input of another node (figure 4). The following operation will not change the function of the network: negate the output of the first node (that is, change f_1 to $\bar{f}_1$) and invert the corresponding input of the second node (change $f_2(\cdots, x_i, \cdots)$ to $f_2(\cdots, \bar{x}_i, \cdots)$). These modifications cancel each other. Such operations are possible only if the set ϕ is closed under negation of the output or inputs.

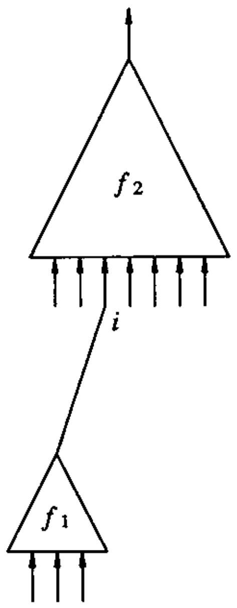


Figure 4: Example of trivial redundancy in a polyfunctional network.

b. Nontrivial redundancy.

A more important form of redundancy is shown in figure 5. This figure shows how one function, namely $F(x_1, x_2, x_3) = x_1 x_2 \bar{x}_3 + \bar{x}_1 x_2 x_3$ can be implemented in two entirely different ways. One implementation cannot be derived from the other by a trivial operation such as negating inputs and/or outputs. The reason for this redundancy is the fan-out that occurs at the inputs and inside the network.

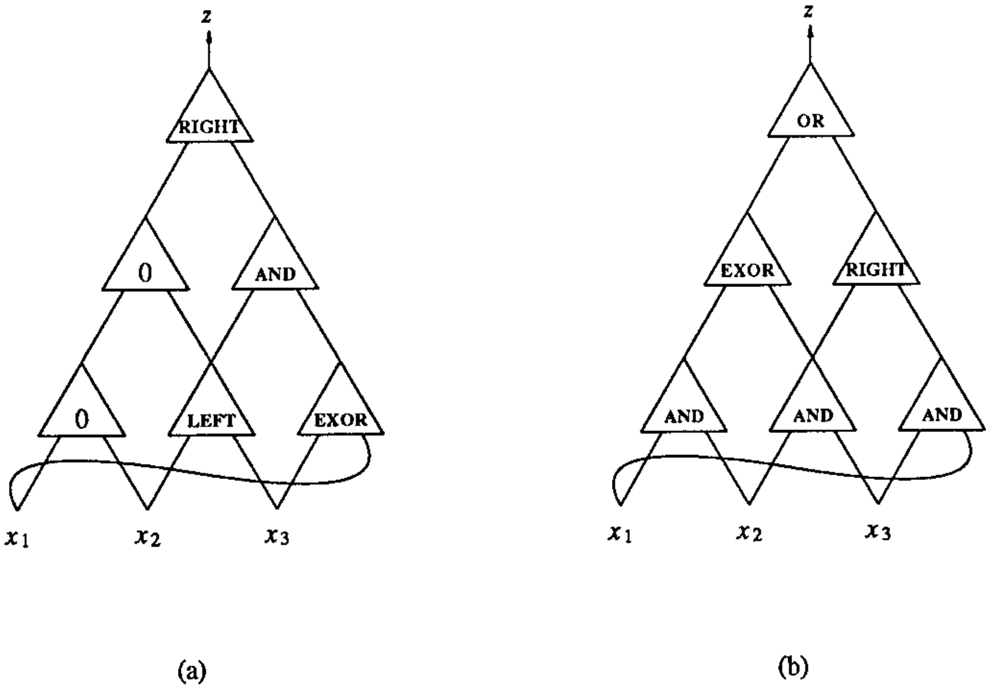


Figure 5: Example of non-trivial redundancy in a polyfunctional network.

1.2. Problem Description

Assume that a P-net definition P and a Boolean function G are specified. The problem is to implement the given function with the given network by finding an appropriate assignment for all the node functions. Stated differently, the problem is to decompose a Boolean function of n inputs into a set of smaller functions of k inputs each ($k<n$) in such a way that it matches the given network structure. More formally, a *decomposition problem* is defined as follows:

Given: a specific P-net definition $\mathsf{P} \equiv [\mathsf{N}, z, \mathbf{x}, \mathsf{l}]$ and a Boolean function $G(\mathbf{x}) = G(x_1, \ldots, x_n)$,

Find: an assignment α such that $F_\alpha(\mathbf{x}) \equiv G(\mathbf{x})$.

As a secondary problem, it may be required to find different (or all) possible assignments $\{\alpha_j\}$ that implement the network function, if more than one exists.

2. PRELIMINARIES

In this section the principles behind the decomposition strategy, to be developed in the next section, are presented. The strategy consists of the repeated application of two basic operations, namely *selection* and *reduction*. The first subsection below gives an overview of the existing theory of decomposition of Boolean functions, which forms the basic tool for the selection step. The second subsection presents the concept of reduction, used to determine the function of the remainder of the network after one or more nodes have received a functional assignment.

2.1. Theory of Decomposition of Boolean Functions [2-5]

A Boolean function $G(x_1, \ldots, x_n)$ is *decomposable* if G can be realized as a composition of functions of fewer than n variables each. Let $X = \{x_1, \ldots, x_n\}$ represent the set of input variables and assume that $A_1, A_2, \ldots, A_k$ are subsets of X such that $\bigcup_{i=1}^{k} A_i = X$, then

$$G(X) = G'(f_1(A_1), f_2(A_2), \ldots, f_k(A_k))$$

is a decomposition of the function G. This decomposition is shown schematically in figure 6.

Decompositions of Boolean functions can be classified as either disjunctive or non-disjunctive. In a *disjunctive* decomposition the input variables to different functions f_i are disjoint, that is, $\forall i, j, i \neq j : A_i \cap A_j = \emptyset$. If, for some $i, j, i \neq j : A_i \cap A_j \neq \emptyset$, the

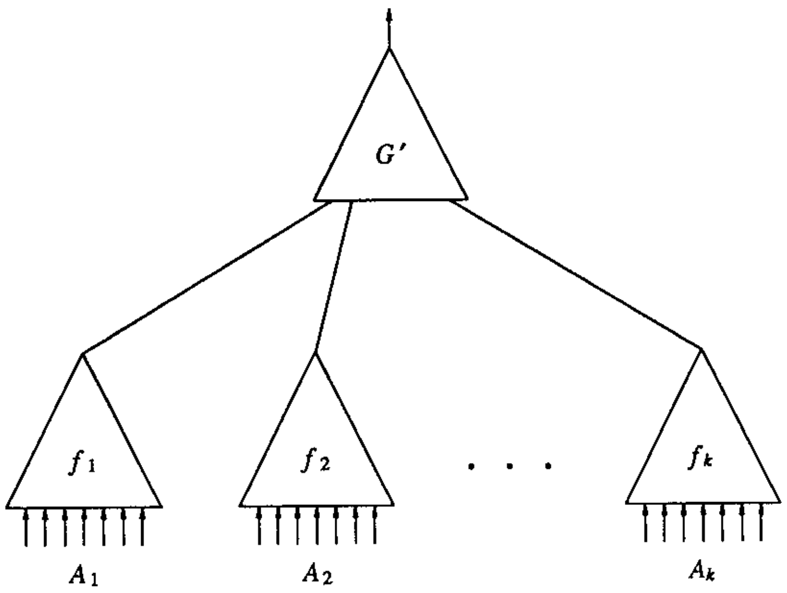


Figure 6: Decomposition of a Boolean function.

decomposition is called *nondisjunctive*.

2.1.1. Simple disjunctive decomposition

A simple disjunctive decomposition is a decomposition of the form $G(X)=G'(f_1(A_1),A_2)$, with $A_1 \cap A_2 = \varnothing$ and $A_1 \cup A_2 = X$. It can be characterized by arranging the truth table of G as in figure 7. The rows of the map represent entries with equal values for the variables in A_2, and the columns represent entries with equal values for the variables in the set A_1. The truth table arranged this way is called a *decomposition map* or *decomposition chart* with respect to the variable sets (A_1,A_2). If A_1 has u input variables ($|A_1|=u$) and A_2 has v input variables ($|A_2|=v$), and hence $|X|=n=u+v$, then each row of the decomposition map has 2^u entries and each column has 2^v entries.

THEOREM 1: [4] A completely specified Boolean function $G(X)$ has a simple

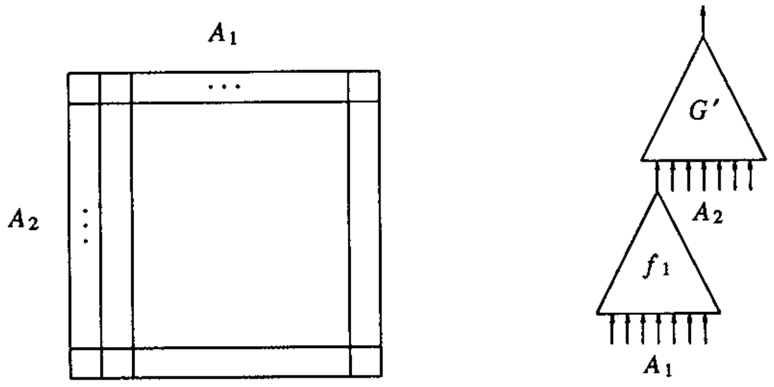


Figure 7: Decomposition chart for the variable sets (A_1,A_2).

disjunctive decomposition, $G(X)=G'(f_1(A_1),A_2)$, if and only if its decomposition chart with respect to the variable sets (A_1,A_2) has at most two distinct columns (columns with different patterns of 0's and 1's). □

EXAMPLE 2: A simple disjunctive decomposition of the function $G(x_1,x_2,x_3)= x_1 \oplus x_2 \oplus x_3$ with respect to $A_1=\{x_1,x_2\}$ and $A_2=\{x_3\}$ is: $G(x_1,x_2,x_3) = G'(f_1(x_1,x_2),x_3)$ with $z_1=f_1(x_1,x_2)=x_1 \oplus x_2$ and $G'(z_1,x_3)=z_1 \oplus x_3$. Its decomposition chart, shown in figure 8, has two distinct columns, namely "01" and "10." □

EXAMPLE 3: In figure 9 $G(x_1,x_2,x_3,x_4,x_5)$ is decomposed with respect to $A_1=\{x_1,x_2,x_3\}$ and $A_2=\{x_4,x_5\}$. In this example

$$G(x_1,x_2,x_3,x_4,x_5) = x_4(\bar{x}_1x_2+\bar{x}_2x_3)+x_5(x_1x_2+\bar{x}_2\bar{x}_3)\,,$$

$$z_1=f_1(x_1,x_2,x_3)=\bar{x}_1x_2+\bar{x}_2x_3\,,$$

$$G'(z_1,x_4,x_5)=x_4z_1+x_5\bar{z}_1\,.$$

Again, the decomposition chart of G has only 2 distinct columns, namely "0110" and "0011." □

If all the columns of a decomposition chart are identical then the function is

		$A_1=\{x_1,x_2\}$			
		00	01	11	10
$A_2=\{x_3\}$	0	0	1	0	1
	1	1	0	1	0

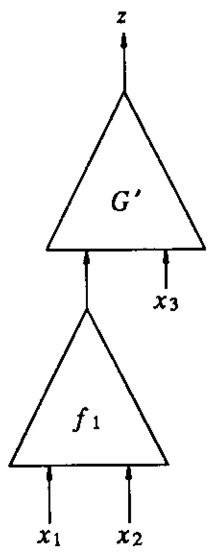


Figure 8: Example of a simple disjunctive decomposition of a Boolean function with 3 variables.

independent of the variables in the set A_1. In this case $G(X)=G'(A_2)$ and any choice for the function f_1 is valid. If the decomposition chart has two distinct columns, but the column patterns consist of all 0's or all 1's, then the function is independent of the variables of the set A_2, $G(X)=G'(f_1(A_1))$, and f_1 is determined by the column differences of the decomposition chart. In a typical case the decomposition map will have 2 non-trivial columns.

The decomposition map for an incompletely specified function has *don't care* entries. Two columns (rows) of a decomposition map are *compatible* if the *don't care* entries can be assigned 0's or 1's such that the columns (rows) become identical. The decomposition of an incompletely specified Boolean function is generally not unique since different assignments to the *don't care* entries may result in different decompositions. If more than two columns of the decomposition map are mutually incompatible then a simple disjunctive decomposition with respect to the particular variable sets is not possible.

		$A_1=\{x_1,x_2,x_3\}$							
		000	001	011	010	110	111	101	100
$A_2=\{x_4x_5\}$	00	0	0	0	0	0	0	0	0
	01	1	0	0	0	1	1	0	1
	11	1	1	1	1	1	1	1	1
	10	0	1	1	1	0	0	1	0

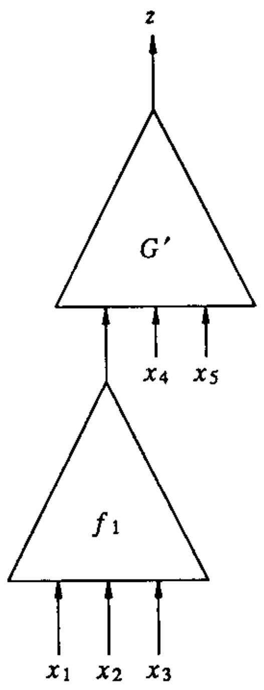


Figure 9: Example of a simple disjunctive decomposition of a Boolean function with 5 variables.

A classical problem in logic circuit design is to find the sets A_1 and A_2 such that a simple disjunctive decomposition is possible. In the decomposition problem, however, the network, and therefore the sets A_1 and A_2, are *given* and one needs to find the functions G' and f_1, if they exist.

2.1.2. Simple nondisjunctive decomposition

In a *nondisjunctive* decomposition the input variables of different functions f_i have some common elements. Hence, if

$$G(X) = G'(f_1(A_1), \ldots, f_k(A_k))$$

with $\bigcup_{i=1}^{k} A_i = X$ and $\exists i, j,\ i \neq j$ such that $A_i \cap A_j \neq \emptyset$, then the expression represents a nondisjunctive decomposition. A Boolean function $G(X)$ has a *simple* nondisjunctive decomposition if $G(X)=G'(f_1(A_1),A_2)$ with $A_1 \cup A_2=X$ and $A_1 \cap A_2 \neq \emptyset$. Let $A_1 \cap A_2=A_{12}$, $|A_{12}|=v$, $B_1=A_1-A_{12}$, $|B_1|=u_1$, $B_2=A_2-A_{12}$, $|B_2|=u_2$, and hence $u_1+u_2+v=n=|X|$. For each of the 2^v values for the variables in A_{12} we define a decomposition map with 2^{u_2} rows (corresponding to B_2) and 2^{u_1} columns (corresponding to B_1).

THEOREM 2: [4] A completely specified Boolean function $G(X)$ has a simple nondisjunctive decomposition if and only if each of these maps has not more 2 distinct columns. □

EXAMPLE 4: In figure 10, $B_1=\{x_1,x_2\}$, $B_2=\{x_4,x_5\}$, and $A_{12}=\{x_3\}$. The functions of the two nodes are:

$$f_1(x_1,x_2,x_3)=\bar{x}_3(x_1x_2)+x_3(x_1+x_2)\ ;$$

$$G'(z_1,x_3,x_4,x_5)=\bar{x}_3[z_1 \otimes (x_4+x_5)]+x_3[z_1+(x_4x_5)]\ .\ \square$$

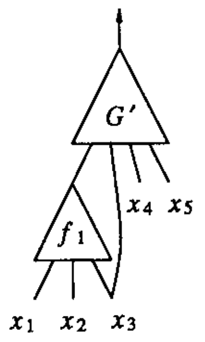
In this paper we treat the nondisjunctive case in an alternative way. Every nondisjunctive decomposition is converted into a disjunctive decomposition by introducing for each input $x_i \in A_{12}$ an additional *virtual* variable x_i^v. This virtual variable x_i^v must always be equal to x_i, and therefore the truth table entries with $x_i^v \neq x_i$ are *don't care*.

EXAMPLE 5: In figure 11, $B_1=\{x_1\}$, $A_{12}=\{x_2\}$, $B_2=\{x_3\}$. The introduction of x_2^v makes the network disjunctive with $A_1=\{x_1,x_2\}$ and $A_2=\{x_2^v,x_3\}$. However, since $x_2^v \equiv x_2$ half of the new decomposition chart contains *don't care* entries. □

Each virtual variable introduces *don't cares* equal in number to the size to the original truth table. The decomposition chart that results is called the *virtual* decomposition chart. The original one is called the *real* decomposition chart.

The theorem below follows directly from theorems 1 and 2; it is given here without proof.

THEOREM 3: A Boolean function $G(X)$ has a simple nondisjunctive decomposition if



$B_1=\{x_1,x_2\}$

$$B_2=\{x_4,x_5\}\quad \begin{array}{c|c|c|c|c|}
 & 00 & 01 & 11 & 10 \\ \hline
00 & 1 & 1 & 0 & 1 \\ \hline
01 & 0 & 0 & 1 & 0 \\ \hline
11 & 0 & 0 & 1 & 0 \\ \hline
10 & 0 & 0 & 1 & 0 \\ \hline
\end{array}$$

$x_3=0$

$B_1=\{x_1,x_2\}$

$$B_2=\{x_4,x_5\}\quad \begin{array}{c|c|c|c|c|}
 & 00 & 01 & 11 & 10 \\ \hline
00 & 0 & 1 & 1 & 1 \\ \hline
01 & 0 & 1 & 1 & 1 \\ \hline
11 & 1 & 1 & 1 & 1 \\ \hline
10 & 0 & 1 & 1 & 1 \\ \hline
\end{array}$$

$x_3=1$

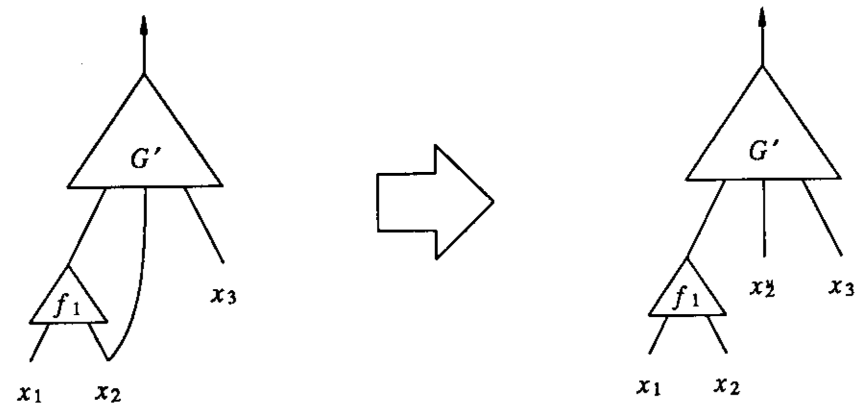
Figure 10: Example of a simple nondisjunctive decomposition of a Boolean function with 5 variables.

and only if its virtual decomposition chart has not more than two mutually incompatible columns. □

2.2. Reduction

The process called *reduction* answers the following question (figure 12). Suppose a polyfunctional network implements the function G and suppose that one of its nodes in the bottom layer, labeled 1, implements the function f_1. What is the *residual* function G' of the remainder of the network?

In figure 12, G is a function of (x_1,x_2,x_3,x_4) and G' is a function of (z_1,x_2,x_3,x_4). If G' were known then G could be deduced by substituting $f_1(x_1,x_2)$ for z_1 in G':



$$x_3\ \begin{array}{c|cccc} & \multicolumn{4}{c}{x_1x_2} \\ & 00 & 01 & 11 & 10 \\ \hline 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \end{array} \quad \Rightarrow \quad x_2'x_3\ \begin{array}{c|cccc} & \multicolumn{4}{c}{x_1x_2} \\ & 00 & 01 & 11 & 10 \\ \hline 00 & 0 & * & * & 1 \\ 01 & 0 & * & * & 1 \\ 11 & * & 1 & 1 & * \\ 10 & * & 1 & 0 & * \end{array}$$

Figure 11: Example showing how a nondisjunctive decomposition can be made disjunctive by introducing virtual variables.

$$G(x_1,x_2,x_3,x_4) = G'(f_1(x_1,x_2),x_2,x_3,x_4)\ .$$

In the decomposition problem, instead of G' and f_1 being given, G and f_1 are given and G' must be deduced.

EXAMPLE 6: Suppose

$$G(x_1,x_2,x_3,x_4) = (x_1+x_2) \oplus (x_3x_4)$$

and $f_1(x_1,x_2)=x_1+x_2$, then

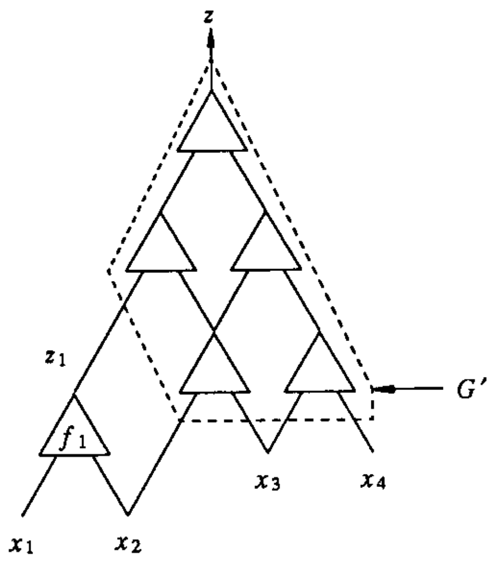


Figure 12: Schematic representation of reduction in a polyfunctional network.

$$G'(z_1,x_2,x_3,x_4) = z_1 \oplus (x_3x_4)\ .$$

If $f_1(x_1,x_2)=\bar{x}_1\bar{x}_2$, then

$$G'(z_1,x_2,x_3,x_4) = \bar{z}_1 \oplus (x_3x_4)\ .$$

But what if for example $f_1(x_1,x_2) = x_1x_2$? In this case reduction is *impossible* for the following reason. If the network input is **x**=(0,0,0,0) then the network output should be z=G(0,0,0,0)=0; the output of node 1 for this input will be z_1=f_1(0,0)=0. If the network input is **x**=(1,0,0,0) then the network output must be z=G(1,0,0,0)=1; z_1=f_1(1,0) is still equal to 0. What should be the truth value of G' for (z_1,x_2,x_3,x_4) = (0,0,0,0)? The two requirements imposed by G are conflicting. Hence, the assignment $f_1(x_1,x_2)$=x_1x_2 can never be correct for this function. □

Another important feature of reduction is that, besides sometimes being impossible, it also sometimes generates *don't care* entries in the truth table of the residual network function.

Example 7: Consider the network of figure 13 with $G(x_1,x_2,x_3) = (\bar{x}_1 x_2) x_2 x_3 = (\bar{x}_1 x_2) x_3$. Assume node 1 implements $f_1(x_1,x_2) = \bar{x}_1 x_2$. Is in this example $G'(z_1,x_2,x_3) = z_1 x_2 x_3$ or $G'(z_1,x_2,x_3) = z_1 x_3$? The answer is: neither of them. Since z_1 cannot be 1 if x_2 is 0, nothing is specified regarding G' when $(z_1,x_2)=(0,0)$, and hence $G'(1,0,0)$ and $G'(1,0,1)$ are *don't cares*.

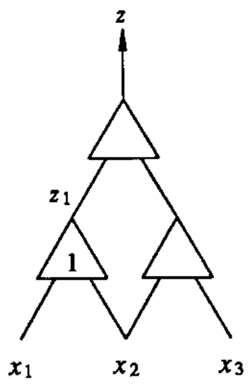
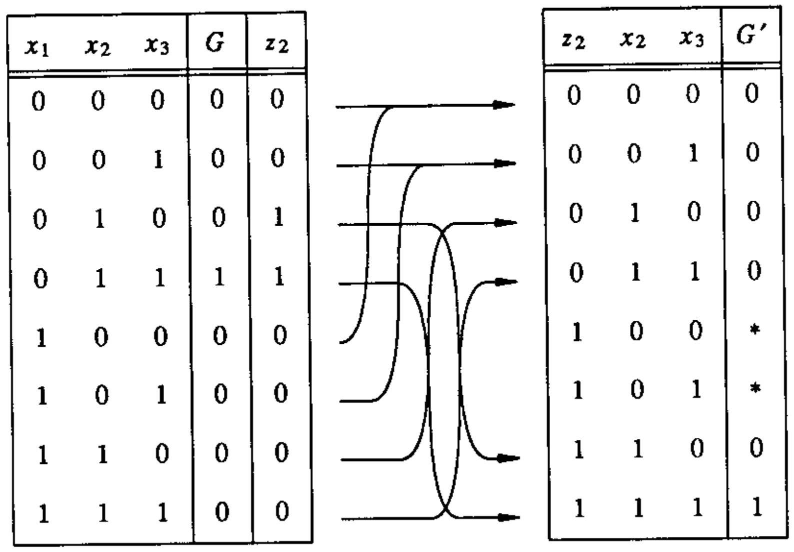


Figure 13: Example of reduction in a 3-input polyfunctional network.

Figure 14 shows how the reduction proceeds in this example. It is represented as a mapping from the truth table of G to the truth table of G'. A number of truth table entries of G are mapped into the same entry in G'. For instance $G(0,0,0)$ and $G(1,0,0)$ are both mapped into $G'(0,0,0)$. If these entries of G were different then a conflict would occur and the reduction would not be possible. A number of truth table entries of G' do not receive a value, for instance $G'(1,0,0)$ and $G'(1,0,1)$. These entries are *don't cares* (represented by ''*'' in the figure). □

The algorithm for the reduction process, expressed in a Pascal-like pseudo-language, is as follows:



x_1	x_2	x_3	G	z_2
0	0	0	0	0
0	0	1	0	0
0	1	0	0	1
0	1	1	1	1
1	0	0	0	0
1	0	1	0	0
1	1	0	0	0
1	1	1	0	0

z_2	x_2	x_3	G'
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	*
1	0	1	*
1	1	0	0
1	1	1	1

Figure 14: Reduction as a mapping from one truth table to another.

```
function reduce($G$, $f$, conflict);
begin initialize $G'$ to all don't-care;
      conflict ← false;
      for all $s \in \{0,1\}^n$
      do begin z ← $f(s_1)$;
               if $G'(z, s_2)$ = don't-care
               then $G'(z, s_2)$ ← $G(s)$
               else if $G'(z, s_2) \not\sim G(s)$
                    then conflict ← true
         end;
      return($G'$)
end;
```

s is an input vector for the network; s_1 is the part of it received by the current node and s_2 is the part fed to the remainder of the network. "$\not\sim$" denotes *incompatibility*:

$$a \not\sim b \Leftrightarrow a \neq don't\text{-}care \text{ and } b \neq don't\text{-}care \text{ and } a \neq b \ .$$

The opposite is *compatibility*:

$$a \sim b \Leftrightarrow a = don't\text{-}care \text{ or } b = don't\text{-}care \text{ or } a = b \ .$$

The procedure `reduce` can also be used if G itself contains *don't cares*. A *don't care* entry in G may be mapped into any entry of G' without causing a conflict.

3. DECOMPOSITION ALGORITHM

This section develops a procedure for the decomposition problem. Nothing is assumed about the structure of the given network. Rather than designing a specific special-purpose algorithm that applies only to a restricted set of networks, we develop a general-purpose strategy that applies to any P-net. The algorithm is a search strategy consisting of the repeated application of a selection and reduction process. In the case of a disjunctive binary tree the search degenerates to a deterministic scheme.

The skeleton of the search strategy is presented first (section 3.1), the case of binary trees is discussed next (section 3.2), and the same strategy is then applied to general networks (section 3.3).

3.1. Principles of the Search Algorithm

A brute-force solution to the decomposition problem might consist of exhaustively generating all possible combinations of nodal assignments until a combination is found that produces the desired network function. If redundant assignments are sought then the search continues until more solutions are found. In other words, the strategy consists of exhaustively generating all possible assignments α_j and for each assignment verifying whether $F_{\alpha_j} \equiv G$.

Denote N_G the number of ways G can be implemented on the given network:

$$N_G = | \{ \alpha | F_\alpha \equiv G \} | .$$

The probability that a randomly selected assignment α achieves the function G is:

$$\text{Prob}[F_\alpha \equiv G] = \frac{N_G}{P} .$$

where $P = p^l$ is the total number of possible assignments for the P-net. Denote A the number of assignments examined before the correct one is found. Bounds for A are: $A_{\min}=1$ and $A_{\max}=P-N_G+1$. If the assignments are tried in a random order then A obeys a negative hypergeometric distribution with parameters N_G, $P-N_G$, and 1. [6] The mean of such a distribution is

$$A_{av} = \frac{P+1}{N_G+1} .$$

The decomposition strategy discussed here generates a limited number of assignments (less than P) in an order such that A_{av} is as small as possible. The algorithm tries only assignments selected by a *selection criterion*. Its strength is measured by how well it succeeds in reducing A_{av}.

The selection criterion is global: which assignments it selects depends on the network function and the entire structure of the network. In our approach, however, the global assignment is decomposed into a sequence of nodal assignments and the selection criterion is also local to each node. The nodes are treated in a sequential bottom-up fashion. At each node a few candidate functions are selected and tried one after the other. The decomposition algorithm then becomes a search through a tree structure. Every leaf of the tree corresponds to a complete assignment and A is the number of leaf nodes that are examined.

In a search procedure the computational time is not directly related to the number of leaf nodes examined, but rather depends on the size of that part that is actually traversed. We call S the number of nodes of the tree that are traversed. Bounds for S are: $S_{\min}=l$, the number of nodes in the network (the depth of the search tree), and $S_{\max}=p^l=P$, the total number of assignments (the overall size of the tree). There is no direct relationship between S_{av} and A_{av}.

The search procedure has 2 major components. First, a local *selection procedure* reduces the number of functions to be considered at each node from $p=|\phi|$ to some smaller number $p'=|\phi'|$. It is based on the observation that a number of functions can be excluded from examination and that a few functions are *likely* to achieve the given network function. This component of the search procedure can be expressed as:

$$\phi' \leftarrow \mathrm{S}(\phi, G, i, \mathrm{P})$$

where S is the selection procedure, ϕ is the functional set of the node, G is the given network function, i is the label of the current node, and P is the network definition.

A second component of the search procedure is necessary because the function to be assigned to a node depends on previous assignments to other nodes. Therefore, the selection procedure S should be coordinated between different nodes. This coordination is accomplished by using a reduction computation between two nodes. The first node (in the bottom layer) selects an assignment based on the given network function. It then passes the residual network function to the next node, which makes a selection based on this residual function, and so on. The coordination between nodes is therefore achieved by using successively modified network functions.

EXAMPLE 8: Figure 15 shows a simple P-net. Suppose the given network function is $G(x_1,x_2,x_3) = (x_1 \oplus x_2)(x_2+x_3)$. Treating the nodes in the order (2,3,1), the decomposition might proceed as follows:

Node 2: $G(x_1,x_2,x_3) = (x_1 \oplus x_2)(x_2+x_3)$,
$f_2(x_1,x_2) = x_1 \oplus x_2$,
$G'(z_2,x_2,x_3) = z_2(x_2+x_3)$.

Node 3: $G(z_2,x_2,x_3) = z_2(x_2+x_3)$,
$f_3(x_2,x_3) = x_2+x_3$,
$G'(z_2,z_3) = z_2z_3$.

Node 1: $G(z_2,z_3) = z_2z_3$,
$f_1(z_2,z_3) = z_2z_3$,
$G'(z) = z$.

This example demonstrates redundancy. Indeed, since $(x_1 \oplus x_2)(x_2+x_3) = \bar{x}_1x_2x_3$, an alternative solution might be:

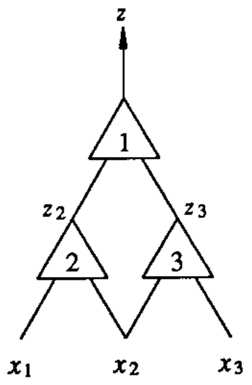


Figure 15: Example of a decomposition problem.

Node 2: $G(x_1,x_2,x_3) = \bar{x}_1 x_2 x_3$,

$f_2(x_1,x_2) = \bar{x}_1 x_2$,

$G'(z_2,x_2,x_3) = z_2 x_3$.

Node 3: $G(z_2,x_2,x_3) = z_2 x_3$,

$f_3(x_2,x_3) = x_3$,

$G'(z_2,z_3) = z_2 z_3$.

Node 1: $G(z_2,z_3) = z_2 z_3$,

$f_1(z_2,z_3) = z_2 z_3$,

$G'(z) = z$.

Alternative assignments, obtained by equivalence or inversion operations, are demonstrated below:

Node 2: $G(x_1,x_2,x_3) = \bar{x}_1 x_2 x_3$,

$f_2(x_1,x_2) = x_1 + \bar{x}_2$,

$G'(z_2,x_2,x_3) = \bar{z}_2 x_3$.

Node 3: $G(z_2,x_2,x_3) = \bar{z}_2 x_3$,

$f_3(x_2,x_3) = \bar{x}_3$,

$G'(z_2,z_3) = \bar{z}_2 \bar{z}_3$.

Node 1: $G(z_2,z_3) = \bar{z}_2 \bar{z}_3$,

$f_1(z_2,z_3) = \bar{z}_2 \bar{z}_3$,

$G'(z) = z$. □

3.2. Binary Tree Networks

We now apply the principles discussed in the previous sections to binary tree networks. Extensions to ternary or higher-order trees are trivial.

A *disjunctive* binary tree network is a network where the 2 input values of a node depend on a disjoint set of input variables, that is, each input variable is connected to only one node. In a *nondisjunctive* binary tree network the 2 inputs to a node may depend on an overlapping set of input variables. A nondisjunctive tree can be reduced to a disjunctive one by the introduction of virtual variables. Figure 16 shows a disjunctive binary tree with 8 inputs.

The number of virtual inputs for the network is $m=2^i$, where i is the number of layers in the tree. For a disjunctive tree $m=n$; for a nondisjunctive tree $m>n$. The number of nodes in the network is $l=m-1$. A disjunctive tree is not universal ($Q<2^{2^n}$; $\gamma<1$) and it exhibits only trivial redundancy. By contrast, a nondisjunctive tree can be universal provided there are enough fan-out connections in the inputs and the redundancy will typically be nontrivial.

The algorithm for decomposing a given Boolean function onto a binary tree works as follows. Starting with any node in the bottom layer, for instance node 4 in figure 16, a function is assigned using the decomposition chart with respect to $A_1=\{x_1,x_2\}$ and $A_2=\{x_3, \ldots, x_m\}$. (In a disjunctive tree, only a single function and its negation are possible. In a nondisjunctive tree, the assignment is not unique as a result of the *don't care*

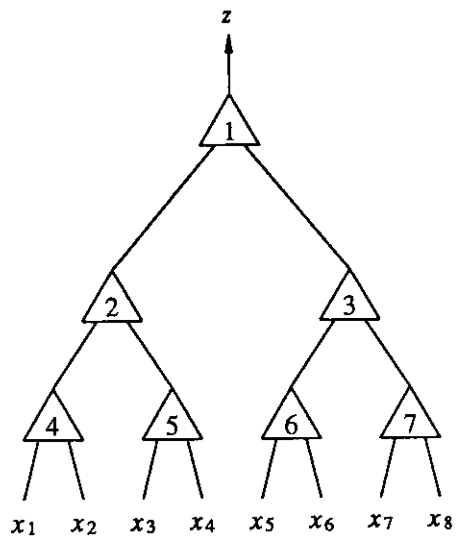


Figure 16: Example of a binary tree network with 8 inputs. It has 7 nodes and 3 layers.

entries in the virtual decomposition chart, and a search is necessary.) Next, the residual function is determined by reduction. (It may generate *don't care* entries unless the tree is disjunctive.) The procedure treats the remaining nodes in a similar fashion, taking them one by one in a bottom-up order, that is, child nodes are assigned functions before their parent node. For the example of figure 16 two possible traversals of the nodes are (4,5,6,7,2,3,1) and (4,5,2,6,7,3,1).

To conclude, disjunctive binary trees have a very simple deterministic decomposition strategy; no search is necessary and $S_{av}=S_{\min}=S_{\max}=l$. For a nondisjunctive binary tree, a search is typically required and $S_{av}, S_{\max}>l$.

EXAMPLE 9: Figure 17 illustrates a decomposition in a nondisjunctive binary tree. The function for the network is $G(x_1,x_2,x_3)=x_1x_2+x_2x_3$. The figure shows the real and virtual decomposition charts. A possible assignment for node 2 would be $f_2(x_1,x_2)=\bar{x}_1x_2$, resulting in a residual function G' as indicated. However, no assignment is possible for node 3 since its decomposition chart has 3 mutually incompatible columns, and backtracking is required. The assignment $f_2(x_1,x_2)=x_1x_2$ leads to a different residual function and $f_3(x_2^v,x_3)=x_2^v x_3$ is a possible assignment for node 3. This leads to a correct global assignment. □

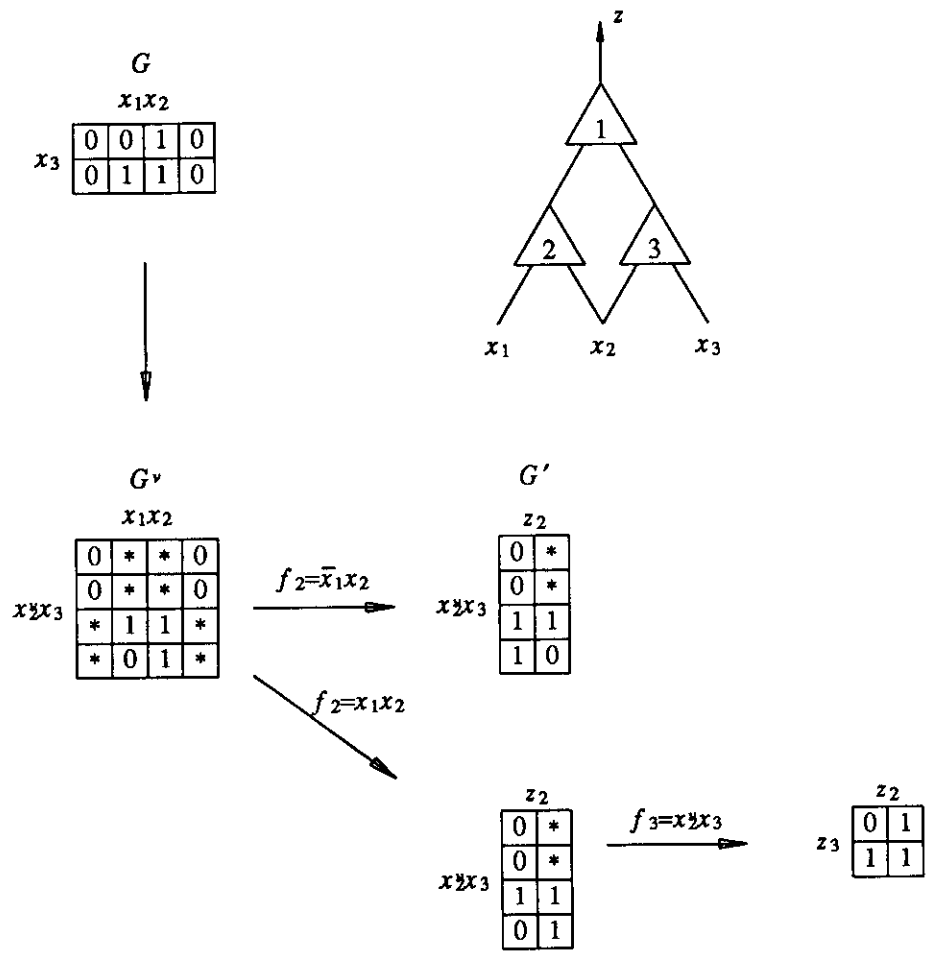


Figure 17: Example showing the need for search and backtracking in a non-disjunctive binary tree network.

3.3. General Polyfunctional Networks

The search algorithm is now extended to include all possible P-nets. Two possible local selection criteria are presented, resulting in two different algorithms. The *cautious* algorithm finds all possible (redundant) solutions, but it is by necessity very defocussed and may require a lot of search. The *adventurous* algorithm is more focussed, but it does not aim at finding all the solutions. These two selection criteria are presented first. A preliminary analysis of the complexity of the adventurous algorithm follows. The last subsection presents a program that implements the adventurous algorithm.

3.3.1. Cautious selection criterion

Consider a node in the bottom layer of a P-net and suppose it takes x_1 and x_2 as inputs. If neither x_1 nor x_2 are inputs to the remainder of the network then the decomposition is disjunctive. In this case only two functional assignments (negations of each other) are applicable for that node. If the network function contains *don't care* entries then multiple assignments may be possible and must be tried in sequence.

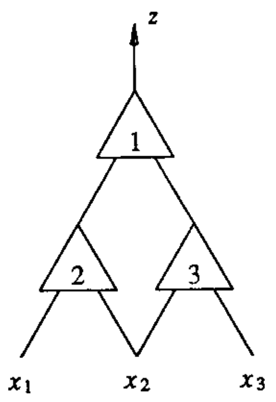
Now consider the case where x_2 is an input to the remainder of the network as well. The virtual decomposition chart must be used to determine the nodal assignment; mutually incompatible columns must have a different functional value. For node 2 in figure 18 only the columns $G_{\delta 1}$ and $G_{\gamma 1}$ are mutually incompatible and should be assigned different functional values. Hence 8 nodal assignments are possible, as shown in the figure.

For this particular network the virtual decomposition chart has at most 2 pairs of mutually incompatible columns, namely ($G_{\delta 0}$,$G_{\gamma 0}$) and ($G_{\delta 1}$,$G_{\gamma 1}$). If these 2 pairs are indeed incompatible then only 4 local assignments must be tried. If only one pair is incompatible then 8 assignments are possible. In the extreme all 16 possible functions of 2 inputs have to be examined.

If *both* x_1 and x_2 are inputs to the remainder of the network as well then (as in

$$G \qquad\qquad\qquad\qquad G^v$$

$$\begin{array}{c|cccc} & \multicolumn{4}{c}{x_1x_2} \\ \hline x_3 & 0 & 0 & 1 & 0 \\ & 0 & 1 & 1 & 0 \end{array} \longrightarrow \begin{array}{c|cccc} & \multicolumn{4}{c}{x_1x_2} \\ \hline & 0 & * & * & 0 \\ x_2^v x_3 & 0 & * & * & 0 \\ & * & 1 & 1 & * \\ & * & 0 & 1 & * \end{array}$$



POSSIBLE ASSIGNMENTS:

x_1	0	0	1	1
$\bar{x}_1 x_2$	0	1	0	0
$x_1 \oplus x_2$	0	1	0	1
$x_1 \otimes x_2$	1	0	1	0
$x_1+\bar{x}_2$	1	0	1	1
$\bar{x}_1$	1	1	0	0
$\bar{x}_1+\bar{x}_2$	1	1	0	1

Figure 18: Example of cautious selection with one virtual variable x_2^v.

figure 19) any pair of columns is never incompatible and therefore all 16 local functions have to be examined.

In general, assuming that a node has k inputs, v of which are inputs to the rest of the network as well ($0 \le v \le k$), and assuming that the network function is completely specified, then the number of functional options for that node is *at least* 2^{2^v}. The selection procedure reduces ϕ to ϕ' with the size of ϕ' bounded by

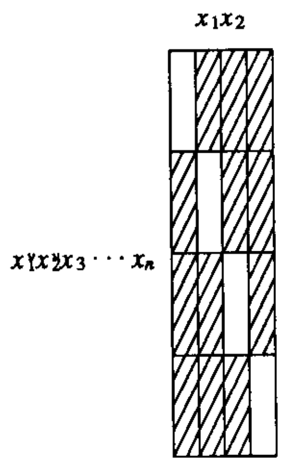


Figure 19: Virtual decomposition chart with two virtual variables x_1^v and x_2^v.

$$2^{2^v} \le p' = |\phi'| \le 2^{2^k} \quad .$$

If the given network function contains *don't cares* then the lower bound is larger. Hence, *don't cares* increase the amount of search needed to find a decomposition. Fan-out connections in the network introduce *don't care* entries in the network truth table, and they constitute a major source of search complexity in the decomposition problem.

The cautious selection procedure S_c, selecting functions that have different values for mutually incompatible columns of the virtual decomposition chart, is listed below:

```
function $S_c(\phi, G, i, P)$;
begin $G^v \leftarrow$ virtual($G$, i, P);
      $\phi' \leftarrow \emptyset$;
      for all $f \in \phi$
      do begin retain $\leftarrow$ true;
               for all $(s, s') \in (\{0,1\}^k)^2$
               do if $(G_s^v \not\sim G_{s'}^v)$ and $(f_s = f_{s'})$
                  then retain $\leftarrow$ false;
               if retain
               then $\phi' \leftarrow \phi' \cup f$
         end;
      return($\phi'$)
end;
```

virtual is a function that generates the virtual truth table; G_s^v is the s-th column of the virtual decomposition chart.

The program listed above implements the test

$$\forall\, s, s' \in \{0,1\}^k : G_s^v \not\sim G_{s'}^v \Rightarrow f_s \neq f_{s'} .$$

This test, however, is automatically included in the reduction process. Indeed, the reduction will return a conflict if and only if two mutually incompatible columns of the virtual decomposition chart receive the same functional value. In other words, a function selected by the cautious selection will not cause a conflict in the reduction step, and, vice-versa, a conflict in the reduction computation occurs only for a function not selected by the cautious criterion. Hence, S_c is redundant and one may as well use the reduction process to make the selection. Therefore, the procedure can be simplified:

```
function $S_c(\phi, G, i, P)$;
begin return($\phi$)
end;
```

In other words, the cautious criterion does not make any selection at all and feeds the reduction phase with *all possible* local functions. This program is therefore doing the maximum amount of search and relies only on the reduction step to cut down the number of assignments. In return, it generates all possible redundant assignments after successive backtrackings.

3.3.2. Adventurous selection criterion

We now discuss a possibility to further reduce the size of ϕ', based on the real decomposition chart. This decomposition chart generally contains more than two mutually incompatible groups of columns. How can local functions be selected based on this contradictory decomposition chart?

EXAMPLE 10: In figure 20 there are 3 mutually incompatible classes of columns in the real decomposition chart of node 2, namely $C_1=\{G_{00},G_{11}\}$, $C_2=\{G_{01}\}$, and $C_3=\{G_{10}\}$. We select three functions (plus their negations) by mapping these 3 mutually incompatible classes into 2 groups and assigning the same functional value to columns of the same group. The conflict that results from assigning two incompatible columns the same value is ignored for the time being. We assume it will be taken care of by the remainder of the network. Three groupings are possible: $\{C_1 \cup C_2, C_3\}$, $\{C_3 \cup C_1, C_2\}$, and $\{C_2 \cup C_3, C_1\}$. Each grouping results in a functional assignment, as shown in figure 20. □

$$G$$

$$\begin{array}{c|cccc} & \multicolumn{4}{c}{x_1x_2} \\ \hline x_3 & 0 & 0 & 1 & 0 \\ & 0 & 1 & 1 & 0 \end{array}$$

POSSIBLE ASSIGNMENTS:

x_1x_2	0 0 1 0
$\bar{x}_1x_2$	0 1 0 0
x_2	0 1 1 0
$\bar{x}_2$	1 0 0 1
$x_1+\bar{x}_2$	1 0 1 1
$\bar{x}_1+\bar{x}_2$	1 1 0 1

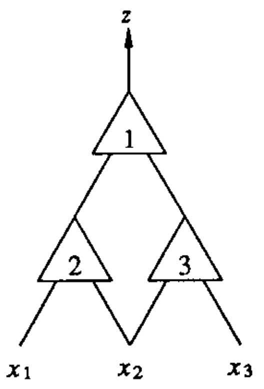


Figure 20: Example of adventurous selection.

This selection is based on the incompatibilities in the real decomposition chart only and does not need any knowledge of how many or which local variables are inputs to the remainder of the network as well.

If the network function G contains *don't care* entries then one column (containing *don't care* entries) could be compatible with two other columns that are mutually incompatible (see for example figure 21). Two alternative assignments are then possible, and both must be tried.

x_3 \ x_1x_2	00	01	11	10
0	1	1	1	0
1	*	0	1	0

Figure 21: Column G_{00} is compatible with both G_{01} and G_{11}, but G_{01} and G_{11} are mutually incompatible.

EXAMPLE 11: Figure 22 illustrates that the adventurous algorithm does not necessarily generate *all* solutions. With the network function $G(x_1,x_2,x_3)=x_1x_2$, the only functions tried for node 2 are $f_2(x_1,x_2)=x_1x_2$ and its negation. However, the choice $f_2(x_1,x_2)=x_1$ and $f_3(x_2,x_3)=x_2$, also a valid solution, is not found. □

The adventurous algorithm neglects some correct assignments but reduces the total size of its search tree. It is intuitively very appealing. Consider the example of figure 22: why should $f_2(x_1,x_2)=x_1$ or $f_2(x_1,x_2)=\bar{x}_1x_2$ be tried? None of them causes a reduction conflict and both might be part of a solution, but $f_2(x_1,x_2)=x_1x_2$ is definitely the most appropriate choice, the most likely to lead to a correct assignment. In general, different local assignments lead to different residual functions G' and it depends on the structure of the remainder of the network which G' is implementable.

The program for the adventurous algorithm is listed below:

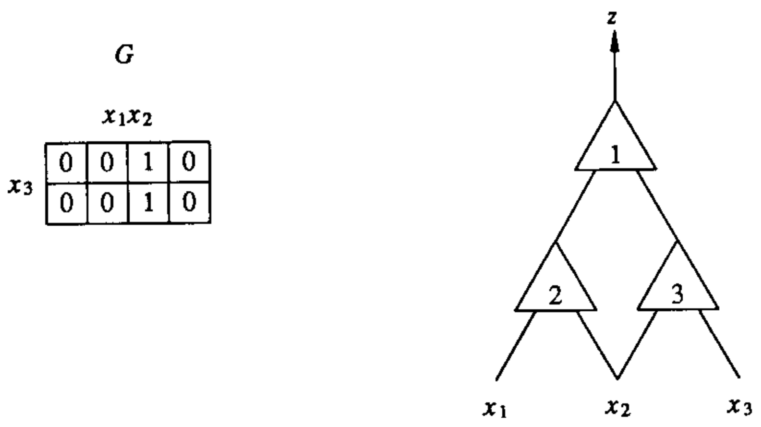


Figure 22: Example showing that the adventurous algorithm does not necessarily generate all solutions to a decomposition problem.

```
function $S_a(\phi, G, i, P)$;
begin $\phi' \leftarrow \emptyset$;
      for all $f \in \phi$
      do begin retain $\leftarrow$ true;
               for all $(s, s') \in (\{0,1\}^k)^2$
               do if $(G_s \sim G_{s'})$ and $(f_s \neq f_{s'})$
                  then retain $\leftarrow$ false;
               if retain
               then $\phi' \leftarrow \phi' \cup f$
         end;
      return($\phi'$)
end;
```

In summary, the adventurous selection criterion is the complement of the cautious criterion. Instead of assigning *different* functional values to *incompatible* columns of the *virtual* decomposition chart, it assigns *equal* functional values to *compatible* columns of the *real* decomposition chart. The relationship between both selection criteria is as follows. The cautious selection is based on the formal property (theorem 3)

$$\forall\, s, s' \in \{0,1\}^k : G_s^y \not\sim G_{s'}^{y} \Rightarrow f_s \neq f_{s'} \,. \qquad (1)$$

The adventurous selection uses:

$$\forall\, s,s' \in \{0,1\}^k : G_s \sim G_{s'} \Rightarrow f_s = f_{s'}\,. \tag{2}$$

The latter is *not* a theorem (we showed a counterexample above) but a heuristic. By contrast, (1) is a theorem that holds *for all* network functions and *for all* assignments implementing the given function. It remains to be proven that *for all* network functions *there exists* a corresponding assignment for which (2) holds. This question is left for future research.

3.3.3. Characteristics of the adventurous algorithm

The combined adventurous decomposition program is listed below:

```
procedure decomp($G$,$i$,$\alpha$,P);
begin if $i$=0
      then print assignment $\alpha$
      else begin $\phi'$ ← $S_a$($\phi$,$G$,i,P);
                 for all $f_j \in \phi'$
                 do begin $\alpha'$ ← $\alpha \cup \{i \to f_j\}$;
                          $G'$ ← reduce($G$,$f_j$,conflict);
                          if not conflict
                          then begin $i'$ ← next($i$,P);
                                     decomp($G'$,$i'$,$\alpha'$,P)
                               end
                    end
           end
 end;
```

G and G' are respectively the network function and the residual network function for node i, the latter to passed to the next node. The procedure `next` calculates the label of the next node in a bottom-up fashion; label "0" is returned when the top node is reached. If i=0 then the top node has been assigned a local function and, since apparently the corresponding reduction succeeded, the complete assignment implements either the given network function or its negation.

Some performance predictions of this algorithm include the following. The amount of memory required for each node is roughly the size of the network truth table. However, the network truth table decreases in size as the algorithm proceeds through the network. For example, the sizes of the network truth tables for two different node traversals of the network of figure 23 is shown in table 1. The first traversal is superior in terms of memory requirements.

node	7	8	9	10	4	5	6	2	3	1	total
size	32	32	32	32	16	16	16	8	8	4	196
node	7	8	4	9	5	2	10	6	3	1	total
size	32	32	32	32	32	32	32	16	8	4	252

Table 1: Size of the memory required at each node for different traversals of the nodes of the network shown in figure 23.

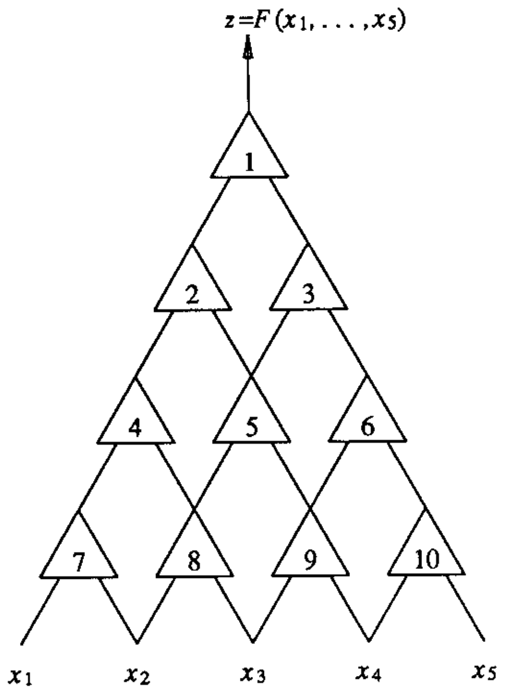


Figure 23: A simple 5-input polyfunctional network.

The amount of search necessary, S, depends not only on the particular network structure, but also on the given network function. The following analysis leads to an upper bound for $S_{\max}$.

Assume each node has k inputs and the network has n inputs. The number of different columns in the real decomposition chart is less than 2^k, the total number of columns in the decomposition chart. Additionally, the length of each column is 2^{n-k} entries and therefore at most $2^{2^{n-k}}$ columns are possible. If C represents the number of different columns in the decomposition chart then

$$C \leq \min\{2^k, 2^{2^{n-k}}\} = C_{\max} .$$

This upper bound is smallest for $k=2$ and for $k=n-1$ ($C_{\max}=4$) and is largest for $k=2^{n-k}$, that is, $\log_2(k)+k=n$. If C different columns exist then the adventurous criterion will select 2^C local functions, which is bounded by $2^{C_{\max}}$. The size of the network function changes from node to node, and so does $C_{\max}$. A pessimistic upper bound for $S_{\max}$ would be to assume $C_{\max}$ constant for all the nodes: $S_{\max}<2^{l \times C_{\max}}$. A tighter upper bound for $S_{\max}$ would have to take into account the size of the network function at each node (as shown in table 1) and the number of assignments excluded by the reduction conflict detection. Furthermore, this would still only give an upper bound, namely the number of steps if the resulting tree is traversed entirely.

Simulation showed for the 3-input P-network of figure 3: $S_{\max}=99$ and $S_{av}=26$. This is certainly an enormous improvement over the upper limit $2^{l \times C_{\max}}=2^{24}$. Further experimental results, using the program described in the next section, are given in table 2. Different numbers in the column S are for different network functions G.

3.3.4. Implementation

The adventurous algorithm decomp has been implemented in a Pascal program. It uses a more efficient formulation of the adventurous selection process. Instead of *testing* all members of ϕ with the criterion, it *generates* the set ψ of all functions that satisfy the criterion. It then checks each one of these functions for membership in ϕ. In other words, S_a is implemented as:

n	k	l	S	$2^{l \times C_{\max}}$
3	2	6	11, 55, 58	2^{24}
4	3	4	95, 246, 288	2^{16}
4	2	12	158, 613, 775	2^{48}
4	2	24	61, 178, 205	2^{96}
4	2	28	236, 592, 695	2^{112}
7	2	18	39	2^{72}

Table 2: Experimental data for the adventurous decomposition program.

```
function $S_a$($\phi$,$G$,i,P);
begin $\psi \leftarrow$ generate-all($G$,i,P);
      $\phi' \leftarrow \psi \cap \phi$;
      return($\phi'$)
end;
```

If ϕ is universal then $\phi' \equiv \psi$. This approach is more efficient in time and in storage requirements than the version presented earlier.

The program was also written to be a little more general than described above. It is able to invoke itself recursively on a subnetwork. This allows the user to define a P-net as a network of basic nodes and then use this network as a *macro*-node or module in the definition of a larger network, and so on (see for example figure 24). The program decomposes the network function and assigns functions to each subnetwork assuming they are basic nodes of the network. It then further subdecomposes the assigned functions for these smaller networks, and so on. The decomposition is organized in a depth-first manner, that is, the subdecompositions are treated before the next macro-node is considered. The decomposition backtracks if the current subnetwork cannot implement its assigned function.

As a result of this multilevel strategy, backtracking can be done in a more radical fashion than would be possible in a single-level definition of the same P-net. The rationale is the following. If a dead-end is found at for example macro-node 3 of the

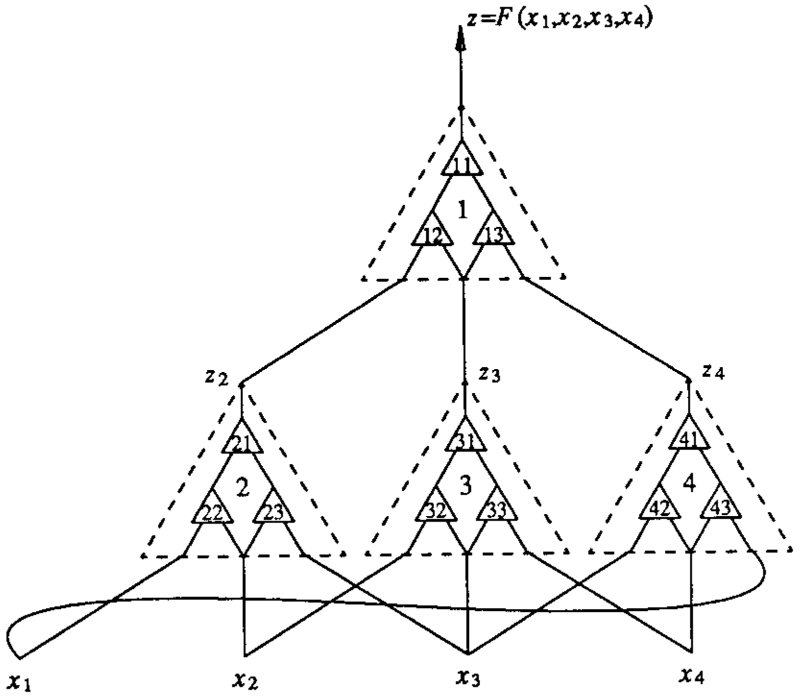


Figure 24: Example of a multilevel P-net.

network in figure 24 (that is, all the options selected for this node generate a reduction conflict) then there is no need to continue with different assignments for the nodes in the macro-node 2. In other words, there is no need to backtrack inside the subnetwork 2 and try different assignments for the nodes 21, 22, or 23 with the same assignment for the subnetwork 2. Such different assignments would only cause the same dead-end at macro-node 3. Hence, the backtracking can be continued until the next assignment for macro-node 2 is found.

The program reads the definition of the network P=[N,z,**x**,I] from a file. N defines the node, z and **x** label the network output and inputs, and I defines the interconnection structure. The network specifications are written in a simple language whose context-free grammar is listed below.

```
<networkdescription> ::= <atomdefinition> ; <networkdefs> .
<atomdefinition> ::= define <representation>
                        functions { <truthtableset> } |
                     define <representation>
                        functions all
<representation> ::= <identifier> = <identifier> ( <parameterlist> )
<identifier> ::= <letterordigit> [ <identifier> ]
<letterordigit> ::= <letter> | <digit>
<parameterlist> ::= <identifier> [ , <parameterlist> ]
<truthtableset> ::= <truthtable> [ , <truthtableset> ]
<truthtable> ::= <zeroorone> [ <truthtable> ]
<networkdefs> ::= <networkdefinition> [ ; <networkdefs> ]
<networkdefinition> ::= define <representation>
                        interconnect { <nodelist> }
<nodelist> ::= <nodedefinition> [ ; <nodelist> ]
<nodedefinition> ::= <identifier> : <representation>
```

The definition of the the P-net of figure 3 is listed below.

```
% Definition of a 3-input P-net with 6 nodes.

% Definition of the node
define z = atom(x1,x2) functions all;
% Definition of the 3-input network
define z = F(x1,x2,x3)
       interconnect { node4 : z4 = atom (x1,x2);
                      node5 : z5 = atom (x2,x3);
                      node6 : z6 = atom (x3,x1);
                      node2 : z2 = atom (z4,z5);
                      node3 : z3 = atom (z5,z6);
                      node1 : z  = atom (z2,z3) }.
```

The definition of the P-network of figure 24 is:

```
% 4-input P-net built with 3-input P-nets.

% Definition of the atomic node
define y = atom(x1,x2) functions all;
% Definition of the 3-input network:
define z = f(x1,x2,x3)
       interconnect { a2 : z2 = atom(x1,x2);
                      a3 : z3 = atom(x2,x3);
                      a1 : z  = atom(z2,z3) };
% Definition of the 4-input network:
define z = F(x1,x2,x3,x4)
       interconnect { m2 : z2 = f(x1,x2,x3);
                      m3 : z3 = f(x2,x3,x4);
                      m4 : z4 = f(x3,x4,x1);
                      m1 : z  = f(z2,z3,z4) }.
```

Lines starting with a "%" are comment lines. The first few lines of the definition specify N≡(k,ϕ). Each **define** statement lists z and **x** and the following **interconnect** statement defines I.

The basic node (called `atom` in both cases) could be defined arbitrarily by giving a list of local truth tables. For example:

```
% 2-input node implementing AND, LEFT, RIGHT, OR
define z = atom(x1,x2)
       functions { 0001, 0011, 0101, 0111 } ;
```

The decomposition program treats the nodes (or macro-nodes) in the order given by the network specification. Therefore the node connections I should be ordered in a bottom-up sequence. A label cannot be used as input to a node unless it was defined earlier as the output label of another node or unless it is a network input. Additionally, the subnetworks must be defined in a depth-first manner, meaning that the node must be defined first, then the subnetworks, and finally the global network. A subnetwork must be defined before it can be used as macro-node in another interconnection definition.

This language allows the specification of any polyfunctional net with only a single node, as defined in section 1. The adventurous algorithm described earlier does not require all nodes to be identical, and the program and definition language could be extended in this direction. However, except for this restriction the program is completely general. It does not make any assumptions regarding the universality of the node, the universality of the network, the structure of the interconnections, or the size of the nodes. Furthermore, it finds multiple solutions, although typically not all of them.

Below is a partial script of the program using the 3-input network of figure 3. The network function is $G(x_1,x_2,x_3) = x_1x_2+x_2x_3+x_3x_1$. Inputs typed by the user are in **bold** face.

```
specify network file: Nw3.6
specify truth table of 3 variables (x1,x2,x3):
01101011
   1: 1 | node4: 0000... ok.
   2: 1 | node5: 0000... conflict.
   3: 1 | node5: 0010... conflict.
   4: 1 | node5: 0100... conflict.
   5: 1 | node5: 0110... ok.
   6: 1 | node6: 0000... conflict.
   7: 1 | node6: 0001... conflict.
   8: 1 | node6: 0100... conflict.
   9: 1 | node6: 0101... conflict.
  10: 1 | node6: 1010... conflict.
  11: 1 | node6: 1011... conflict.
  12: 1 | node6: 1110... conflict.
  13: 1 | node6: 1111... conflict.
  14: 1 | node5: 1001... ok.
          .
          .
          .
  45: 1 | node3: 0101... conflict.
  46: 1 | node3: 0110... ok.
  47: 1 | node1: 0000... conflict.
  48: 1 | node1: 0111... ok.
implementation found after 48 steps:
1 | node4: 0001
1 | node5: 0011
1 | node6: 0110
1 | node2: 0001
1 | node3: 0110
1 | node1: 0111
continue? [y,n,#] no
search aborted.

number of steps:     48
maximum stack size: 366 words
other memory used:  286 records

do you want to run another decomposition? no
```

An example of a decomposition problem for the network of figure 24 with network function $G(x_1,x_2,x_3,x_4) = x_1x_3+x_2(x_1 \oplus x_4)$ is given below:

```
specify network file: Nw4.4.3
specify truth table of 4 variables (x1,x2,x3,x4):
0110100100110011
   1: 1 | m2: 00000000... ok.
   2: 2 | m2.a2: 0000... ok.
   3: 2 | m2.a3: 0000... ok.
   4: 2 | m2.a1: 0000... ok.
   5: 1 | m3: 00000000... conflict.
   6: 1 | m3: 00010010... conflict.
   7: 1 | m3: 00100001... conflict.
   8: 1 | m3: 00110011... conflict.
   9: 1 | m3: 01001000... conflict.
  10: 1 | m3: 01011010... ok.
  11: 2 | m3.a2: 0000... conflict.
  12: 2 | m3.a2: 0011... ok.
  13: 2 | m3.a3: 0000... conflict.
  14: 2 | m3.a3: 0101... ok.
  15: 2 | m3.a1: 0000... conflict.
  16: 2 | m3.a1: 0110... ok.
  17: 1 | m4: 00000000... conflict.
  18: 1 | m4: 00000101... conflict.
  19: 1 | m4: 00001010... conflict.
  20: 1 | m4: 00001111... conflict.
  21: 1 | m4: 01010000... conflict.
  22: 1 | m4: 01010101... conflict.
  23: 1 | m4: 01011010... conflict.
  24: 1 | m4: 01011111... conflict.
  25: 1 | m4: 10100000... conflict.
  26: 1 | m4: 10100101... conflict.
  27: 1 | m4: 10101010... conflict.
  28: 1 | m4: 10101111... conflict.
  29: 1 | m4: 11110000... conflict.
  30: 1 | m4: 11110101... conflict.
  31: 1 | m4: 11111010... conflict.
  32: 1 | m4: 11111111... conflict.
  33: 1 | m3: 01101001... ok.
          .
          .
          .
 145: 2 | m4.a1: 0100... ok.
 146: 1 | m1: 00000000... conflict.
 147: 1 | m1: 00000101... conflict.
 148: 1 | m1: 00101010... ok.
 149: 2 | m1.a2: 0000... conflict.
 150: 2 | m1.a2: 0111... ok.
 151: 2 | m1.a3: 0000... conflict.
 152: 2 | m1.a3: 0001... conflict.
 153: 2 | m1.a3: 0010... conflict.
 154: 2 | m1.a3: 0011... conflict.
 155: 2 | m1.a3: 0100... conflict.
 156: 2 | m1.a3: 0101... ok.
 157: 2 | m1.a1: 0000... conflict.
 158: 2 | m1.a1: 0010... ok.
implementation found after 158 steps:
1 | m2: 00000101
2 | m2.a2: 0011
2 | m2.a3: 0101
2 | m2.a1: 0001
1 | m3: 01101001
2 | m3.a2: 0110
2 | m3.a3: 0101
2 | m3.a1: 0110
1 | m4: 01010000
2 | m4.a2: 0011
2 | m4.a3: 0101
2 | m4.a1: 0100
1 | m1: 00101010
2 | m1.a2: 0111
2 | m1.a3: 0101
2 | m1.a1: 0010
continue? [y,n,#] no
search aborted.

number of steps:    158
maximum stack size: 802 words
other memory used:  453 records

do you want to run another decomposition? no
```

Answering the question "`continue? [y,n,#]`" with "**yes**" will make the program backtrack and find another solution.

4. CONCLUSIONS AND FUTURE RESEARCH

The decomposition problem in its most general form is a hard problem that can require a large amount of search. Sometimes a solution is found early in the search, but there is no guarantee that this will always be the case, or that it will be true on the average. If on the average the size of ϕ is reduced to $p'=|\phi'|$ then it would require $O((p')^l)$ steps to traverse the tree in its entirety. In the worst case, the solution for the decomposition problem requires an almost exhaustive search.

The main issue is not so much the amount of search needed, but the nature of the *responsibility assignment* task, namely how to assign certain pieces of the global system goal to certain parts of the system. In terms of polyfunctional networks this means specifically: how to assign a smaller function to a node or subnetwork given a certain function for the network as a whole. When can an assignment of responsibility be done in a straightforward fashion? When is it easy and when is it hard?

The multilayered nature of the network is not an obstacle in determining which part of the network should implement a certain piece of the network function. The example of the disjunctive tree shows that for any network function implementable on this network the assignments for all the nodes can always be decided by a straightforward algorithm. The number of node assignments is linear in the number of nodes. What is it that makes the assignment so simple in this case and so complex in others?

The size of the node versus the size of the whole network is an important issue. If the node or subnetwork to be assigned responsibility is only a little smaller in size than the whole network, then the problem is simpler than if the node is much smaller. In the latter case the network function must satisfy strict requirements in order for the assignment to be focussed to one or a few possibilities (as in the case of the disjunctive binary tree). In the former case most network functions allow a simple solution. If the network function happens to depend only on the variables that are inputs to a given node then the assignment is solved trivially.

In general, the sources of difficulty are the fan-out connections (external or internal). Fan-outs introduce virtual variables, which in turn introduce *don't care* entries in the network truth table. The decomposition then requires a search. Furthermore, in such a case the responsibility assignment is not unique, because the structure is redundant.

The fundamental reason behind the necessity for a search is that the assignment cannot be entirely local and must use global knowledge of the network structure. If multiple assignments are possible at a given node, each one will produce a different residual function. Deciding which local assignments are correct requires knowing which residual functions are implementable on the remaining network. Coordinating local assignments with global requirements is therefore of crucial importance. The local assignments should depend not only on the assignments made to previous nodes, but also on the structure of the remainder of the network (the part that has not yet been considered by the search). The question whether a (residual) function is implementable on a (remainder of a) P-net is different from the problem of actually finding an implementation. The former is a binary question; the latter is a search for a particular solution. We do not know yet whether the former is computationally simpler.

Different traversals of the nodes are possible, opening a possible improvement to the decomposition algorithm. Some orderings might lead to a faster solution (less backtracking) than others. Which order should be preferred? It depends on the network function. The node with the simplest assignment, the smallest number p', should be taken first. This node has the smallest number of mutually incompatible groups of columns in its decomposition chart, or the maximum number of mutually compatible columns. Finding this node is a different problem than the original decomposition problem, and its complexity remains to be evaluated.

An ordering of nodes such that at each step p' is minimal would reduce the complexity of the problem substantially. Finding this optimal order of traversal can be facilitated by using parallelism. Assume that all the nodes in the bottom layer receive the global function at the same time and compute ϕ' concurrently. A global control mechanism can then designate a single winner to perform the reduction step. The same process can be repeated for the nodes in the bottom layer of the remainder of the network, and so on. Using parallelism the nodes are treated in an *optimal* order at no cost in time. By comparing the size of this optimal search path with the size of an average search path one could determine if the extra cost of parallelism is offset by the improvements in speed.

Three important topics regarding the decomposition problem are open for future research.

a. Existing decomposition algorithm

Many aspects of the decomposition algorithm need to be explored not only to improve not only the performance of the algorithm itself, but also to provide more insight into the responsibility assignment problem in general. Several improvements to the selection criterion are possible, which would further focus the search. Further theoretical or statistical analysis of the performance of the algorithm is needed. The development of estimates (instead of a very pessimistic upper bound) for S_{av} would allow a better characterization of the complexity of the decomposition problem. Finally, the possibilities for exploiting parallelism in the decomposition process itself should be explored.

b. Top-down versus bottom-up strategies

A complementary decomposition algorithm, proceeding in a top-down fashion, has not been developed or explored. Still, a top-down procedure would be intuitively very appealing. The difficulties encountered in such an approach are different from those in a bottom-up strategy, but the exact nature of these differences remains to be explored. Insight gained from understanding a complementary strategy would undoubtedly shed additional light on the bottom-up strategy as well. A unification of both approaches may be the ultimate goal.

c. Incompletely specified functions.

The existing decomposition algorithm does not assume that the given network function is completely specified: it is already dealing with *don't cares* internally. However, so far we have always considered cases of completely specified functions, and possible shortcuts with *don't cares* have not been examined. It would again be desirable to do a theoretical or statistical analysis of this issue and identify appropriate ways of treating *don't cares*. *R*

REFERENCES

1. Aleksander, I., ''Structure/Function Considerations for Digital Systems that Contain Polyfunctional Elements,'' *Computers and Digital Techniques* **1**(4), pp. 165-170 (October 1978).
2. Curtis, H.A., ''A Generalized Tree Circuit,'' *Journal of the ACM* **8**(4), pp. 484-496 (October 1961).
3. Curtis, H.A., ''Generalized Tree Circuit - The Basic Building Block of an Extended Decomposition Theory,'' *Journal of the ACM* **10**(4), pp. 562-581 (October 1963).
4. Friedman, A.D. and P.R. Menon, *Theory and Design of Switching Circuits,* Computer Science Press, Inc., Woodland Hills, CA (1975).
5. Hight, S.L., ''Complex Disjunctive Decomposition of Incompletely Specified Boolean Functions,'' *IEEE Transactions on Computers* **C-22**(1), pp. 103-110 (January 1973).
6. Johnson, N.L. and S. Kotz, *Urn Models and Their Application,* John Wiley & Sons (1977).
7. Rosenblatt, F., *Principles of Neurodynamics,* Spartan Books, Washington, D.C. (1962).
8. Urbano, R.H., ''Structure and Function in Polyfunctional Nets,'' *IEEE Transactions on Computers* **C-17**(2), pp. 152-173 (February 1968).