S. J. Hong
R. G. Cain
D. L. Ostapko

# MINI: A Heuristic Approach for Logic Minimization

**Abstract:** MINI is a heuristic logic minimization technique for many-variable problems. It accepts as input a Boolean logic specification expressed as an input-output table, thus avoiding a long list of minterms. It seeks a minimal implicant solution, without generating all prime implicants, which can be converted to prime implicants if desired. New and effective subprocesses, such as expanding, reshaping, and removing redundancy from cubes, are iterated until there is no further reduction in the solution. The process is general in that it can minimize both conventional logic and logic functions of multi-valued variables.

## Introduction

• *Minimization problem*

The classical approach to two-level Boolean logic minimization uses a two-step process which first generates all prime implicants and then obtains a minimal covering. This approach, developed by Quine [1, 2] and McCluskey [3], is a considerable improvement over constructing and comparing all possible solutions. The generation of prime implicants has evolved to a relatively simple process as a result of the efforts of Roth [4], Morreale [5], Slagle et al. [6] and many others. However, the number of prime implicants of one class of $n$-variable functions is proportional to $3^n/n$ [7]. Thus, for many functions, the number of prime implicants can be very large. In addition, the covering step poses an even greater problem because of its well known computational complexity. Because of the required storage and computations, machine processing to obtain the minimum solution by the classical approach becomes impractical for many-variable problems.

Many attempts have been made to increase the size of problems that can be minimized by sacrificing absolute minimality or modifying the cost function used in covering [6, 8–11]. Su and Dietmeyer [12] and Michalski [13, 14] have reported other serious departures from the classical approach. One recently developed computer program, which essentially represents the state of the art, is said to be able to handle functions of as many as 16 variables [15]. Successful minimization of selected larger functions has also been reported [4, 14]. However, many practical problems of 20 to 30 input variables cannot be handled by the approaches described above and it does not appear that the classical approach can be easily extended to encompass functions of that size.

• *Heuristic approach*

The approach presented here differs from the classical one in two aspects. First, the cost function is simplified by assigning an equal weight to every implicant. Second, the final solution is obtained from an initial solution by iterative improvement rather than by generating and covering prime implicants.

Limiting the cost function to the number of implicants in the solution has the advantage of eliminating many of the problems associated with local minima. Since only the number of implicants is important, their shapes can be altered as long as the coverage of the minterms remains proper. The methods of modifying the implicants are similar to those that one might use in minimizing a function using a Karnaugh map. The MINI process starts with an initial solution and iteratively improves it. There are three basic modifications that are performed on the implicants of the function. First, each implicant is reduced to the smallest possible size while still maintaining the proper coverage of minterms. Second, the implicants are examined in pairs to see if they can be reshaped by reducing one and enlarging the other by the same set of minterms. Third, each implicant is enlarged to its maximal size and any other implicants that are covered are removed. Thus, both the first process, which may reduce an implicant to nothing, and the third process, which removes covered implicants, may reduce the number of implicants in the solution. The second process facilitates

the reduction of the solution size that occurs in the other two processes. The order in which the implicants are reduced, reshaped, and enlarged is crucial to the success of the procedure. The details of these processes and the order in which they are applied to the implicants is discussed in later sections. However, the general approach is to iterate through the three main procedures until no further reduction is obtained in the size of the solution.

Our algorithm is designed for minimizing "shallow functions," those functions whose minimal solution contains at most a few hundred implicants regardless of the number of variables. Most practical problems are of this nature because designers usually work with logic specifications that contain no more than a few hundred conditions. The designer is able to express the function as a few hundred implicants because the statement of the problem leads to obvious groupings of minterms. The purpose of the algorithm is to further minimize the representation by considering alternative groupings that may or may not be obvious from the statement of the problem.

To facilitate the manipulation of the implicants in the function, a good representation of the minterms is necessary. The next section describes the cubical notation that is used.

• *Generalized cube format*

The universe of $n$ Boolean variables can be thought of as an $n$-dimensional space in which each coordinate represents a variable of two values, 0 or 1. A Karnaugh map is an attempt to project this $n$-dimensional space onto a two-dimensional map, which is usually effective for up to five or six variables. Each lattice point (vertex) in this $n$-dimensional space represents a *minterm*, and a special collection of these minterms forms an *implicant*, which is seen as a cube of vertices. Following Roth [4], the usual definition of a cube is an $n$-tuple vector of 0, 1 and $X$, where 0 means the complement value of the variable, 1 the true value, and $X$ denotes either 0 or 1 or both values of the variable. The following example depicts the meaning of the usual cube notation.

*Example 1a*

Consider a four-variable ($A, B, C$ and $D$) universe.

| *Implicant* | *Cube notation* | *Meaning* |
|---|---|---|
| $\overline{A}\,\overline{B}\,C\,\overline{D}$ | 0 0 1 0 | Minterm with $A = B = D = 0, C = 1$ |
| $A\,\overline{C}$ | 1 $X$ 0 $X$ | Minterms with $A = 1$, $B = 0$ or 1, $C = 0$, $D = 0$ or 1 |
| U = universe | $X\ X\ X\ X$ | Minterms with $A = 0$ or 1, $B = 0$ or 1, $C = 0$ or 1, $D = 0$ or 1 |
| $\emptyset$ = null | $\emptyset$ | No minterms |

A more convenient machine representation of 0, 1 and $X$ in the cube is to denote them as binary pairs, i.e., to code 0 as 10, 1 as 01, and $X$ as 11. This representation has the further meaning that 10 is the first of the two values (0) of the variable, 01 is the second value (1), and 11 is the first or the second or both values. Naturally, the code 00 represents no value of the variable and, hence, any cube containing a 00 for any variable position depicts a null cube.

*Example 1b*

Consider the encoded cube notation of Example 1a.

| *Cubes* | *Encoded cubes* |
|---|---|
| 0 0 1 0 | 10 10 01 10 |
| 1 $X$ 0 $X$ | 01 11 10 11 |
| $X\ X\ X\ X$ | 11 11 11 11 |
| $\emptyset$ | 10 00 11 01 |

(The 00 entry can be in any variable position. The other values are immaterial.)

We call this encoded cube notation a *positional* cube notation since the positions of the 1's in each binary pair denote the occupied coordinate values of the corresponding variables. With this notation, any non-Boolean variable, which has multiple values, can be accommodated in a straightforward manner. If a variable has $t$ values, the portion corresponding to that variable in the positional cube notation is a binary $t$-tuple. The positions of each 1 in this $t$-tuple denote the values of the $t$-valued variables occupied by the minterms in the cube. Su and Cheung [16] use this positional cube notation for the multiple-value logic. A Boolean variable is a special case of the multiple-value variable.

Consider $P$ variables; let $p_i$ denote the number of values the variable $i$ takes on. We call the $p_i$-tuple in the positional cube notation the $i$th *part* of the cube (there are $P$ parts); $p_i$ is called the part size, which is the total number of values there are in the $i$th coordinate of the $P$-dimensional multiple-value logic space. Notice that in a cube, the values specified by the 1's in a part are to be ORed, and this constrained part is to be ANDed with other parts to form an implicant.

Any Boolean (binary) output function $F$ with $P$ multiple-value inputs can be mapped into a $P$-dimensional space by inserting 1's in all points where $F$ must be true and 0's in all points where $F$ must be false. The unspecified points can be filled with $d$'s, meaning the DON'T CARE output conditions. (Often, the 1's and $d$'s are specified and the 0's are filled later.) A list of cubes represents the union of the vertices covered by each cube and is called a cubical cover of the vertices, or simply a *cover*. The goal of the MINI procedure is to cover all of the 1's and none of the 0's with a cover containing a minimum number of cubes. The covers exclusively covering the 1's,

0's, and unspecified points are called, respectively, the ON cover, the OFF cover, and the DON'T CARE cover. When there is no confusion, these covers will be denoted by $F$, $\overline{F}$, and $DC$, respectively.

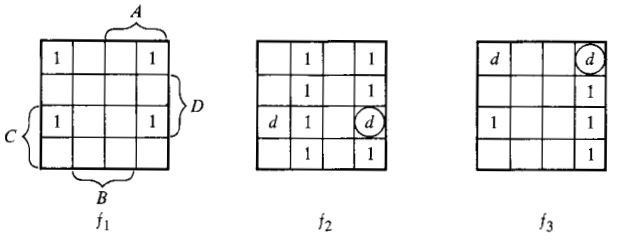
For multiple-Boolean-output functions $(f_1, f_2, \cdots, f_m)$, a tag field [18] has been catenated to the input portion of a cube to denote the multiple-output implicant. We can add an additional $m$ valued dimension for the outputs. This new dimension can be interpreted as representing a multiple-value variable called the *output*. The traditional tag field of an $m$-tuple binary vector corresponds to our output part in a cube. If the $i$th bit of the output part is a 1, the $i$th output is occupied by the cube. We call the whole multiple-output space the generalized universe. Any cube in this universe automatically denotes a multiple-output cube. We denote by $F$ the whole of the multiple-output functions $f_1$ through $f_m$. The MINI procedure aims to cover $F$ with a minimal number of cubes in the generalized space.

For generality, we also group input variables into a set multiple-value variables such that the new variables $X_i$ comprising $n_i$ of Boolean input variables have $2^{n_i}$ values and are called parts. The part sizes are defined as $p_i$ for inputs and $m$ for the output. When groups of inputs are processed through small decoders, the values of decoder output correspond to the multiple values of parts. Each part constitutes a coordinate in the generalized space. The specification of the function is assumed to be a list of regular Boolean cubes with the output tags. The output tag is composed of 0, 1, and $d$, where 0 means no information, 1 means the cube belongs to the output, and $d$ means the cube is a DON'T CARE for the output. The output side of this specification is the same used by Su and Dietmeyer [12], sometimes known as the output connection matrix.

*Example 2a*

A Boolean specification and its Karnaugh map.

| *Inputs* | | | | *Outputs* | | |
|---|---|---|---|---|---|---|
| $A$ | $B$ | $C$ | $D$ | $f_1$ | $f_2$ | $f_3$ |
| 0 | 1 | $X$ | $X$ | 0 | 1 | 0 |
| 1 | 0 | $X$ | $X$ | 0 | 1 | 1 |
| $X$ | 0 | 0 | 0 | 1 | 0 | $d$ |
| $X$ | 0 | 1 | 1 | 1 | $d$ | 1 |




The circled $d$'s in the Karnaugh map show the conflict between 1's and $d$'s. We allow the specification to have conflicts for the sake of enabling the designer to write a concise specification. Any such conflict will be overridden by the $d$'s in our MINI process. Suppose now the inputs are partitioned as $X_1 = \{A, B\}$ and $X_2 = \{C, D\}$. The specification of Example 2a is preprocessed to the generalized positional cube notation as shown below. We call this preprocess a decoding step.

*Example 2b*

Decoding Boolean specification into the cube format: There are three parts; $X_1$ and $X_2$, which take on the four values 00, 01, 10 and 11, and the output, with part size 3. The DON'T CARE cover overrides the ON cover.

| $X_1$ | $X_2$ | *Output* | |
|---|---|---|---|
| 0100 | 1111 | 010 | |
| 0010 | 1111 | 011 | $F$ |
| 1010 | 1000 | 100 | $F$ |
| 1010 | 0001 | 101 | $F$ |
| 1010 | 1000 | 001 | $DC$ |
| 1010 | 0001 | 101 | $DC$ |

The first four cubes for $F$ (ON cover) are the decoded cubes in Example 2a with the output $d$'s replaced with 0's. The last two $DC$ cubes are obtained by decoding only those cubes with $d$'s and replacing the $d$'s with 1's and any non-$d$ output with 0's.

### •• *Classical concepts in cubical notation*

Several classical concepts have immediate generalizations to the cube structure described in the previous section. The correspondences between a minterm and a point and between an implicant and a cube have already been described. In addition, a prime implicant corresponds to a cube in which no part can admit any more 1's without including some of the $\overline{F}$ space. Such a cube is called a prime cube.

A useful concept in minimization is the size of a cube, which is the number of minterms that the cube contains. It follows from this definition that the size of a cube is independent of the partition of the space into which it is mapped or decoded. Thus, the size of a cube is given by

$$\textit{cube size} = \prod_{i=1}^{P} (\text{number of 1's in part } p_i). \qquad (1)$$

Since a cube with one variable per part represents the usual Boolean implicant, each implicant can be mapped into any partitioned cube. Because the resulting cubes can in some cases be merged when the Boolean implicants could not, we have the following theorem.

*Theorem 1* The minimum number of cubes that can represent a function in a partitioned space is less than or equal to the minimum number of cubes in the regular Boolean minimization.

To manipulate the cube representation of a function, it is necessary to define the OR, AND, and NOT operations.

1. The OR of two cubes $C_1$ and $C_2$ is a list containing $C_1$ and $C_2$. The OR of two covers $A$ and $B$ is thus the catenation of the two lists.
2. The AND of two cubes $C_1$ and $C_2$ is a cube formed by the bit-by-bit AND of the two cubes. The AND of two covers $A$ and $B$ follows from the above by distributing the AND operation over the OR operation.
3. The NOT of a cube or cover is a list containing the minterms of the universe that are not contained in the cube or cover. The algorithm for constructing this list is discussed in a later section.

The simplest way to decrease the number of cubes of a given problem is to merge some of the cubes in the list. Although this is not a very powerful process, it is well worth applying to the initial specification, especially if there are many entries (a minterm-by-minterm specification is a good example). The following shows the merging of two cubes, which is similar to the merging of two unit-distance Boolean implicants, e.g., $AB\bar{C} \vee ABC = AB$.

*Definition* The *distance* between two cubes $C_1$ and $C_2$ is defined as the number of parts in which $C_1$ and $C_2$ differ.

*Lemma 1* If $C_1$ and $C_2$ are distance one apart, then $C_1 \vee C_2 = C_3$, where $C_3$ is a bit-by-bit OR of $C_1$ and $C_2$.

*Proof* Let us assume that the difference is in the first part. Let

$$C_1 = a_1 a_2 \cdots a_{p_1} | b_1 b_2 \cdots b_{p_2} | \cdots | n_1 n_2 \cdots n_{p_p}$$

and

$$C_2 = \alpha_1 \alpha_2 \cdots \alpha_{p_1} | b_1 b_2 \cdots b_{p_2} | \cdots | n_1 n_2 \cdots n_{p_p},$$

where $a_i, b_i, \cdots$, are 0 or 1. Q.E.D.

The cubes $C_1$ and $C_2$ are identical in all but one dimension or part. Therefore, the vertices covered by $C_1 \vee C_2$ can be covered by a single cube with the union of all coordinate values of $C_1$ and $C_2$ in that differing coordinate, i.e., $(a_1 \vee \alpha_1), (a_2 \vee \alpha_2), \cdots, (a_{p_1} \vee \alpha_{p_2})$.

The concept of subsumption in cubes is similar to subsumption in the Boolean case ($AB\bar{C} \vee B\bar{C} = B\bar{C}$).

*Definition* A cube $C_2$ is said to *cover* another cube $C_1$ if for every 1 in $C_1$ there is a corresponding 1 in $C_2$. In other words, $C_1$ AND NOT $C_2$ (bit-by-bit) is all 0's. Since the cube $C_1$ is completely contained in $C_2$, it can be removed from the list, thus reducing the number of cubes of the solution in progress.

*Example 3*
Consider a three-part example as follows.

$$F = \begin{cases} 1\,0\,1\,0 & 0\,1 & 1\,0 & \text{cube 1} \\ 1\,0\,1\,0 & 1\,0 & 1\,0 & \text{cube 2} \\ 0\,0\,1\,0 & 1\,1 & 1\,0 & \text{cube 3} \end{cases}$$

Cube 1 and cube 2 are distance one apart since they differ only in the second part. The result of merging these two cubes is 1010 11 10, which covers cube 3. Hence, $F$ reduces to one cube, 1010 11 10.

## Description of MINI and some theoretical considerations

• *MINI philosophy*
The minimization process starts from the given initial $F$ cover and $DC$ cover (lists of cubes where each cube has $P$ parts). Each part of a cube can be viewed as designating all allowed values of the multiple-valued logic variable, corresponding to that part. The output part can be interpreted as merely another multiple-value variable which may be called the output. When each part's allowed values are ANDed, the resulting cube describes some of the conditions to be satisfied for the given multiple-output logic function corresponding to $F$ and $DC$ specifications. The objective, then, is to minimize the number of cubes for $F$ regardless of the size and shape of the constituent cubes. This corresponds to minimizing only the number of AND gates without fan-in limit, in the regular Boolean two-level AND-OR minimization. We discuss later a simple way of modifying the solution to suite the classical cost criterion.

The basic idea is to merge the cubes in some way toward the minimum number. To do this, MINI first "explodes" the given $F$ cover into a disjoint $F$ cover where the constituent cubes are mutually disjoint. The reasons are

1. To avoid the initial specification dependency. The given cubes may be in an awkward shape to be merged.
2. To introduce a reasonable freedom in clever merging by starting with small, but not prohibitively numerous, fragments such as a minterm list.

The disjoint $F$ is an initial point of the ever decreasing solution. At any point of the process from there on, a guaranteed cover exists as a solution. A subprocess called *disjoint sharp* is used for obtaining the disjoint $F$.

Given a list of cubes as a solution in progress, a merging of two or more cubes can be accomplished if a larger cube containing these cubes can be found in $F \vee DC$

space to replace them. The more merging is done, the smaller the solution size becomes. We call this the cube *expansion* process. The expansion first orders the given cubes and proceeds down the list until no more merging is possible. This subprocess is not unlike a human circling the "choice" prime implicants in a Karnaugh map. Obviously, one pass through this process is not sufficient.

The next step is to reduce the size of each cube to the smallest possible one. The result of the cube expansion leaves the cubes in near prime sizes. Consequently, some vertices may be covered by many cubes unnecessarily. The cube *reduction* process trims all the cubes in the solution to increase the probability of further merging through another expansion step. Any redundant cube is removed by the reduction and, hence, it also ensures a nonredundant cover.

The trimmed cubes then go through the process called the cube *reshaping*. This process finds all pairs of cubes that can be reshaped into other pairs of disjoint cubes covering the same vertices as before. This step ends the preparation of the solution for another application of cube expansion.

The three subprocesses, expansion, reduction, and reshaping, are iteratively applied until there is no more decrease in the solution size. This is analogous to the trial and error approach used in the Karnaugh map method. We next describe each of these subprocesses and discuss the heuristics used. Brief theoretical considerations are given to formulate new concepts and to justify some of the heuristics.

• *Disjoint sharp process (complementation)*

The sharp operation $A \# B$, defined as $A \wedge \bar{B}$, is well known. It also yields the complement of $A$ since $\bar{A} = U \# A$, where $U$ denotes the universe. Roth [4] first defined the process to yield the prime implicants of $A\bar{B}$ and used it to generate all prime implicants of $F$ by computing $U \# (U \# (F \vee DC))$. He later reported [17] that Junker modified the process to yield $A\bar{B}$ in mutually disjoint implicants. The operation is easily adapted to our general cubical complex as described in this section.

The disjoint sharp operation, $\textcircled{\#}$, is defined as follows: $A \textcircled{\#} B$ is the same cover as $A\bar{B}$, and the resultant cubes of $A \textcircled{\#} B$ are mutually disjoint. To obtain this, we give a procedural definition of $\textcircled{\#}$ by which $A \textcircled{\#} B$ can be generated. Consider two cubes $A = \pi_1\pi_2 \cdots \pi_p$ and $B = \mu_1\mu_2 \cdots \mu_p$.

*Lemma 2* $A \textcircled{\#} B = C = \vee_{i=1}^{P} C_i$, where $C_i$ is given by

$$C_1 = (\pi_1\bar{\mu}_1)\ \pi_2\ \pi_3 \cdots \pi_p,$$

$$C_2 = (\pi_1\mu_1)\ (\pi_2\bar{\mu}_2)\ \pi_3 \cdots \pi_p,$$

$$C_3 = (\pi_1\mu_1)\ (\pi_2\mu_2)\ (\pi_3\bar{\mu}_3) \cdots \pi_p,$$

$$\vdots$$

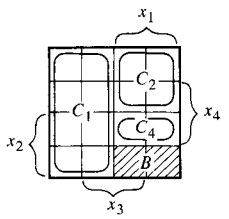
$$C_p = (\pi_1\mu_1)\ (\pi_2\mu_2)\ (\pi_3\mu_3) \cdots (\pi_p\bar{\mu}_p), \qquad (2)$$

and AND and NOT operations are performed in a bit-by-bit manner. Whenever any $C_i$ becomes a null cube, i.e., $\pi_i\bar{\mu}_i = \emptyset$, $C_i$ is removed from the $A \textcircled{\#} B$ list.

*Proof* It is obvious that the $C_i$ are mutually disjoint; $C = A\bar{B}$ has to be shown. Since for all $i$, $C_i \subseteq A$ and $C_i \subseteq \bar{B}$, we have $C \subseteq A\bar{B}$. We must show now that every vertex $W \in A\bar{B}$ also belongs to $C$. Let $W = w_1w_2 \cdots w_p$; then each $w_i$ is covered by $\pi_i$ and there exists at least one $w_i$ which is not covered by $\mu_i$. Let the first part where $w_i$ is not covered by $\mu_i$ be $\hat{\imath}$. From Eq. (2), we see that $W \in C_{\hat{\imath}}$ and, therefore, $w \in C$. Q.E.D.

*Example 4a*

A Karnaugh map example for $A$ = universe $= XXXX$ ($\pi_1\pi_2\pi_3\pi_4 =$ 11 11 11 11 in our notation) and $B = 11X0$ ($\mu_1\mu_2\mu_3\mu_4 =$ 01 01 11 10), the shaded area of the map. Then




$$C_1 = 0XXX\ (10\ 11\ 11\ 11),$$
$$C_2 = 10XX\ (01\ 10\ 11\ 11),$$
$$C_3 = \text{null}\quad (01\ 01\ \underline{00}\ 11) - (\text{delete}),$$
$$C_4 = 11X1\ (01\ 01\ 11\ 01).$$

Equation (2) can be expressed more concisely as

$$C_j = (\pi_1 \wedge \mu_1)(\pi_2 \wedge \mu_2) \cdots (\pi_{j-1} \wedge \mu_{j-1})(\pi_j \wedge \bar{\mu}_j) \times \pi_{j+1}\pi_{j+2} \cdots \pi_p, \qquad (3)$$

which shows that the $\mu_j$ are complemented in order from part 1 through part $P$. The parts can be complemented in an arbitrary order and still produce a valid $A \textcircled{\#} B$. Let $\sigma$ denote an arbitrary permutation on the index set 1 through $P$. Then Eq. (3) can be rewritten as

$$C_j = (\pi_{\sigma(1)} \wedge \mu_{\sigma(1)})(\pi_{\sigma(2)} \wedge \mu_{\sigma(2)}) \cdots (\pi_{\sigma(j-1)} \wedge \mu_{\sigma(j-1)})(\pi_{\sigma(j)} \wedge \bar{\mu}_{\sigma(j)}) \pi_{\sigma(j+1)} \cdots \pi_{\sigma(p)}. \qquad (4)$$

It is easily shown that the proof of Lemma 2 is still valid if the index set is replaced by the permuted index set. In addition, $A \textcircled{\#} B$ may be performed for any given permutation and the result will always yield the same number of cubes. However, the shapes of the resultant cubes can vary depending on the part permutation $\sigma$, as shown in Example 4b.

*Example 4b*

Let $A = 1101\ 10\ 11$ and $B = 0101\ 11\ 01$. We calculate $A \circledast B$ with two distinct part-permutations, using Eq. (4).

$$A \circledast B = \begin{Bmatrix} 1000\ 10\ 11 \\ 0101\ 10\ 10 \end{Bmatrix} \text{ if } \sigma = (1, 2, 3)(C_2 = \emptyset);$$

$$A \circledast B = \begin{Bmatrix} 1101\ 10\ 10 \\ 1000\ 10\ 01 \end{Bmatrix} \text{ if } \sigma = (2, 3, 1)(C_1 = \emptyset).$$

The extension of the $\circledast$ operation to include the covers as the left- and the right-side arguments is similar to the regular # case. One difference is that the left-side argument cover $F$ of $F \circledast G$ must already be disjoint to produce the desired disjoint $F\bar{G}$ cover. Thus, when $F = \vee f_i$ with the $f_i$ disjoint and $g$ is another cube, $F \circledast g$ is defined as

$$F \circledast g = \vee (f_i \circledast g). \tag{5}$$

If $G = \vee_{j=1}^{n} g_j$, where the $g_j$ are not necessarily disjoint,

$$F \circledast G = ((\cdots((F \circledast g_1) \circledast g_2)\cdots) \circledast g_{n-1}) \circledast g_n. \tag{6}$$

If $F$ is not in disjoint cubes, the above calculations still produce a cover of $F\bar{G}$, but the resultant cubes may not be disjoint. The proof of the above extensions of $\circledast$ is simple and we omit it here.

The definition of $F \circledast G$ given in Eq. (6) can be generalized to include the permutation on the cubes of $G$. One can replace each $g_i$ in Eq. (6) with a permuted indexed $g_{\sigma(i)}$. This cube ordering $\sigma$ for the right-side argument $G$ influences the shape and the number of resultant cubes in $F \circledast G$. Example 5 illustrates the different outcome of $F \circledast G$ depending on the order of the cubes of $G$.

*Example 5*

Let $F$ be the universe and $G$ be given as follows.

Produce $\bar{G}$ disjoint $= F \circledast G$.

$$F = 11\ 1111\ 11$$

$$G = \begin{Bmatrix} 10\ 1101\ 11 - g_1 \\ 11\ 0010\ 01 - g_2 \end{Bmatrix}$$

$$F \circledast g_1 = \begin{Bmatrix} 01\ 1111\ 11 \\ 10\ 0010\ 11 \end{Bmatrix}$$

$$(F \circledast g_1) \circledast g_2 = \begin{Bmatrix} 01\ 1101\ 11 \\ 01\ 0010\ 10 \\ 10\ 0010\ 10 \end{Bmatrix}$$

$$F \circledast g_2 = \begin{Bmatrix} 11\ 1101\ 11 \\ 11\ 0010\ 10 \end{Bmatrix}$$

$$(F \circledast g_2) \circledast g_1 = \begin{Bmatrix} 01\ 1101\ 11 \\ 11\ 0010\ 10 \end{Bmatrix}$$

The part ordering $\sigma = (1, 2, 3)$ is used for both cases to show the effect of just $g_1, g_2$ ordering.

As shown by examples 4b and 5, there are two places where permutation of the order of carrying out the $\circledast$ process affects the number of cubes in the result. One is the *part* ordering in cube-to-cube $\circledast$, and the other is the *right-argument* cube ordering. The choice of these two permutations makes a considerable difference in the number of cubes of $F \circledast G$. Since we obtain $\bar{F}$ as $U \circledast (F \vee DC)$ and $F$ as $U \circledast (\bar{F} \vee DC)$ initially, we choose these permutations such that a near minimal number of disjoint cubes will result. The detailed algorithm on how these permutations are selected is presented in a later section. We mention here that these permutations do not affect the outcome in the case of the regular sharp process, because the regular sharp produces *all* prime cubes of the cover. The disjoint $\bar{F}$ obtained in the process of obtaining the disjoint $F$ as above is put through one pass of the cube expansion process (see next section) to quickly reduce the size and thus facilitate the subsequent computations. The $\bar{F}$ used thereafter need not be disjoint.

When the left argument of $\circledast$ is the universe, the result is the complement of the right argument. Since we treat the multiple Boolean outputs $f_1, f_2, \cdots, f_m$ as one part of a single generalized function $F$, we now explain the meaning of $\bar{F}$.

*Theorem 2* The output part of $\bar{F}$ represents $\bar{f}_1, \cdots, \bar{f}_m$.

*Proof* The complementation theorem in [18] states that if $F = \vee E_i f_i$, $\vee E_i = 1$ and $E_i f_i = f_i$, then $\bar{F} = \vee E_i \bar{f}_i$. Let $E_i$ in our case be the whole plane of $f_i$ in the universe; i.e., $E_i$ is a cube denoted by all 1's in every input part and a single 1 in the $i$th position of the output part. Obviously, $\vee E_i = 1$, $E_i F = E_i f_i = f_i$, and $F = \vee f_i = \vee E_i f_i$. Hence, $\bar{F} = \vee E_i \bar{f}_i = \vee \bar{f}_i$. Q.E.D.

• *Cube expansion process*

The cube expansion procedure is the crux of the MINI process. It is principally in this step that the number of cubes in the solution decreases. The process examines the cubes one at a time in some order and, from a given cube, finds a prime cube covering it and many of the other cubes in the solution. All the covered cubes are then replaced by this prime cube before a next remaining cube is expanded.

The order of the cubes we process is decided by a simple heuristic algorithm (described later). This ordering tends to put those cubes that are hard to merge with others on the top of the list. Therefore, those cubes that contain any essential vertex are generally put on top of the other cubes. This ordering approximates the idea of taking care of the extremals first in the classical covering step. Thus, a "chew-away-from-the-edges" type of merging pattern evolves from this ordering.

Let $S$ denote the solution in progress; $S$ is a list of cubes which covers all $F$-care vertices and none of the $\bar{F}$-care

vertices, possibly covering some of the $DC$ vertices. Now, from a given cube $f$ of $S$, we find another cube in $F \vee DC$, if any, that will cover $f$ and hopefully many of the other cubes in $S$, to replace them. This is accomplished by first expanding the cube $f$ into one prime cube that "looks" the best in a heuristic sense. The local extraction method (see, for instance, [7]), also builds prime cubes around the periphery of a given cube. The purpose there is to find an extremal prime cube in the minimization process. Even though the local extraction approach does not generate all prime cubes of the function, it does generate all prime cubes in the peripheries, which can still be too costly for many-variable problems. To approximate the power of local extraction, the expansion process relies on the cube ordering and other subprocesses to follow. Since only one prime cube is grown and no branching is necessary, the cube expansion process requires considerably less computation than the local extraction process.

The expansion of a cube is done one part at a time. We denote by $SPE(f; k)$ the single-part expansion of $f$ along part $k$; $SPE$ can be viewed as a generalized implementation of Roth's coface operation on variable $k$.

*Definition* Two disjoint cubes $A$ and $B$ are called *k-conjugates* if and only if $A$ and $B$ have only one part $k$ where the intersection is null; i.e., when part $k$ of both $A$ and $B$ is replaced with all 1's, the resultant cubes are no longer disjoint.

*Example 6*
Let $f$ be $101X$ in regular Boolean cube notation. The cubes $0X11$, $X1XX$ and $1000$ are examples of 1-, 2- and 3-conjugates of $f$, respectively. There is no 4-conjugate of $f$ in this case.

Let $H(f; k)$ be the set of all cubes in $\bar{F}$ that are $k$-conjugates of the given cube $f$ in $S$.

$$H = \{g_i | f \text{ and } g_i \text{ are } k\text{-conjugates}\}, \tag{7}$$

where we assume that the $\bar{F}$ is available as $\bar{F} = \vee g_i$, which is obtained as a by-product of the disjoint $F$ calculation. (Since the cube expansion process makes use of $\bar{F}$, we say that $S$ is expanded against $\bar{F}$.) Further denote as $Z(f; k)$ the bit-by-bit OR of the part $k$ of all cubes in $H(f; k)$. When $H(f; k)$ is a null set, $Z(f; k)$ is all 0's. The single-part expansion of $f = \pi_1\pi_2 \cdots \pi_k \cdots \pi_p$ along part $k$ is defined as

$$SPE(f; k) = \pi_1\pi_2 \cdots \pi_{k-1}\, \overline{Z(f; k)}\, \pi_{k+1} \cdots \pi_p, \tag{8}$$

where $\bar{Z}$ denotes bit-by-bit complementation.

*Example 7a*
Let $f$ and $\bar{F}$ be as follows. The $SPE$ along parts 1, 2 and 3 is obtained.

$$f = \ 10\ 10\ 0110$$

$$\bar{F} = \begin{cases} 11\ 11\ 1000 = g_1, \\ 11\ 01\ 0011 = g_2, \\ 01\ 10\ 0001 = g_3. \end{cases}$$

Then,

$$H(f; 1) = \emptyset \text{ and } Z(f; 1) = 00$$

$$SPE(f; 1) = \underline{11}\ 10\ 0110,$$

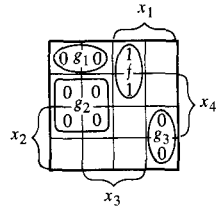
$$H(f; 2) = \{g_2\} \text{ and } Z(f; 2) = 01$$

$$SPE(f; 2) = 10\ \underline{10}\ 0110 = f,$$

$$H(f; 3) = \{g_1\} \text{ and } Z(f; 3) = 1000$$

$$SPE(f; 3) = 10\ 10\ \underline{0111}.$$

*Example 7b*
In the regular Boolean case, let $f = 101X$ and let $\bar{F} = g_1 \vee g_2 \vee g_3 = \{00X0,\ 0XX1,\ 110X\}$ as shown in the Karnaugh map below.




| $K$ | $H(f; k)$ | $\overline{Z(f; k)}$ | $SPE(f; k)$ |
|---|---|---|---|
| 1 | $g_1, g_2$ | 1 | $\underline{1}0\ 1X = f$ |
| 2 | null | $X$ | $1\underline{X}1\ X$ |
| 3 | null | $X$ | $10\ \underline{X}X$ |
| 4 | null | $X$ | $10\ 1\ \underline{X} = f$ |

Notice that the Boolean case is a degenerate case where 1 or 0 in any variable can stay the same or become an $X$ when the coface operation succeeds.

*Lemma 3* Let $C$ be any cube in $F \vee DC$ which contains the cube $f$ of $S$. Then part $k$ of $C$ is covered by part $k$ of $SPE(f; k)$. That is, if $C = \mu_1\mu_2 \cdots \mu_k \cdots \mu_p$, the 1's in $\mu_k$ are a subset of the 1's in $\overline{Z(f; k)}$.

*Proof* Suppose $\mu_k$ contains a 1 that is not in $\overline{Z(f; k)}$. This implies $\mu_k \cdot Z(f; k) \neq \emptyset$, which in turn implies that there exists a cube $g$ in $\bar{F}$ which is a $k$-conjugate of $f$ and part $k$ of $g$ has a non-null intersection with $\mu_k$. Since $C$ covers $f$, and $f$ and $g$ have non-null intersection in every part but $k$, $C$ and $g$ have non-null intersection. This contradicts the hypothesis that $C$ is in $F \vee DC$. Q.E.D.

It follows from the above that part $k$ of $SPE(f; k)$ is prime in the sense that no other cube in $F \vee DC$ containing $f$ can have any more 1's in part $k$ than $SPE(f; k)$ has. We define part $k$, $\overline{Z(f; k)}$, or $SPE(f; k)$ as a *prime* part, which leads to the following observation.

*Theorem 3* A cube is prime if and only if every part of the cube is a prime part.

A cube can be expanded in every part by repeatedly applying the *SPE* as follows:

$$expand(f) = SPE(\cdots SPE(SPE(SPE(f; 1); 2); 3)\cdots; p). \quad (9)$$

To be more general, let $\sigma$ be an arbitrary permutation on the index set 1 through $p$; then

$$expand(f) = SPE(\cdots SPE(SPE(SPE(f; \sigma(1)); \sigma(2)); \sigma(3))\cdots; \sigma(p)). \quad (10)$$

The result of $expand(f)$ may not be distinct for distinct part permutations. However, the part permutation does influence the shape of the expanded cube, and each expansion defines a prime cube containing $f$ by Theorem 3.

*Example 8*
Let $f$, $\bar{F}$ and the part permutations be as follows.

$f =$ 01 1000 10 | *part permutation* : *expand*($f$)

$$\bar{F} = \begin{cases} 11\ 0100\ 11 \mid \sigma_1 = (1, 2, 3) & : 11\ 1011\ 10 : A \\ 10\ 0011\ 01 \mid \sigma_2 = (1, 3, 2) & : 11\ 1000\ 11 : B \\ 01\ 0110\ 01 \mid \sigma_3 = (3, 2, 1) & : 01\ 1001\ 11 : C \end{cases}$$

The part permutations (2, 1, 3) and (2, 3, 1) both produce $A$ and (3, 1, 2) produces $B$.

There is no guarantee of generating all prime cubes containing $f$ even if all possible part permutations are used unless, of course, $f$ happens to belong to an essential prime cube. The goal is not to generate prime cubes but rather to generate an efficient cover of the function. Therefore, a heuristic procedure is used to choose a permutation for which $expand(f)$ covers as many cubes of $S$ as possible. Consider a cube $C(f)$ defined by

$$C(f) = \overline{Z(f; 1)}\ \overline{Z(f; 2)}\cdots\overline{Z(f; P)}. \quad (11)$$

For Example 8, $C(f)$ is 11 1011 11. Obviously, $C(f)$ is not always contained in $F \vee DC$. However, any expansion of $f$ can at best eliminate those cubes of $S$ that are covered by $C(f)$ which is called the *over-expanded* cube of $f$. The permutation we choose is derived from examining the set of cubes of $S$ that are covered by $C(f)$.

Let the *super* cube $C$ of a set of cubes $T = \{C_i | i \in I\}$ be the smallest cube which contains all of the $C_i$ of $T$. We state the following lemma omitting the proof.

*Lemma 4* The super cube $C$ of $T$ is the bit-by-bit OR of all the $C_i$ of $T$.

One can readily observe that $C(f)$ is a super cube of all prime cubes that cover $f$ and is also the super cube of the set of cubes $\{SPE(f; k) | k = 1, 2, \cdots, P\}$.

For a given $f \in S$, let $f' = expand(f)$ obtained with a chosen part permutation. If $f'$ covers a subset of cubes $S'$ of $S$, $f \in S'$, the whole set $S'$ can be replaced by $f'$, which decreases the solution size. If, instead of $f'$, one uses a super cube $f''$ of $S'$ in the replacement, the reduction of the solution is not affected. The reason for using $f''$ is that $f'' \subseteq f'$, which implies that $f''$ has a higher probability of being contained in another expanded cube of $S$ than $f'$ does. Of course, $f''$ may not be a prime cube. In the next section we show how this $f''$ is further reduced to the smallest necessary size cube that can replace the $S'$.

The cube expansion process terminates when all remaining cubes of $S$ are expanded. The expansion process described above also provides an alternate definition of an essential prime cube.

*Theorem 4* The cube $expand(f)$ of a vertex $f$ is an essential prime cube (EPC) if and only if $expand(f)$ equals the over-expanded super cube $C(f)$. It follows that when $expand(f)$ is an essential prime cube, the order of part expansion is immaterial. (Proof follows from the remark after Lemma 4.)

• *Cube reduction process*

The smaller the size of a cube, the more likely that it will be covered by another expanded cube. The expansion process leaves the solution in near-prime cubes. Therefore, it is important to examine ways of reducing the size of cubes in $S$ without affecting the coverage. Define the *essential* vertices of a cube as those vertices that are in $F$ and are not covered by any other cube in $S$. Let $f'$ be the supercube of all the essential vertices of a cube $f \in S$; then $f'$ is the smallest cube contained in $f$ which can replace $f$ in $S$ without affecting the solution size. Of course, if $f$ does not contain any essential vertices, then the reduced cube is a null cube and $f$ may be removed from the $S$ list, decreasing the solution size by one. Let $S = f \vee \{S_i | i \in I\}$; then the reduced cube $f'$ can be obtained as

$$f' = \text{the super cube of } f \mathbin{\textcircled{\#}} ((\bigvee_{i \in I} S_i) \vee DC). \quad (12)$$

In Eq. (12) a regular # operation can be used in place of $\textcircled{\#}$. In fact, the irredundant cover method [5, 7, 12] uses the regular # operation between a given cube and the rest of the cubes of a solution. The purpose of this # operation in the irredundant cover method is to remove a redundant cube. In our case the reduction of the size of the given cube is the primary purpose. Regardless of the purpose, we claim that the use of $\textcircled{\#}$ facilitates this type of computation in general. The number of disjoint cubes of a cover is usually much smaller than the number of all prime cubes of the same cover, which is the product of regular # operations.

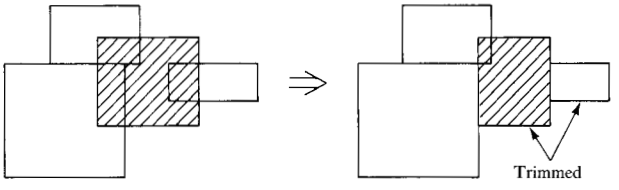
In our programs the reduced cube $f'$ is not obtained in the manner suggested by Eq. (12). Since the super cube is the desired result, a simpler tree type algorithm can be used to determine the appropriate reduction of each part of the given $f$.

The cube reduction process goes through the list of cubes in the solution $S$ in a selected order and reduces each of them. The cube ordering algorithm for the reduction step is a heuristic way to maximize the total cube size reductions; the process removes the redundant cubes and trims the remaining ones.

In the previous section, we mentioned how the replacement cube ($f''$) was found by the expansion process. The size of this cube can be further reduced along with the sizes of the remaining cubes in the solution. We do this within the cube expansion process by first reducing all the remaining cubes in the solution one at a time against the replacement cube $f''$, and then reducing the $f''$ to the smallest necessary size. This is illustrated in the following example.

*Example 9*

Let the replacement cube $f''$ (the shaded area) and some of the remaining cubes of $S$ in the periphery of $f''$ be as shown on the left below. The right side shows the desired cube shapes before the expansion process proceeds to the next cube in the solution.




The reduction of one cube $A$ against another cube $B$ assumes that $B$ does not cover $A$ and that the two differ in at least two parts. Cube $A$ can be reduced if and only if all parts of $B$ cover $A$ except in one part; let that be part $j$. Given $A = \pi_1\pi_2 \cdots \pi_j \cdots \pi_p$ and $B = \mu_1\mu_2 \cdots \mu_j \cdots \mu_p$, the trimmed $A$ becomes $A' = \pi_1\pi_2 \cdots (\pi_j\overline{\mu_j}) \cdots \pi_p$.

• *Cube reshaping process*

After the expansion and reduction steps are performed, the solution in progress contains minimal vertex sharing cubes. The nature of the cubes in $S$ is that there is no cube in $F \vee DC$ that covers more than one cube of $S$. Now we attempt to change the shapes of the cubes without changing their coverage or number. What we adopted is a very limited way of reorganizing the cube shapes, called the cube *reshaping* process. Considering that the reshapable cubes must be severely constrained, it was our surprise to see significant reshaping taking place in the course of minimization runs on large, practical functions.

The reshaping transforms a pair of cubes into another disjoint pair such that the vertex coverage is not affected. Let us assume that $S$ is the solution in progress in which no cube covers another and the distance between any two cubes is greater than or equal to two. Let $A$ and $B$ be two cubes in $S$. Then the cubes $A = \pi_1\pi_2 \cdots \pi_p$ and $B = \mu_1\mu_2 \cdots \mu_p$ in that order are said to satisfy the *reshaping condition* if and only if

1. The distance between $A$ and $B$ is exactly two.
2. One part of $A$ covers the corresponding part of $B$.

Let $i$ and $j$ be the two parts in which $A$ and $B$ differ and let $j$ be the part in which $A$ covers $B$, i.e., $\pi_j$ covers $\mu_j$; $\pi_i$ cannot cover $\mu_i$ for, if it did, then $A$ would cover $B$. The two cubes

$$A' = \pi_1\,\pi_2 \cdots \pi_i \cdots (\pi_j \wedge \overline{\mu_j}) \cdots \pi_p \tag{13}$$

and

$$B' = \pi_1\,\pi_2 \cdots (\pi_i \vee \mu_i) \cdots \mu_j \cdots \pi_p \tag{14}$$

are called the reshaped cubes of $A$ and $B$. The process is called *reshape* $(A; B)$.

*Lemma 5* The reshaped cubes $A'$ and $B'$ are disjoint and $A \vee B = A' \vee B'$.

*Proof* The $j$th part of $A'$ is $\pi_j \wedge \overline{\mu_j}$ (bit-by-bit) and the $j$th part of $B'$ is $\mu_j$; hence, $A'$ and $B'$ are disjoint. In reshaping, $A$ is split into two cubes $A'$ and $A'' = \pi_1\pi_2 \cdots (\pi_j \wedge \mu_j) \cdots \pi_p$. But $(\pi_j \wedge \mu_j) = \mu_j$ because $\pi_j$ covers $\mu_j$; thus the distance between $A''$ and $B$ is one. So $A''$ and $B$ merge into the single cube $B'$. Q.E.D.

The reshape operation between $A$ and $B$ is order dependent. If the cubes in $S$ are not trimmed, it may be possible to perform reshape in either of two ways (e.g., if $A$ and $B$ are distance two apart, $\pi_i \supset \mu_i$ and $\pi_j \subset \mu_j$). Since $A$ is split and one part is merged with another cube $B$, the natural order would be the larger cube first and the smaller cube second when checking the conditions for reshaping. After reshaping, $A'$, the remaining part of $A$, has a greater probability of merging since it has been reduced in size.

*Example 10a*

Let $S$ consist of three cubes $A$, $B$ and $C$ as follows.

$$S = \begin{cases} 11\ 0110\ 01 : A \\ 10\ 1001\ 01 : B \\ 01\ 0110\ 10 : C \end{cases}$$

$A$ and $B$ satisfy the reshaping condition to yield

$A' = 01\ 0110\ 01$ (can merge with $C$ in the next expansion step)

$B' = 10\ 1111\ 01$

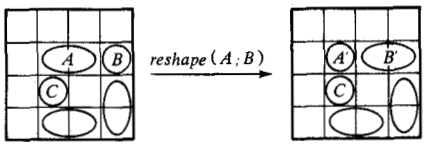
Or $A$ and $C$ may yield

$A' = 10\ 0110\ 01$ (can merge with $B$)

$C' = 01\ 0110\ 11$

*Example 10b*

Regular Boolean Karnaugh map example.




$A'$ can merge with $C$ in the next expansion step. Or $A$ and $C$ could be reshaped to $A'$ and $C'$ such that $A'$ and $B$ can merge later.

The reshaping operation can be viewed as a special case of the consensus operation. Notice that the reshaped cube $B'$ is the consensus term between $A$ and $B$. The reshaping condition holds only if the pair of cubes can be represented by a consensus cube plus another cube for the remainder of the vertices covered by $A$ and $B$. The consensus operation is used in classical minimization methods to generate prime implicants from a given implicant list of functions.

### Algorithmic description of MINI

This section describes the algorithms which implement the procedures outlined in the previous section. The algorithms are intended as a level of description of MINI which is between the theoretical considerations and a real program. They show the flow of various subprocesses and the management of many heuristics.

• *Main procedure*

M1. Accept the Boolean specification.
M2. Accept the partition description.
M3. Extract the $F$-care specification and decode into cubes according to the partition description. Assign to $F$.
M4. Generate DON'T CARE specification ($DC$) due to any inputs which appear in more than one part.
M5. Extract the original DON'T CARE specification and add to the $DC$ specification generated in M4.
M6. Decode the $DC$ specification into cubes. Assign to $DC$.
M7. Generate the partition description in the format required by subsequent programs.
M8. Let $F$ be the distance one merging of $F \vee DC$.
M9. Let $\bar{F}$ be $U$ ⊛ $F$.
M10. Let $\bar{F}$ be $\bar{F}$ expanded against $F$.
M11. Generate the disjoint $F$ by $U$ ⊛ ($\bar{F} \vee DC$).
M12. Let $F$ be $F$ expanded against $\bar{F}$ and compute the solution size.
M13. Reduce each cube of $F$ against the other cubes in $F \vee DC$.
M14. Reshape $F$.
M15. Let $F$ be $F$ expanded against $\bar{F}$ and compute the solution size.
M16. If the size of the new solution is smaller than the size of the solution immediately before the last execution of M13, go to M13. Otherwise $F$ is the solution.

*Remarks*

M3 and M6 give the $F$ and the $DC$ covers, respectively. M8 is performed for computational advantage only. M9 produces the disjoint $\bar{F}$ cover. M10 is for computational advantage. M11 produces the disjoint $F$ cover. M13–16 form the main loop which produces decreasing size solutions which contain all of the $F$ vertices and perhaps some of the $DC$ vertices. M1 through M6 are for the Boolean specified functions. If the original specification is in cube notation for $F$ and $DC$, the procedure should start at M7.

• *Preparatory algorithms*

Assume that the function is given as a Boolean specification and that the partition information is given as a permuted input list and a part size list. The sum of the numbers in the input part size list should equal the length of the permuted input list. For example, the input variable permutation (0 1 3 5 4 2 1) and the decoder sizes (2 2 3). imply that the inputs are partitioned as (0, 1), (3, 5), and (4, 2, 1). The variable number 1 is assigned to both the first and the third parts. The order of variables within a part does not influence the minimization, but it does influence the bit pattern in the part.

*Separate F and DC specifications.*

$F$ specification:

PF1. Eliminate from the original specification all rows that do not contain a 1 in the output portion.
PF2. Replace each $d$ (DON'T CARE symbol) in the output portion with a 0.

$DC$ specification:

PDC1. Select all rows of the original specification that contain at least one $d$ in the output portion.
PDC2. In the output portion, replace each 1 with a 0.
PDC3. In the output portion, replace each $d$ with a 1.

*Decode the given Boolean specification of F (the output portion now contains only 1's and 0's) into a cubical representation with the given partition.*

PD1. Construct a matrix $G$ whose $k$th column is the column of the input specification that corresponds to the $k$th variable in the permuted input list.
PD2. Perform steps PD2–6 for each input part from first to last.
PD3. Let $NF$ be the first $p$ columns of $G$, where $p$ is the number of variables in the part.
PD4. Drop the first $p$ columns of $G$.
PD5. Decode each row of $NF$ into the bit string of length $2^p$, which corresponds to the truth table entries for that row. (For example, −10 becomes 001000100.)
PD6. For the next part, go to PD3.

PD7. The output portion of the specification without change becomes the output part of the decoded cubes.

*Generate, for M4, the DC specification due to the assignment of an input to more than one part. The result has the form of the matrix G given in PD1. The output part of each of the resultant cubes contains all 1's.*

PMDC1. Start with a null $DC$.

PMDC2. Repeat steps PMDC3–7 for all variables.

PMDC3. If variable $I$ appears in $k$ parts, generate the following $2^k - 2$ rows of $DC$. The matrix $M$ gives the input parts and the output parts are all 1's.

PMDC4. Let $M$ be a matrix of −'s with dimensions $(2^k - 2)$ rows × (length of the permuted input list) columns.

PMDC5. Let $W$ be a $(2^k - 2) \times k$ matrix of 1's and 0's where the $n$th row represents the binary value of $n$. The all-0 and all-1 rows are not present.

PMDC6. The $t$th column of $W$ replaces the column of $M$ that corresponds to the $t$th occurrence of variable $I$ in the permuted input list.

PMDC7. For the next variable, go to PMDC3.

• *Distance one merging of cubes*

S1. Consider the cubes as binary numbers and reorder them in ascending (or descending) order by their binary values.

S2. The bits in part $k$ (initially $k = P$, the last part) are the least significant ones. Starting from the top cube, compare adjacent cubes. If they are the same except in part $k$, remove both cubes and replace them with the bit-by-bit OR of the two cubes. Proceed through the entire list.

S3. Reorder the cubes using only the bits in part $k$. In case of a tie, preserve the previous order.

S4. Part $k - 1$ now contains the least significant bits. Let $k$ be $k - 1$ and go to S2. Terminate when all parts have been processed.

*Remark*

If any set of cubes are distance one apart and the difference is in part $k$, the set of cubes will appear in a cluster when ordered using the bits of part $k$ as the least significant positions.

• *Disjoint sharp of a cover F against a cover G* ($F$ ⓐ $G$)

Ordering of right side argument; reorder cubes of $G$:

ORDG1. Sum the number of 1's in each part of the list $G$ and divide by the part size to obtain the average density of 1's in each part.

ORDG2. For each part, starting from the most dense to the least dense, do steps ORDG3–6.

ORDG3. Sum the number of 1's per bit position for every bit in the part. Order the bits from most 1's to least 1's.

ORDG4. Do steps ORDG5, 6 for all bits in the part in the order computed in ORDG3.

ORDG5. Reorder the cubes of $G$ such that the cubes with a 1 in the bit position appear on top of the cubes with a 0 in the bit position. Within the two sets, preserve the previous order.

ORDG6. Go to ORDG5 for the next bit of the part. If all bits in a part are done, go to ORDG3 for the next part.

ORDG7. Terminate when the last bit of the last part has been processed.

*Remarks*

The ordering procedure has been obtained from numerous experiments. The objective is to order $G$ such that the number of cubes produced by the disjoint sharp will be as small as possible. One of the properties of the above ordering is that it tends to put the larger cubes on top of the smaller cubes.

$F$ ⓐ $G$:

DSH1. Order $G$ according to ORDG.

DSH2. Remove the first cube of $G$ and assign it to the current cube ($CW$).

DSH3. Let $Z$ be the list of cubes in $F$ which are disjoint from $CW$. Remove $Z$ from $F$.

DSH4. Compute the internal part ordering for $F$ ⓐ $CW$ as follows: For each part compute the number of cubes in $G$ that are disjoint from $CW$ in that part. Order the parts such that the number of cubes that are disjoint in that part are in descending order.

DSH5. Using Eq. (4), compute $F$ ⓐ $CW$ with the part permutation given by DSH4; then add the result to the $Z$ list.

DSH6. If $G$ is empty, the process terminates and the $Z$ list is the result. If $G$ is not empty, let $F$ be the $Z$ list and go to DSH2.

*Remarks*

ORDG and DSH4 are the two heuristic ordering schemes used in the sharp process. These two heuristics were chosen so that the disjoint sharp process would produce a small number of disjoint cubes.

• *Expansion of F against G*

Ordering the cubes of $F$:

ORDF1. Sum the number of 1's in every bit position of $F$.

ORDF2. For every cube in $F$, obtain the weight of the cube as the inner product of the cube and the column sums.

ORDF3. Order the cubes in $F$ such that their weights are in ascending order.

*Remarks*

This ordering tends to place on top of the list those cubes that are hard to merge with other cubes. If a cube can expand to cover many other cubes, the cube must have 1's where many other cubes have 1's, and hence its weight is large. This heuristic ordering produces the effect of "chewing-away-from-the-edges." When there is a huge DON'T CARE space, $F \vee DC$ can be used instead of $F$ in ORDF1, for more effective expansion of cubes.

*The expansion process:*

EXP1. Order the cubes of $F$ according to ORDF.

EXP2. Process the unexpanded cubes of $F$ in order. Let $f$ be the current cube to be expanded.

EXP3. For each part $k$, compute the $k$-conjugate sets $H(f;k)$ given by Eq. (5) and their $Z(f;k)$; then form the over-expanded cube $C(f)$ given by Eq. (9).

EXP4. Let $Y$ be the set of cubes of $F$ that are covered by $C(f)$.

EXP5. For each part, compute the weight as the number of cubes in $Y$ whose part $k$ is covered by part $k$ of $f$.

EXP6. Order the parts in ascending order of their weights.

EXP7. Let $ZW$ be the expanded $f$ using the above part permutation and Eq. (8).

EXP8. Let $Y$ be all of the cubes of $F$ that are covered by $ZW$ and remove $Y$ from $F$.

EXP9. Let $S$ be the super cube of $Y$.

EXP10. Find all cubes in $F$ that are covered by $ZW$ in all parts but one. Let these cubes by $Y$.

EXP11. Reduce each cube of $Y$ against $ZW$.

EXP12. Let $T$ be the super cube of $Y$. Let $ZW$ be $ZW \wedge (S \vee T)$.

EXP13. The modified expanded $f$ is $ZW$. Append $ZW$ to the bottom of $F$.

EXP14. If there are any unexpanded cubes in $F$, go to EXP2.

*Remarks*

EXP3–6 defines the internal part permutation. The idea is to expand first those parts that, when expanded, will cover the most cubes that were not covered by the original cube. EXP8 removes all covered cubes. The $S$ of EXP9, which is contained in $ZW$, could replace $f$ now if a cube reduction were not employed. By EXP10–11, all remaining cubes of $F$ are reduced. The intersection of $T$ and $ZW$ denotes the bits of the initial expanded prime cube $ZW$ which were necessary in the reduction of any cube. The final replacement for the original cube $F$ is thus $(ZW \wedge T) \vee S = ZW \wedge (T \vee S)$. The cube that replaces $f$ is the smallest subcube of a prime cube containing $f$ that can contain and reduce the same cubes of $F$ that the prime cube can.

• *Reduction of cubes*

The actual experimental program for this algorithm is quite different from a straightforward disjoint sharp process. For efficient computation a tree method of determining essential bits of a cube is used. The algorithm given below is only a conceptual one. First the cubes to be reduced (given as $F$) are reordered according to ORDF except that ORDF3 is modified to order cubes in descending order of their weights. This ordering tends to put cubes that have many bits in common with other cubes on top of the list. It is assumed that the $DC$ list is also given.

RED1. Order the cubes of $F$ with the modified ORDF.

RED2. Do steps RED2–4 for all cubes of $F$ in order. Let the current cube be $f$.

RED3. Replace $f$ with the super cube of the disjoint sharp of $f$ against $DC \vee (F - f)$; $F - f$ denotes all the cubes of $F$ except $f$. If the super cube is a null cube, $f$ is simply removed from the list.

RED4. Go to RED2 for the next cube.

• *Reshape the cubes of F*

RESH1. Order the cubes of $F$ by the modified ORDG used in RED1.

RESH2. Do for all cubes of $F$ in order. Let the current cube be $C1$.

RESH3. Proceed through the cubes below $C1$ one at a time until a reshape occurs or until the last cube is processed. Let the current cube be $C2$.

RESH4. If $C1$ covers $C2$, remove $C2$ from $F$ and go to RESH3.

RESH5. If $C1$ and $C2$ are distance one apart, remove $C1$ and replace $C2$ with $C1$ bit-by-bit OR $C2$ and mark the ORed entry as reshaped. Go to RESH2.

RESH6. If $C1$ and $C2$ do not meet the reshaping condition, go to RESH3.

RESH7. If $C1$ and $C2$ meet the reshaping condition, form the reshaped cubes $C1'$ and $C2'$. Replace $C2$ with $C1'$ and $C1$ with $C2'$. Mark these cubes as reshaped. Go to RESH2.

RESH8. If $C1$ is not the last cube in the list, go to RESH2.

RESH9. Let all reshaped cubes be $R$ and all unchanged cubes be $T$.

RESH10. Let $C1$ range over all cubes in $R$ and $C2$ range over all cubes in $T$; then repeat RESH2–8.

*Remarks*

The reordering of RESH1 puts more "splitable" cubes at the top of the list. RESH2 and RESH3 initiate the pair-

wise comparison loop. Conditions of RESH4 or RESH5, which result in the removal of a cube, may occur as a result of the current reshape process or as a result of a previous reduce process. RESH10 gives "stubborn" cubes another chance to be reshaped.

## Discussion

• *Summary*

A general two-level logic function minimization technique, MINI, has been described. The MINI process does not generate all prime implicants nor perform the covering step required in a classical two-level minimization. Rather, the process uses a heuristic approach that obtains a near minimal solution in a manner which is efficient in both computing time and storage space.

MINI is based on the positional cube notation in which groups of inputs and the outputs form separate coordinates. Regular Boolean minimization problems are handled as a particular case. The capability of handling multiple output functions is implicit.

Given the initial specification and the partition of the variables, the process first maps or decodes all of the implicants into the cube notation. These cubes are then "exploded" into disjoint cubes which are merged, reshaped, and purged of redundancy to yield consecutively smaller solutions. The process makes rigorous many of the heuristics that one might use in minimizing with a Karnaugh map.

The main subprocesses are

1. Disjoint sharp.
2. Cube expansion.
3. Cube reduction.
4. Cube reshaping.

The expansion, reduction, and reshaping processes appear to be conceptually new and effective tools in practical minimization approaches.

• *Performance*

The MINI technique is intended for "shallow" functions, even though many "deep" functions can be minimized successfully. The class of functions which can be minimized is those whose final solutions can be expressed in a few hundred cubes. Thus, the ability to minimize a function is not dependent on the number of input variables or minterms in the function. We have successfully minimized several 30-input, 40-output functions with millions of minterms, but have failed (due to the storage limitation of an APL 64-kilobyte work space) to minimize the 16-variable EXCLUSIVE OR function which must have $2^{15}$ cubes in the final solution.

For an $n$-input, $k$-output function, define the effective number of input variables as $n + \log_2 k$. For a large class of problems, our experience with the APL program in a 64-kilobyte work space indicates that the program can handle almost all problems with 20 to 30 effective inputs. The number of minterms in the problem is not the main limiting factor.

The performance of MINI must be evalutated using two criteria. One is the minimality of the solution and the other is the computation time. Numerous problems with up to 36 effective inputs have been run; MINI obtained the actual minimum solution in most of these cases. The symmetric function of nine variables, $S_{3456}^{9}$, contains 420 minterms and 1680 prime implicants when each variable is in its own part (i.e., the regular Boolean case). The minimum two-level solution is 84 cubes. The program produced an 85-cube solution in about 20 minutes of 360/75 (APL) CPU time. The minimality of the algorithm is thus shown to be very good, considering the difficulty of minimizing symmetric functions in the classical approach, due to many branchings. A large number of very shallow test cases, generated by the method shown in [19], were successfully minimized, although a few cases resulted in one or two cubes more than the known minimum solutions.

The run time is largely dependent on the number of cubes in the final solution. This dependence results because the number of basic operations for the expand, reduce, and reshape processes is proportional to the square of the number of cubes in the list. It is difficult to compare the computation time of MINI with classical approach programs. The many-variable problems run on MINI could not be handled by the classical approach because of memory space and time limitations. For just a few input variables (say, up to eight variables), both approaches use comparable run times. However, the complexity of computation grows more or less exponentially with the number of variables in the classical minimization, even though the problem may be a shallow one. An assembly language version of MINI is now almost complete. The run time can be reduced by a factor as large as 50, requiring only a few minutes for most of the 20- to 30-effective-input problems. Thus it appears that MINI is a viable alternative to the classical approach in minimizing the practical problems with many input and output variables.

• *Minimal solutions in the classical sense*

The MINI process tries to minimize the number of cubes or implicants in the solution. The cubes in the solution may not be prime, as in the classical minimization where the cost function includes the price (number of input connections to AND and OR gates) of realizing each cube. But if such consideration becomes beneficial, a prime

cube solution *can* be obtained from the result of MINI. This is done by first applying the reduction process to the output part of each cube in the solution and then expanding all the input parts of the cubes in any arbitrary part order. The MINI solution can also be reduced to smaller cubes by putting through an additional reduction step.

• *Multiple-valued logic functions*

It was mentioned that each part of the generalized universe may be considered as a multiple-valued logical input. By placing $n_i$ Boolean variables in part $i$, we presented the MINI procedure with part lengths equal to $2^{n_i}$ except for the output part. MINI can handle a larger class of problems if the specification of a function is given directly in the cube format.

By organizing problems such as medical diagnoses, information retrieval conditions, criteria for complex decisions, etc. in multiple-value variable logic functions, one can minimize them with MINI and obtain aid in analysis. This is demonstrated with the following example.

*Example: Nim*

The game of *Nim* is played by two persons. There is an arbitrary number of piles of matches and each pile may initially contain an arbitrary number of matches. Each player alternately removes any number (greater than zero) of matches from one pile of his choice. The player who removes the last match wins the game. The strategy of the game has been completely analyzed and the player who leaves the so-called "correct position" is assured of winning the game, for the other player must return it to an incorrect position, which can then be made into a correct one by the winning player.

The problem considered contains five piles and each pile has two places for matches. Thus a pile can have no match, one match, or two matches at any phase of the game. Taking the number of matches in a pile as values of variables, we have a five-variable problem and each variable has three values (0, 1 or 2). Out of 243 ($3^5$) possible positions, 61 are correct. The 182 remaining incorrect positions were specified and minimized by the MINI program. For instance, the incorrect position (0, 1, 1, 0, 2) is specified as (100 010 010 100 001) in the generalized coordinate format. Using this result and the fact that all variables are symmetric, one can deduce the incorrect positions:

1. Exactly two piles are empty (cubes 1–10) or no pile is empty (cube 21).
2. Only one pile has two matches (cubes 11, 13, 17–19).
3. Only one pile has one match (cubes 12, 14–16, 20).

The MINI result identified the 21 cubes shown below.

| | | | | | |
|---|---|---|---|---|---|
| *1* | 011 | 100 | 011 | 100 | 011 |
| *2* | 100 | 011 | 100 | 011 | 011 |
| *3* | 100 | 011 | 011 | 100 | 011 |
| *4* | 011 | 011 | 100 | 011 | 100 |
| *5* | 011 | 100 | 011 | 011 | 100 |
| *6* | 011 | 100 | 100 | 011 | 011 |
| *7* | 011 | 011 | 011 | 100 | 100 |
| *8* | 100 | 011 | 011 | 011 | 100 |
| *9* | 011 | 011 | 100 | 100 | 011 |
| *10* | 100 | 100 | 011 | 011 | 011 |
| *11* | 001 | 110 | 110 | 110 | 110 |
| *12* | 101 | 101 | 101 | 010 | 101 |
| *13* | 110 | 110 | 110 | 001 | 110 |
| *14* | 010 | 101 | 101 | 101 | 101 |
| *15* | 101 | 010 | 101 | 101 | 101 |
| *16* | 101 | 101 | 010 | 101 | 101 |
| *17* | 110 | 001 | 110 | 110 | 110 |
| *18* | 110 | 110 | 001 | 110 | 110 |
| *19* | 110 | 110 | 110 | 110 | 001 |
| *20* | 101 | 101 | 101 | 101 | 010 |
| *21* | 011 | 011 | 011 | 011 | 011 |

• *Further comments*

In the case of the single output function $F$, the designer invariably has the option of realizing either $F$ or $\bar{F}$. The freedom to realize either is often a consequence of the availability of both the true and the complemented outputs from the final gate. However, it may also be a consequence of the acceptability of either form as input to the next level. Given the choice of the output phases for a multiple-output function, a best phase assignment would be the one that produces the smallest minimized result. Since there are $2^k$ different phase assignments for $k$ output functions, a non-exhaustive heuristic method is desired. One way to accomplish this would be to double the outputs of the function by adding the complementary phase of each output before minimization. The phases can be selected in a judicious way from the combined minimized result. This approach adds only a double-size output part in the MINI process. The combined result is just about double the given one-phase minimization. Hence, using the MINI approach a phase-assigned solution can be attained in about four times the time required to minimize the given function that has every output in true phase.

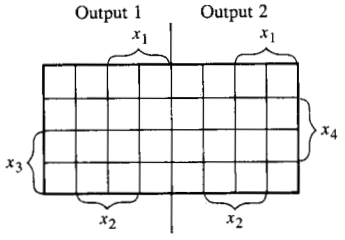
Our successful experience with the MINI process suggests both challenging theoretical problems and interesting practical programs. A theoretical characterization of functions, which either confirms or refutes the MINI heuristics, would be useful. The number of times the cube expansion process need be iterated is another matter requiring further study. Currently, we terminate the iteration if there is no improvement from the previous application of the expansion step.

## Acknowledgments

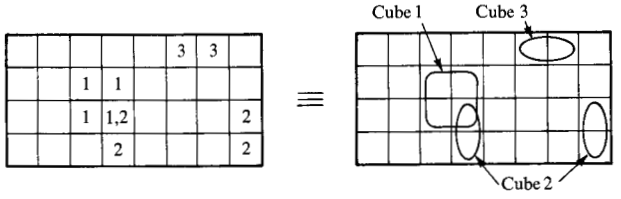
We thank Harold Fleisher and Leon I. Maissel for their constructive criticism and many useful discussions in the course of the development of MINI. James Chow has implemented most of it in assembly language with vigor and originality.

## Appendix

Here we give an example of a four-input, two-output Boolean function to illustrate the major steps of MINI. Most functions of this size get to the minimal solution by the first expansion alone. This example, however, is an exception and illustrates all the subprocesses of MINI. We use the Karnaugh map for the illustrations, rather than the cube notation. The conventions for this map are as follows:




The ordered cubes are denoted by numbers in the vertices of the cubes, as follows.




The function to be minimized and the effects of the subprocesses are shown in Fig. A1.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | 20 | 14 | 13 | | 16 | 17 | |
| 11 | | 21 | 22 | | | | 19 |
| 23 | 1 | 15 | 8 | 7 | 6 | 10 | 4 |
| 12 | 18 | | 3 | | 9 | 2 | 5 |

(a)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | 9 | 9 | 10 | | 5 | 5 | |
| 4 | | 3 | 4 | | | | 2 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 8 | 7 | | 8 | | 5 | 5 | 6 |

(b)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 5 | | | | 7 | | | 7 |
| | 2 | | | 4 | 6 | 6 | |
| | | | | | | | |
| | | 1 | | 3 | | | |

(c)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 3 | | | | 7 | | | 7 |
| | 1 | | | 5 | 4 | 4 | |
| | | | | | | | |
| | | 2 | | 6 | | | |

(d)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | 6 | 6 | 4 | | 11 | 11 | |
| 12 | | 3 | 12 | | | | 1 |
| 12 | 9 | 9 | 12 | 8 | 7 | 7 | 8 |
| 10 | 2 | | 10 | | 11 | 11 | 5 |

(e)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | 5 | 5 | 4 | | 3 | 3 | |
| 2 | | 7 | 2 | | | | 9 |
| 1,2 | 1 | 1 | 1,2 | 1 | 1 | 1 | 1 |
| 6 | 6 | | 4 | | 3 | 3 | 8 |

(f)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | 5 | 5 | 3 | | 2 | 2 | |
| 4 | | 7 | 4 | | | | 9 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 6 | 6 | | 3 | | 2 | 2 | 8 |

(g)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | 8 | 8 | 4 | | 8 | 8 | |
| 1 | | 2 | 7 | | | | 7 |
| 9 | 9 | 9 | 9 | 9 | 9 | 9 | 9 |
| 5 | 5 | | 6 | | 3 | 3 | 6 |

(h)

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | 7 | 2,7 | 2 | | 7 | 7 | |
| 1 | | 2 | 2,6 | | | | 6 |
| 8 | 8 | 8 | 8 | 8 | 8 | 8 | 8 |
| 4 | 4 | | 5 | | 3 | 3 | 5 |

**Figure A1** Example of the major steps of MINI operating on a 23-minterm list.

(a) Unit-distance-merged $F$, ordered for (#) right argument.

(b) Disjoint $\bar{F}$ ordered for expansion against $F$ of a.

(c) Expanded $\bar{F}$ ordered for (#) right argument.

(d) Disjoint $\bar{F}$ ordered for expansion against $\bar{F}$ of c; beginning of the decreasing solution.

(e) Expanded $F$ ordered for reduction.

(f) Reduced $F$ ordered for reshaping.

(g) Reshaped $F$ ordered for another expansion against $\bar{F}$.

(h) Final expanded $F$, the minimal solution of eight cubes; cubes 1, 3–6 are not prime.

## References

1. W. V. Quine, "The Problem of Simplifying Truth Functions," *Am. Math. Monthly* **59,** 521 (1952).
2. W. V. Quine, "A Way to Simplify Truth Functions," *Am. Math. Monthly* **62,** 627 (1955).
3. E. J. McCluskey, Jr., "Minimization of Boolean Functions," *Bell Syst. Tech. J.* **35,** 1417 (1956).
4. J. P. Roth, "A Calculus and An Algorithm for the Multiple-Output 2-Level Minimization Problem," Research Report RC 2007, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, February 1968.
5. E. Morreale, "Recursive Operators for Prime Implicant and Irredundant Normal Form Determination," *IEEE Trans. Comput.* **C-19,** 504 (1970).
6. J. R. Slagle, C. L. Chang and R. C. T. Lee, "A New Algorithm for Generating Prime Implicants," *IEEE Trans. Comput.* **C-19,** 304 (1970).
7. R. E. Miller, *Switching Theory, Vol. 1: Combinatorial Circuits*, John Wiley & Sons, Inc., New York, 1965.
8. E. Morreale, "Partitioned List Algorithms for Prime Implicant Determination from Canonical Forms," *IEEE Trans. Comput.* **C-16,** 611 (1967).
9. C. C. Carroll, "Fast Algorithm for Boolean Function Minimization," Project Themis Report AD680305, Auburn University, Auburn, Alabama, 1968 (for Army Missile Command, Huntsville, Alabama).
10. R. M. Bowman and E. S. McVey, "A Method for the Fast Approximation Solution of Large Prime Implicant Charts," *IEEE Trans. Comput.* **C-19,** 169 (1970).
11. E. G. Wagner, "An Axiomatic Treatment of Roth's Extraction Algorithm," Research Report RC 2205, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, September 1968.

12. Y. H. Su and D. L. Dietmeyer, "Computer Reduction of Two-Level, Multiple Output Switching Circuits," *IEEE Trans. Comput.* **C-18,** 58 (1969).
13. R. S. Michalski, "On the Quasi-Minimal Solution of the General Covering Problem," *Proceedings of the Fifth International Symposium on Information Processing (FCIP69)* **A3,** 125, 1969 (Yugoslavia).
14. R. S. Michalski and Z. Kulpa, "A System of Programs for the Synthesis of Combinatorial Switching Circuits Using the Method of Disjoint Stars," *Proceedings of International Federation of Information Processing Societies Congress 1971*, Booklet TA-2, p. 158, 1971 (Ljubljana, Yugoslavia).
15. D. L. Starner, R. O. Leighon and K. H. Hill, "A Fast Minimization Algorithm for 16 Variable Boolean Functions," submitted to *IEEE Trans. Comput.*
16. Y. H. Su and P. T. Cheung, "Computer Minimization of Multi-valued Switching Functions," *IEEE Trans. Comput.* **C-21,** 995 (1972).
17. J. P. Roth, "Theory of Cubical Complexes with Applications to Diagnosis and Algorithmic Description," Research Report RC 3675, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, January 1972.
18. S. J. Hong and D. L. Ostapko, "On Complementation of Boolean Functions," *IEEE Trans. Comput.* **C-21,** 1072 (1972).
19. D. L. Ostapko and S. J. Hong, "Generating Test Examples for Heuristic Boolean Minimization," *IBM J. Res. Develop.* **18,** 469 (1974); this issue.

*Received December 10, 1973; revised April 30, 1974*

*S. J. Hong, a member of the IBM System Products Division laboratory in Poughkeepsie, New York, is on temporary assignment at the University of Illinois, Urbana, Illinois 61801; R. G. Cain is located at the IBM System Development Division Laboratory, Poughkeepsie, New York 12602; and D. L. Ostapko, is located at the IBM System Products Division Laboratory, Poughkeepsie, New York 12602.*