ANALYSIS OF MINIMIZATION ALGORITHMS
FOR MULTIPLE-VALUED PROGRAMMABLE LOGIC ARRAYS*

Parthasarathy Tirumalai
52L/57, Hewlett-Packard Company
5301 Stevens Creek Boulevard
Santa Clara, CA 95051

Jon T. Butler
Dept. of Electr. and Comp. Eng.
Naval Postgraduate School
Monterey, CA 93943-5100

ABSTRACT

We compare the performance of three heuristic algorithms [3,6,13] for the minimization of sum-of-products expressions realized by the newly developed multiple-valued programmable logic arrays [9]. Heuristic methods are important because exact minimization is extremely time-consuming. We compare the heuristics to the exact solution, showing that heuristic methods are reasonably close to minimal. We use as a basis of comparison the average number of product terms over a set of randomly generated functions. All three heuristics produce nearly the same average number of product terms. Although the averages are close, there is surprisingly little overlap among the set of functions where the best realization is achieved. Thus, there is a benefit to applying different heuristics and then choosing the best realization.

I. INTRODUCTION

The minimization of sum-of-products expressions in binary logic has received considerable attention for over 30 years. The complexity of the problem has been known for almost as long. Although minimal sum-of-products extraction appears to be a special case of the general minimal set covering problem, it is not. This was proven by Gimpel [7] in 1965. That is, any instance of the set covering problem is an instance of the minimal sum-of-products extraction. In 1972, Karp [8] showed that the set covering problem is NP-complete; thus, so also is minimal sum-of-products extraction. The best known algorithm then requires exponential time.

This is a real barrier; it precludes the exact minimization of functions with even a moderately low number of inputs, e.g. 20. As a result, considerable effort has been devoted to heuristic minimization methods. For example, among the Berkeley VLSI tools is ESPRESSO-IIIC [4], a C program that minimizes binary functions by a set of operations on the prime implicants.

Recently, there has been considerable interest in multiple-valued PLA’s [1,2,3,5,6,9,10,11,13,14,15]. Several have been proposed [11,14,15] and at least one implemented [9]. We know of three heuristic MVL sum-of-products minimization algorithms. Pomper and Armstrong [13] introduced in 1981 a heuristic method that found a near-minimal sum-of-products expression as a direct cover of the function. The algorithm proceeds in two steps: 1. select a minterm and 2. find an implicant that covers the minterm. The first step is accomplished by a random choice of minterm, while the second step is accomplished by choosing the largest implicant. In 1986, Besslich [3] introduced another direct cover method that seeks to cover the “most isolated” minterms first. And in 1987, Dueck and Miller [6] introduced a method which also seeks the most isolated minterm first, but chooses a product term that tends to introduce the fewest discontinuities when subtracted from the function.

There has been little study of the relative merits of heuristic algorithms. To the credit of Brayton, Hachtel, McMullen and Sangiovanni-Vincentelli [4], the realizations produced by ESPRESSO-IIIC were compared with the realizations of MINI, PRESTO, and POP [4] over a set

*Research supported in part by NSF Grant MIP-8706553, in part by NATO Grant 423/84, and in part by an NFS Foundation Grant.

of 56 specifically chosen binary functions. However, the justification of the newly introduced algorithms examined by us has rested on an intuitive notion, supported by examples. In this paper, we analyze six synthesis methods.

1. Random minterm random implicant
2. Pomper and Armstrong [13]
5. Gold
6. Absolute minimization

over a set of 7000 randomly chosen 2-variable 4-valued functions. Our choice of 2-variable functions is influenced by a need to compare heuristics (1-5) with absolute minimal realizations (6). Even with only 2 inputs, absolute minimization requires high computation times, on the order of days. Our choice of 4-values is influenced by the current interest in this radix. The three heuristic algorithms in [3, 6, 13] apply to sum-of-products expressions where sum is MAX. In all three algorithms, the case where sum is truncated SUM is also considered. Because the PLA's produced in CCD [9] use the SUM operation, our analysis uses this operation. Further, we have adapted the heuristics to the case of product terms consisting of interval literals, again since the CCD PLA's realize such functions. A motivation for the study reported here is the development of a minimization method for our CCD PLA CAD tool [10].

This paper is organized as follows. The next section introduces notation and fundamental concepts. Section III describes the six synthesis methods, and Section IV describes the results of a comparative analysis of their performance. In the final section, we summarize the results.

II. BACKGROUND AND NOTATION

Let \( X = \{x_1, x_2, \ldots, x_n\} \) be a set of \( n \) variables, where \( x_i \) takes on values from \( R = \{0, 1, \ldots, r-1\} \). A function \( f(X) \) is a mapping \( f:R^n \rightarrow R \cup \{r\} \), where \( r \) is the don't care value. Specifically, \( f(X) \) is said to be an \( n \)-variable \( r \)-valued function. Fig. 1 below shows a map representation of a 2-variable 5-valued function. Blank entries correspond to 0. An assignment \( z \) of values to variables in \( X \) is called a minterm if \( f(z) \neq 0 \). In Fig. 1, there are 12 minterms all of which yield a 2.

Functions realized by the PLA's described in Kerkhoff and Butler [9] are composed of three functions,

1. MIN: \( f(x_1, x_2) = x_1x_2 (= \text{MIN}(x_1, x_2)) \),
2. SUM: \( f(x_1, x_2) = x_1 + x_2 (= x_1 + x_2 \text{ if } x_1 + x_2 \leq 3 \text{ and } = 3 \text{ otherwise, where } + \text{ is viewed as ordinary addition and } x_i \text{ is viewed as an integer}) \), and
3. literal: \( f(x_1) = a x_1^a (= r-1 \text{ if } a \leq x_1 \leq b \text{ and } = 0, \text{ otherwise}) \).

In binary, the SUM, MIN, and literal correspond to AND, OR, and \( z^* \), where \( z^* \in \{x, \overline{x}\} \). In the realization of functions by a multiple-valued PLA, constants and literals occur as operands of the MIN functions. A product term is the MIN of one nonzero constant and one or more literals. For example, \( f(x_1, x_2) = 2^2 x_1^4 x_2^3 \) is a product term that is 2 when \( x_1 = 2, 3, \text{ or } 4 \) and \( x_2 = 1 \). An implicant of function \( f(X) \) is a product term \( I(X) \) such that \( f(X) \leq I(X) \). A prime implicant of \( f(X) \) is an implicant of \( f(X) \) such that there is no other implicant \( I'(X) \) of \( f(X) \) where \( I'(X) \geq I(X) \). For example, \( 2^2 x_1^4 x_2^3 \) is an implicant of the function in Fig. 1 but it is not a prime implicant. However, \( 2^2 x_1^4 x_2^3 \) is a prime implicant. Any function can be expressed as the SUM of implicants [15]. For example, the function \( f(x_1, x_2) \) in Fig. 1 can be expressed as the SUM of 4 implicants,

\[
2^2 x_1^4 x_2^3 + 2^3 x_1^3 x_2^3 + 2^0 x_1^3 x_2^3 + 2^1 x_1^4 x_2^3. \quad (1)
\]

We use the term sum-of-products to describe functions realized by such PLA's, where sum...
refers to SUM and product refers to MIN. A
sum-of-products expression for function \( f(X) \) is
minimal if there is no other expression for \( f(X) \)
with fewer product terms. For example, (1) is a
minimal sum-of-products expression. Given \( f(X) \),
implicant \( I(X) \) covers a minterm at \( x \) if
\( f(x) = I(x) \). Therefore, \( g(x) = f(x) - I(x) \)
has the property \( g(x) = 0 \), and we say that sub-
tracting \( I(X) \) drives the minterm at \( x \) to 0.

It is interesting to note that the none of the
implicants in (1) are prime. In fact, any sum-of-
products expression for \( f(x_1,x_2) \) containing at
least one prime implicant is not minimal. Such
expressions require at least 5 product terms.
Therefore, in contrast to the case of conventional
binary sum-of-products, it is necessary to con-
sider all implicants. A similar observation was
made with respect to sum-of-products expres-
sions, where sum is MAX and products are the
MIN of unary functions of varying costs [12].

III. MINIMIZATION ALGORITHMS FOR
SUM PLA’S

We consider six synthesis methods. The first
five are heuristic and are based on the direct
cover approach. In this approach, a minterm is
first chosen. Next, an implicant is chosen that
covers this minterm. The implicant is then
subtracted from the function and the process is
repeated until there are no more minterms.

1. Random Minterm Random Implicant

In this heuristic, both the minterm and impli-
cant are chosen randomly, with all choices equally
likely. That is, among all minterms, one is
chosen randomly, and similarly in the case of
implicants. As with the other algorithms, the
chosen implicant is subtracted and the process
repeated on the resulting function. Since no par-
ticular characteristics of the function are used to
determine the choice of minterm or implicant, it
provides a basis of comparison for the next four
heuristics, which use characteristics of the func-
tions to limit, to varying degrees, the choices of
minterms and implicants.

2. Pomper and Armstrong

As in the Random Minterm Random Impli-
cant heuristic, this heuristic chooses the minterm
randomly. However, the implicant is chosen as
the one, which when subtracted, drives the large-
est number of minterms to 0 or don’t care. If
there is more than one such implicant, the largest
is chosen. If there are more than one largest, the
one generated first is chosen. The process is
repeated until the function is completely covered.
A formal description of this algorithm is given in
Appendix I.

Because the Random Minterm Random Implic-
ant heuristic can choose from all possible covers
of any function, there is a nonzero probability
that it will find a minimal realization. However,
if the minimal realizations are a small percentage
of the total number of realizations, non-minimal
realizations will most likely result. Our analysis
shows this clearly. There is then the question of
whether the Pomper and Armstrong heuristic has
a nonzero probability of finding a minimal reali-
zaion for all functions. The example in Fig. 1 is
a function where this probability is 0. Here, a
prime implicant is never in a minimal sum-of-
products realization, and so, for this case, the
Pomper and Armstrong heuristic does not pro-
duce a minimal realization.

3. Besslich

Besslich [3] presents a direct cover approach
for the heuristic minimization of multiple-valued
logic functions using minterm weighting and
implicant detecting transformations. In this
heuristic, each minterm is assigned a weight that
is a measure of the degree to which other min-
terms cluster around it. Minterms in the center
of clusters have the highest weight and isolated
minterms the lowest. The minterm with the
smallest weight is chosen. Next all implicants
that cover the chosen minterm are generated. For
each, an efficiency factor equal to the cost of the
implicant divided by the number of minterms it
covers is calculated and the minterm with the
largest factor is chosen. In a PLA, all implicants
are realized with the same cost (one column).
Therefore, for this case, only the number of min-
terms driven to 0 or don’t care determines which
implicant is chosen. The basic idea of the
Besslich heuristic is to cover the most isolated
minterms first and use implicants that have a low
cost per minterm covered. Besslich [3] does not
mention of how ties are broken. We choose to
break ties among implicants having the same
efficiency factor by using only the largest implic-
ants, and among these by choosing the first one
generated. Thus, the choice of implicants is made
in the same way as in the Pomper and Armstrong heuristic. A formal description of the algorithm is given in Appendix II.

4. Dueck and Miller

Dueck and Miller [6] present a heuristic similar to that of Besslich's with the intent of improving realizations using the SUM operation. In this approach, an isolation factor (IF) is calculated for each minterm with the smallest value (that is, all 1 minterms are considered first; if there are none, then 2 minterms are considered, etc.). The isolation factor of a minterm at \( z \) is inversely proportional to 1 plus the number of adjacent minterms plus the number of logic variables where there are a nonzero number of such minterms. Similar to the weight transform presented by Besslich, the isolation factor provides a measure of the degree to which a specific minterm can combine with other minterms in the function (the correlation is negative, however). The minterm with the highest isolation factor is chosen. All implicants that cover this minterm are then generated. For each of these, a parameter called the relative break count (RBC) is calculated. This provides a measure of the degree to which the function is simplified if the implicant under consideration is chosen. The idea is to judiciously choose implicants that make the remaining function easy to realize. Our use of interval literals requires an adjustment to the Dueck and Miller algorithm. A formal description of our algorithm, modified to the specifications of our problem, is given in Appendix III.

5. Gold

Gold is a heuristic in which the heuristic algorithms of Pomper and Armstrong, Besslich, and Dueck and Miller are applied and the best realization is chosen. It was inspired by the observation that these algorithms displayed a diversity in realizations. That is, no single algorithm is consistently better than the others over all functions; there are classes of functions where one algorithm does better than the others.

6. Absolute Minimization

An algorithm which produces the exact minimal sum-of-products was devised to compare with the results produced by the previous four algorithms. This does essentially an exhaustive search over all possible solutions, starting with the fewest number of product terms. The algorithm requires considerable computer time. A complete description of the algorithm is given in Section VI.

7. Summary

Table I below summarizes the six algorithms.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Choice of Minterm</th>
<th>Choice of Implicant</th>
</tr>
</thead>
<tbody>
<tr>
<td>2. Pomper and Armstrong [13]</td>
<td>Random</td>
<td>Drives most min. to 0</td>
</tr>
<tr>
<td>3. Besslich [3]</td>
<td>Smallest weight</td>
<td>Drives most min. to 0</td>
</tr>
<tr>
<td>5. Gold</td>
<td>Best of 2, 3, and 4</td>
<td>Exhaustive search</td>
</tr>
<tr>
<td>6. Absolute Minimization</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table I. Summary of Minimization Algorithms
IV. COMPARISON OF ALGORITHM PERFORMANCE

For the purpose of comparison, we separate all four-valued, two-variable functions into 17 disjoint classes according to the number of non-zero values in the function. For 14 of the 17 classes, 500 random functions are generated. For each function, the six algorithms are applied and the number of product terms derived. For functions with less than three non-zero values, the average number of product terms can be calculated as follows.

**0 non-zero values:** There is only one (trivial) function, constant zero, in this class. This function requires no implicants in its cover, and all algorithms produce identical results.

**1 non-zero value:** There are 48 functions in this class, and each requires uniquely one implicant. Thus, the average number of implicants required by all algorithms is 1.

**2 non-zero values:** There are \( 3^2 \cdot \binom{16}{2} = 1080 \) such functions. All require two implicants unless there are two adjacent minterms with the same value, in which case a single implicant is sufficient. Of the 1080 functions, 72 can be covered by one implicant. Only the Random Minterm Random Implicant algorithm will fail to find the minimal cover for all functions in this class. Specifically, functions with adjacent identical minterms will be incorrectly minimized half the time, since a cover of a single minterm or two adjacent minterms are equally likely. For a uniformly distributed set of functions, the average and standard deviation on the number of implicants produced by the Random Minterm Random Implicant algorithm are 1.97 and 0.18, respectively. On the other hand, these values for all other algorithms are 1.93 and 0.25, respectively. Our random function generation program was tested on this class and yielded values very close to these, to within 0.18% for the average and to within 3.9% for the standard deviation.

For classes with a larger number of non-zero values, 500 functions were randomly generated. The functions generated had no don't care values. However, they could develop don't care values during the covering process. (Nota bene: A value of 3 in the original function becomes a don't care when fully covered.) Fig. 2 shows the results.

![Graph showing the average number of product terms versus the number of non-zero values for the six algorithms.](image)

Figure 2. The Average Number of Product Terms Versus the Number of Nonzero Values for the Six Algorithms.
It can be seen that the Random Minterm Random Implicant algorithm does quite poorly. Choosing the largest (prime) implicant, as in the Pomper and Armstrong algorithm, provides a significant improvement. There is very little difference between the performance of the Besslich and the Dueck and Miller algorithms; their curves in Fig. 2 are almost identical. Both algorithms choose the most isolated minterm first, and this gives some improvement over that of the Pomper and Armstrong algorithm. Gold, taking advantage of the small overlap between the three heuristic algorithms, provides some further improvement. Still further improvement is achieved by absolute minimization, shown as the lowest curve in the figure.

Another measure of comparison of the algorithms is the number of functions for which the absolute minimal solution is found. Fig. 3 shows how the six algorithms compare on this basis. Here there is a much larger distinction between the algorithms. As the number of non-zero values increases, the number of functions for which an optimum cover is found by the Random Minterm Random Implicant algorithm drops off sharply, to nearly zero. The Pomper and Armstrong algorithm offers considerable improvement, but even in this case only about 39% of the functions are completely minimized when the number of non-zero values is 14-16. The Besslich and Dueck and Miller algorithms are again very close to each other. Approximately, 52% of the functions are minimized for these two algorithms when the number of non-zero values is 14-16. Gold shows a reasonable improvement, minimizing about 75% of the functions with 14-16 non-zero values.

Tables II and III show the numerical values obtained from the programs. Another run was made using different seeds for the random number generator streams. The results from both runs were very similar, indicating that a sufficiently large sample set size was used. An examination of the data suggests the truth of the following.

Conjecture: No more than 10 implicants are needed in a minimal sum-of-products expression of a four-valued, two-variable function, where sum is the SUM operation and the products consist of the MIN of a constant and interval literals on the variables.

The absolute minimization program and the
<table>
<thead>
<tr>
<th>Number of Nonzero Values</th>
<th>Random Minterm Random Implicant</th>
<th>Pomper Armstrong</th>
<th>Besslich</th>
<th>Dueck Miller</th>
<th>Gold</th>
<th>Absolute Minimiza.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>1</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>1.97</td>
<td>1.93</td>
<td>1.93</td>
<td>1.93</td>
<td>1.93</td>
<td>1.93</td>
</tr>
<tr>
<td>3</td>
<td>2.90</td>
<td>2.79</td>
<td>2.78</td>
<td>2.78</td>
<td>2.78</td>
<td>2.78</td>
</tr>
<tr>
<td>4</td>
<td>3.80</td>
<td>3.58</td>
<td>3.58</td>
<td>3.57</td>
<td>3.57</td>
<td>3.57</td>
</tr>
<tr>
<td>5</td>
<td>4.66</td>
<td>4.35</td>
<td>4.34</td>
<td>4.31</td>
<td>4.31</td>
<td>4.31</td>
</tr>
<tr>
<td>6</td>
<td>5.53</td>
<td>5.04</td>
<td>5.02</td>
<td>5.00</td>
<td>4.96</td>
<td>4.96</td>
</tr>
<tr>
<td>7</td>
<td>6.31</td>
<td>5.66</td>
<td>5.63</td>
<td>5.57</td>
<td>5.53</td>
<td>5.52</td>
</tr>
<tr>
<td>8</td>
<td>7.04</td>
<td>6.13</td>
<td>6.07</td>
<td>6.03</td>
<td>5.95</td>
<td>5.94</td>
</tr>
<tr>
<td>9</td>
<td>7.77</td>
<td>6.67</td>
<td>6.59</td>
<td>6.55</td>
<td>6.44</td>
<td>6.40</td>
</tr>
<tr>
<td>10</td>
<td>8.50</td>
<td>7.06</td>
<td>6.97</td>
<td>6.91</td>
<td>6.75</td>
<td>6.69</td>
</tr>
<tr>
<td>11</td>
<td>8.99</td>
<td>7.34</td>
<td>7.25</td>
<td>7.19</td>
<td>6.98</td>
<td>6.92</td>
</tr>
<tr>
<td>12</td>
<td>9.78</td>
<td>7.77</td>
<td>7.56</td>
<td>7.57</td>
<td>7.32</td>
<td>7.20</td>
</tr>
<tr>
<td>13</td>
<td>10.31</td>
<td>7.90</td>
<td>7.63</td>
<td>7.60</td>
<td>7.31</td>
<td>7.17</td>
</tr>
<tr>
<td>14</td>
<td>10.54</td>
<td>7.91</td>
<td>7.62</td>
<td>7.60</td>
<td>7.30</td>
<td>7.08</td>
</tr>
<tr>
<td>15</td>
<td>11.00</td>
<td>7.81</td>
<td>7.64</td>
<td>7.63</td>
<td>7.26</td>
<td>6.97</td>
</tr>
<tr>
<td>16</td>
<td>11.21</td>
<td>7.55</td>
<td>7.33</td>
<td>7.24</td>
<td>6.99</td>
<td>6.71</td>
</tr>
</tbody>
</table>

Table II. Average Number of Product Terms

<table>
<thead>
<tr>
<th>Number of Nonzero Values</th>
<th>Random Minterm Random Implicant</th>
<th>Pomper Armstrong</th>
<th>Besslich</th>
<th>Dueck Miller</th>
<th>Gold</th>
<th>Absolute Minimiza.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>500</td>
<td>500</td>
<td>500</td>
<td>500</td>
<td>500</td>
<td>500</td>
</tr>
<tr>
<td>1</td>
<td>500</td>
<td>500</td>
<td>500</td>
<td>500</td>
<td>500</td>
<td>500</td>
</tr>
<tr>
<td>2</td>
<td>483</td>
<td>500</td>
<td>500</td>
<td>500</td>
<td>500</td>
<td>500</td>
</tr>
<tr>
<td>3</td>
<td>439</td>
<td>494</td>
<td>499</td>
<td>499</td>
<td>500</td>
<td>500</td>
</tr>
<tr>
<td>4</td>
<td>384</td>
<td>493</td>
<td>492</td>
<td>497</td>
<td>499</td>
<td>500</td>
</tr>
<tr>
<td>5</td>
<td>331</td>
<td>477</td>
<td>483</td>
<td>486</td>
<td>497</td>
<td>500</td>
</tr>
<tr>
<td>6</td>
<td>252</td>
<td>460</td>
<td>474</td>
<td>483</td>
<td>500</td>
<td>500</td>
</tr>
<tr>
<td>7</td>
<td>187</td>
<td>438</td>
<td>450</td>
<td>477</td>
<td>496</td>
<td>500</td>
</tr>
<tr>
<td>8</td>
<td>113</td>
<td>405</td>
<td>438</td>
<td>455</td>
<td>493</td>
<td>500</td>
</tr>
<tr>
<td>9</td>
<td>76</td>
<td>375</td>
<td>414</td>
<td>430</td>
<td>481</td>
<td>500</td>
</tr>
<tr>
<td>10</td>
<td>43</td>
<td>334</td>
<td>371</td>
<td>399</td>
<td>470</td>
<td>500</td>
</tr>
<tr>
<td>11</td>
<td>23</td>
<td>313</td>
<td>349</td>
<td>374</td>
<td>469</td>
<td>500</td>
</tr>
<tr>
<td>12</td>
<td>12</td>
<td>268</td>
<td>337</td>
<td>338</td>
<td>441</td>
<td>500</td>
</tr>
<tr>
<td>13</td>
<td>7</td>
<td>231</td>
<td>304</td>
<td>308</td>
<td>430</td>
<td>500</td>
</tr>
<tr>
<td>14</td>
<td>3</td>
<td>191</td>
<td>269</td>
<td>284</td>
<td>394</td>
<td>500</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
<td>197</td>
<td>240</td>
<td>235</td>
<td>363</td>
<td>500</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>202</td>
<td>253</td>
<td>285</td>
<td>365</td>
<td>500</td>
</tr>
</tbody>
</table>

Table III. Number of Functions Where Absolute Minimi. is Achieved.
random function generation routines were written in C. The heuristic algorithms were written in Pascal. The program was executed on a Hewlett-Packard Series 9000 Model 350 workstation running HP-UX (HP's extension of the UNIX operating system). The program took more than 3 days to minimize 7000 randomly generated functions with 3 to 16 non-zero values.

V. ABSOLUTE MINIMIZATION.

The program to compute the absolute minimal realization of a function is a search over combinations of implicants. As observed in Section II, it is necessary to consider all implicants and not just the prime implicants. In the worst case for two-variable four-valued functions \( f(x_1, x_2) = 3 \), this can be large, 300 implicants. Consider a function which has 100 implicants and requires 8 product terms in a minimal cover. Using a brute force enumeration technique, the proof that this is the minimum, requires that all combinations of seven or fewer implicants be examined. There are more than 17 billion such combinations, and assuming that each combination can be generated and examined in 200 microseconds (a value that is better than in our program), it would take more than 944 hours, or 39 days, to examine all the combinations for one function! Clearly this is not acceptable. However, using the lemma given below, the search tree can be significantly reduced.

**Definition:** \( U(z) \) is a *constrained implicant set* of minterm \( z \) for function \( f(X) \), if \( U(z) = \bigcup_{I(z)>0} I(X) \), where \( I(X) \) is an implicant of \( f(X) \).

**Lemma 1:** If \( U(z) \) is a constrained implicant set, then every minimal sum-of-products expression of \( f(X) \) contains at least one implicant from \( U(z) \).

The proof follows from the fact that at least one implicant is needed from each of the \( k \) disjoint sets.

**Definition:** \( U(z) \) is a *minimal* constrained implicant set of a function if and only if \( 0 < |U(z)| \leq |U(y)| \) for all \( y \in X \), where \( U(z) \) is a constrained implicant set.

A minimal constrained implicant set is useful in an efficient search for the absolute minimal sum-of-products expression for a function. The search space can be represented by a tree where each node corresponds to a function, and each edge to an implicant. The root node represents the input function, and all other nodes represent functions obtained by subtracting from the input function, the implicants in the path (from this node) to the root. If a function has \( \phi \) implicants, the root has \( \phi \) descendents. Further, each of these has at most (and probably less) \( \phi - 1 \) descendents. However, rather than consider all implicants at the root node, it is sufficient to consider only implicants from a minimal constrained implicant set, considerably reducing the search space. From Lemma 1, at least one of this set must be in a minimal sum-of-products expression. For example, the average number of implicants for a four-valued, two-variable function with one zero, (obtained from a randomly generated sample set of 500 functions) is 111.2, while the average size of the minimal constrained implicant set is only 7.8. This is a considerable reduction. In fact, without it the computation of the absolute minimum algorithms would be impossible.

The algorithm recursively finds the absolute minimum cover of the function at each node. Let \( G(f) \) denote the minimum number of implicants needed in the minimal sum-of-products expression of function \( f(X) \). Let \( U(z) = \{I_1(X), I_2(X), \ldots, I_k(X)\} \) be any constrained implicant set of \( f(X) \). Let \( g_i(X) = f(X) - I_i(X) \) be the function obtained by subtracting \( I_i(X) \) from \( f(X) \). From the definition of a constrained implicant set, it follows that, \( G(f) = 1 + \min \{G(g_1), G(g_2), \ldots, G(g_k)\} \).

**ABSOLUTE MINIMIZATION ALGORITHM**

```
f ← the input function;
cur_best_soln_size ← M {see below};
cur_soln_size ← 0;
if f has no coverable minterms, then
    output 0 and stop;
recursively_minimize(f);
output cur_best_soln_size;
stop;
```

```
procedure recursively_minimize(f);
U ← some constrained implicant set of f;
while (there exists another implicant in U)
    AND (cur_soln_size + 1) ≥ (cur_best_soln_size) do begin
    I ← the next implicant in U;
```

233
add I to the current solution;  
cur_soln_size ← cur_soln_size + 1;  
g ← f - I;  
if g has no coverable minterms then begin  
make current set the current best set;  
cur_best_soln_size ← cur_soln_size;  
end;  
else if ((cur_soln_size + 1) < cur_best_soln_size) then begin  
recursively_minimize(g);  
delete I from the current solution;  
cur_soln_size ← cur_soln_size - 1;  
end;

M represents the starting value for the solution size and is chosen as the best of the Pomper and Armstrong, Besslich, and Dueck and Miller solutions.

The speed of the absolute minimization algorithm depends on the choice of the constrained implicant set. When \( U(x) \) is small, there are fewer choices for an implicant to cover \( x \), and thus less time is required. This suggests that all possible sets be scanned and the smallest chosen. Although this reduces the total number of implicants scanned, the extra time required at each node may increase the overall program execution time. A heuristic was used to decide whether to spend this extra time or not. In our implementation, \( U(x) \) was first generated by taking a coverable minterm in the function. An improvement was attempted only if the number of implicants in this set was greater than some threshold \( T \). Values of 6 and 9 were tried with the former yielding a faster program.

VI. CONCLUDING REMARKS

There are three surprising results from this study. First, none of the three heuristic sum-of-products minimization algorithms \([3,6,13]\) is clearly superior to the others. Second, Gold, an algorithm that picks the best of the three, does significantly better than any particular one. This indicates that the nearly identical average case performance is due to a diversity of performance on individual functions. Third, the computation times for absolute minimization far exceeded our expectations. Thus what began as an investigation of heuristic methods ended as an investigation of methods to reduce the search required by absolute minimization.

REFERENCES

APPENDIX I

POMPER AND ARMSTRONG [13] ALGORITHM USING INTERVAL LITERALS

Orig_f is the given r-valued n-variable function. f is an intermediate function as it is covered in successive stages of the algorithm. Each minterm in f can have a value in the range [0...r] where r is the radix. A value of 0 was zero in orig_f or was in the range [1...r-2] in orig_f but has now been fully covered,

1,2,...,(r-1) still needs to be covered,

r (don't care) was r in orig_f, or was (r-1) in orig_f but has now been fully covered.

Algorithm:

1. f ← orig_f
2. If f has no minterms in the range [1...(r-1)] then STOP; else choose one, α_min, randomly.
3. From among all implicants that cover α_min, choose the maximal implicant, I.
4. Include I in the solution.
5. f ← f - I
6. Go to step 2.

Choosing the maximal implicant

An implicant can be written as,

$I = \prod_{1 \leq m \leq n} (j_m - i_m + 1)\

$An implicant J is 'larger' than implicant I if the size of J is larger than the size of I. The maximal implicant is chosen as follows:

```
cur_size ← -∞
cur_terms_covered ← -∞
for each implicant I of f that completely covers α_min, do begin
    L_terms_covered ← the number of minterms in f that would be driven to 0 or r (don't care) if I is subtracted from f;
    L_size ← size of implicant I;
    if (L_terms_covered > cur_terms_covered) OR ((L_terms_covered = cur_terms_covered) AND (L_size > cur_size)) then begin
        best_implicant ← I;
        cur_size ← L_size;
        cur_terms_covered ← L_terms_covered;
    end;
end;
```

APPENDIX II

BESSLICH [3] ALGORITHM USING INTERVAL LITERALS

Algorithm:

Each location in orig_f corresponds to an assignment of values in the range [0...(r-1)] to the variables z_1, z_2, ..., z_n. The concatenation of these values is taken as an index which represents this location.

1. f ← orig_f
2. If f has no terms in the range [1...(r-1)], then STOP; else find the most isolated minterm α_min in f.
3. From among all implicants that cover α_min, choose the maximal implicant, I.
4. Include I in the solution.
5. f ← f - I
6. Go to step 2.

Finding the most isolated minterm

Code the function values by mapping all 0's to -1; all r's to 0; and all other values to 1. Let cfn be the coded form of the function f.
Take the weight transform [3] of cfn and pick the minterm with the smallest weight as follows:

\[ \text{cur\_lowest\_wt} \leftarrow \infty \]

for each term \( \alpha \) in cfn such that \( \text{cfn}(\alpha) = 1 \) do begin

\[
\begin{align*}
\text{wt}(\alpha) & \leftarrow \sum_{\beta \in \text{cfn}} [\text{cfn}(\beta) \cdot \text{wt}(\alpha, \beta)] \\
\text{if} \ (\text{wt}(\alpha) < \text{cur\_lowest\_wt}) & \text{ then } \alpha_{\text{min}} \leftarrow \alpha;
\end{align*}
\]

end;

On exit from the for loop, \( \alpha_{\text{min}} \) is chosen as the most isolated minterm. In the loop, \( \text{w}(\alpha, \beta) \) is computed as follows.

Choosing the maximal implicant - Same as with Pomper and Armstrong

APPENDIX III

DUECK AND MILLER [6] ALGORITHM USING INTERVAL LITERALS

Algorithm:

1. \( f \leftarrow \text{orig} \cdot f \)
2. If \( f \) has no terms in the range \([1... (r-1)]\), then STOP; else find the most isolated minterm \( \alpha_{\text{min}} \) in \( f \).
3. From among all implicants that cover \( \alpha_{\text{min}} \), choose the maximal implicant, I.
4. Include I in the solution.
5. \( f \leftarrow f - I \); if \( f \geq 1 \); otherwise, I \( \leftarrow r \).
6. Go to step 2.

Finding the most isolated minterm

Let \( \text{min\_val} \) be the minimum value of all terms in \( f \) that are in the range \([1... (r-1)]\), i.e., neither 0 nor don't care. For each term \( \alpha \) in the function with value equal to \( \text{min\_val} \), compute the clustering factor (CF) as follows -

\[
\text{CF}(\alpha) = \text{DEA}_{\alpha} \cdot \alpha(r-1) + \text{EA}_{\alpha},
\]

where \( \text{EA}_{\alpha} \) is the number of minterms with which \( \alpha \) can be combined in an interval literal and \( \text{DEA}_{\alpha} \) is the number of variables (directions) in which \( \alpha \) can be combined with a nonzero number of minterms.

Dueck and Miller actually define the isolation factor (IF) to be the reciprocal of the clustering factor above and pick the minterm with the highest IF. The same result can be obtained by picking the minterm with the smallest CF, avoiding the division. Ties are broken arbitrarily. Let \( \alpha_{\text{min}} \) be the minterm thus chosen.

Choosing the maximal implicant

\[
\text{cur\_rbc} \leftarrow \infty
\]

for every valid implicant I in \( f \) that completely covers \( \alpha_{\text{min}} \) do begin

\[
\text{rbc} \leftarrow \text{calculated\_rbc\_of\_I\_in\_f};
\]

if \( \text{rbc} < \text{cur\_rbc} \) then begin

\[
\text{best\_implicant} \leftarrow I;
\]

\[
\text{cur\_rbc} \leftarrow \text{rbc};
\]

end;

end;

Calculation of \( \text{rbc} \)

\[
\text{rbc} \leftarrow 0;
\]

for each \( \alpha \) in \( I \) such that \( |\alpha| \neq r \) do begin

along each dimension \( i = 1 \) to \( n \) do begin

(\( \text{if } |\alpha| \leq |I| \) OR (if the neighbor \( \beta \) immediately before \( \alpha \) in this dimension (if such \( \beta \) exists) is not in \( I \) and is such that \( |\beta| = |\alpha| \) - 1) OR (if the neighbor \( \beta \) immediately after \( \alpha \) in this dimension (if such \( \beta \) exists) is not in \( I \) and is such that \( |\beta| = |\alpha| - 1 \)) then

\[
\text{rbc} \leftarrow \text{rbc} - 1;
\]

(\( \text{if the neighbor } \beta \) immediately before \( \alpha \) in this dimension (if such \( \beta \) exists) is not in \( I \) and is such that \( |\beta| = |\alpha| \) OR if the neighbor \( \beta \) immediately after \( \alpha \) in this dimension (if such \( \beta \) exists) is not in \( I \) and is such that \( |\beta| = |\alpha| \)) then

\[
\text{rbc} \leftarrow \text{rbc} + 1;
\]

end;

end;