Pmin: Programmed Minimization Methods

Obviously, logic minimization can be a very involved process. In real logic-design applications you are likely to encounter only two kinds of minimization problems: functions of a few variables that you can "eyeball" using the methods of the previous section, and more complex, multiple-output functions that are hopeless without the use of a minimization program.

We know that minimization can be performed visually for functions of a few variables using the Karnaugh-map method. We'll show in this section that the same operations can be performed for functions of an arbitrarily large number of variables (at least in principle) using a tabular method called the *Quine-McCluskey algorithm*. Like all algorithms, the Quine-McCluskey algorithm can be translated into a computer program. And like the map method, the algorithm has two steps: (a) finding all prime implicants of the function, and (b) selecting a minimal set of prime implicants that covers the function.

Quine-McCluskey algorithm

The Quine-McCluskey algorithm is often described in terms of handwritten tables and manual check-off procedures. However, since no one ever uses these procedures manually, it's more appropriate for us to discuss the algorithm in terms of data structures and functions in a high-level programming language. The goal of this section is to give you an appreciation for computational complexity involved in a large minimization problem. We consider only fully specified, single-output functions; don't-cares and multiple-output functions can be handled by fairly straightforward modifications to the single-output algorithms, as discussed in the References.

Pmin.1 Representation of Product Terms

The starting point for the Quine-McCluskey minimization algorithm is the truth table or, equivalently, the minterm list of a function. If the function is specified differently, it must first be converted into this form. For example, an arbitrary n-variable logic expression can be multiplied out (perhaps using DeMorgan's theorem along the way) to obtain a sum-of-products expression. Once we have a sum-of-products expression, each p-variable product term produces 2^{n-p} minterms in the minterm list.

We showed in Section 4.1.6 that a minterm of an n-variable logic function can be represented by an n-bit integer (the minterm number), where each bit indicates whether the corresponding variable is complemented or uncomplemented. However, a minimization algorithm must also deal with product terms that are not minterms, where some variables do not appear at all. Thus, we must represent three possibilities for each variable in a general product term:

1	Uncomplemented.
0	Complemented.
x	Doesn't appear.

These possibilities are represented by a string of *n* of the above digits in the *cube representation* of a product term. For example, if we are working with product terms of up to eight variables, X7, X6, …, X1, X0, we can write the following product terms and their cube representations:

cube representation

$$\begin{aligned} \mathsf{X7}' \cdot \mathsf{X6} \cdot \mathsf{X5} \cdot \mathsf{X4}' \cdot \mathsf{X3} \cdot \mathsf{X2} \cdot \mathsf{X1} \cdot \mathsf{X0}' &\equiv 01101110 \\ \mathsf{X3} \cdot \mathsf{X2} \cdot \mathsf{X1} \cdot \mathsf{X0}' &\equiv \text{xxxx1110} \\ \mathsf{X7} \cdot \mathsf{X5}' \cdot \mathsf{X4} \cdot \mathsf{X3} \cdot \mathsf{X2}' \cdot \mathsf{X1} &\equiv \text{1x01101x} \\ \mathsf{X6} &\equiv \text{x1xxxxxx} \end{aligned}$$

Notice that for convenience, we named the variables just like the bit positions in *n*-bit binary integers.

In terms of the *n*-cube and *m*-subcube nomenclature of Section 2.14, the string 1x01101x represents a 2-subcube of an 8-cube, and the string 01101110 represents a 0-subcube of an 8-cube. However, in the minimization literature, the maximum dimension *n* of a cube or subcube is usually implicit, and an *m*-subcube is simply called an “*m*-cube” or a “cube” for short; we’ll follow this practice in this section.

To represent a product term in a computer program, we can use a data structure with *n* elements, each of which has three possible values. In C, we might make the following declarations:

```
typedef enum {complemented, uncomplemented, doesntappear} TRIT;
typedef TRIT[16] CUBE;  /* Represents a single product
                           term with up to 16 variables */
```

However, these declarations do not lead to a particularly efficient internal representation of cubes. As we’ll see, cubes are easier to manipulate using conventional computer instructions if an *n*-variable product term is represented by two *n*-bit computer words, as suggested by the following declarations:

```
#define MAX_VARS 16     /* Max # of variables in a product term */
typedef unsigned short WORD;   /* Use 16-bit words */
struct cube {
  WORD t;  /* Bits 1 for uncomplemented variables. */
  WORD f;  /* Bits 1 for complemented variables.   */
};
typedef struct cube CUBE;
CUBE P1, P2, P3;    /* Allocate three cubes for use by program. */
```

Here, a WORD is a 16-bit integer, and a 16-variable product term is represented by a record with two WORDs, as shown in Figure Pmin-1(a) on the next page. The first word in a CUBE has a 1 for each variable in the product term that appears uncomplemented (or “true,” t), and the second has a 1 for each variable that appears complemented (or “false,” f). If a particular bit position has 0s in both WORDs, then the corresponding variable does not appear, while the case

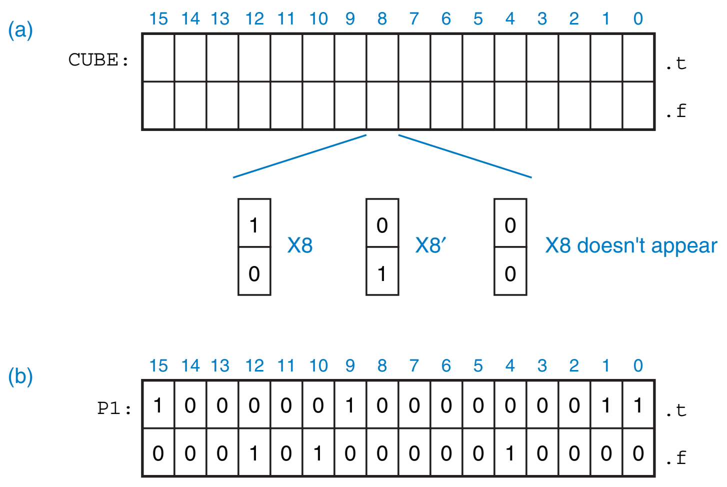


Figure Pmin-1 Internal representation of 16-variable product terms in a C program: (a) general format; (b) P1 = X15·X12′·X10′·X9·X4′·X1·X0

of a particular bit position having 1s in both `WORD`s is not used. Using this scheme, the program variable `P1` in Figure Pmin-1(b) represents the product term P1 = X15 · X12′ · X10′ · X9 · X4′ · X1 · X0. If we wished to represent a logic function F of up to 16 variables, containing up to 100 product terms, we could declare an array of 100 `CUBE`s:

```
CUBE F[100];      /* Storage for a logic function
                     with up to 100 product terms. */
```

Using this cube representation, it is possible to write short, efficient C functions that manipulate product terms in useful ways. Table Pmin-1 shows

Figure Pmin-2 Cube manipulations: (a) determining whether two cubes are combinable using theorem T10, term · X + term · X′ = term; (b) combining cubes using theorem T10.

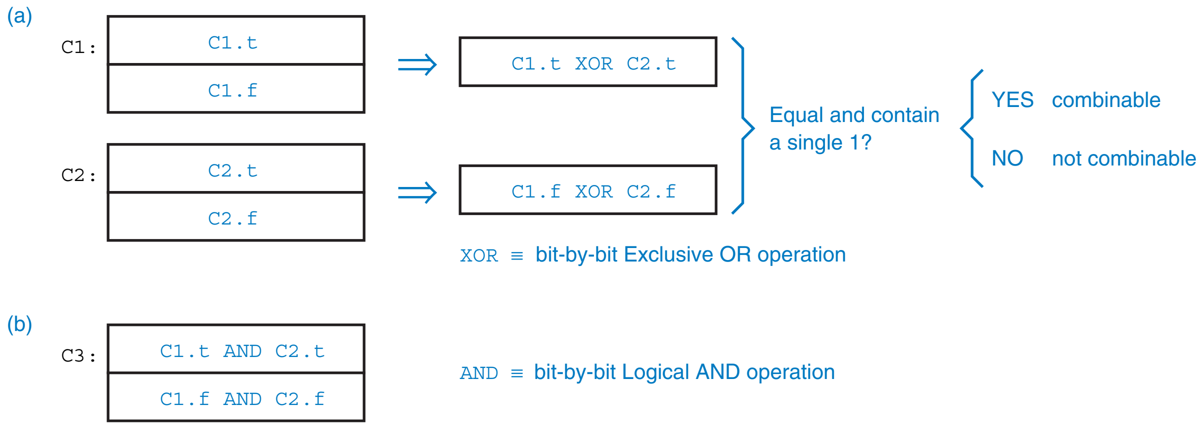


Table Pmin-1 Cube comparing and combining functions used in minimization program.

```
int EqualCubes(CUBE C1, CUBE C2)       /* Returns true if C1 and C2 are identical.  */
{
  return ( (C1.t == C2.t) && (C1.f == C2.f) );
}

int Oneone(WORD w)           /* Returns true if w has exactly one 1 bit.         */
{                            /* Optimizing the speed of this routine is critical   */
  int ones, b;               /*   and is left as an exercise for the hacker.       */
  ones = 0;
  for (b=0; b<MAX_VARS; b++) {
    if (w & 1) ones++;
    w = w>>1;
  }
  return((ones==1));
}

int Combinable(CUBE C1, CUBE C2)
{                          /* Returns true if C1 and C2 differ in only one variable, */
  WORD twordt, twordf;     /* which appears true in one and false in the other.      */

  twordt = C1.t ^ C2.t;
  twordf = C1.f ^ C2.f;
  return( (twordt==twordf) && Oneone(twordt) );
}

void Combine(CUBE C1, CUBE C2, CUBE *C3)
                         /* Combines C1 and C2 using theorem T10, and stores the   */
{                        /*   result in C3.  Assumes Combinable(C1,C2) is true.     */
  C3->t = C1.t & C2.t;
  C3->f = C1.f & C2.f;
}
```

several such functions. Corresponding to two of the functions, Figure Pmin-2 depicts how two cubes can be compared and combined if possible using theorem T10, $\mathrm{term} \cdot \mathrm{X} + \mathrm{term} \cdot \mathrm{X}' = \mathrm{term}$. This theorem says that two product terms can be combined if they differ in only one variable that appears complemented in one term and uncomplemented in the other. Combining two m-cubes yields an $(m+1)$-cube. Using cube representation, we can apply the combining theorem to a few examples:

$$\begin{aligned}
010 + 000 &= 0\text{x}0 \\
00111001 + 00111000 &= 0011100\text{x} \\
101\text{xx}0\text{x}0 + 101\text{xx}1\text{x}0 &= 101\text{xxxx}0 \\
\text{x}111\text{xx}00110\text{x}000\text{x} + \text{x}111\text{xx}00010\text{x}000\text{x} &= \text{x}111\text{xx}00\text{x}10\text{x}000\text{x}
\end{aligned}$$

Pmin.2 Finding Prime Implicants by Combining Product Terms

The first step in the Quine-McCluskey algorithm is to determine all of the prime implicants of the logic function. With a Karnaugh map, we do this visually by identifying “largest possible rectangular sets of 1s.” In the algorithm, this is done by systematic, repeated application of theorem T10 to combine minterms, then 1-cubes, 2-cubes, and so on, creating the largest possible cubes (smallest possible product terms) that cover only 1s of the function.

The C program in Table Pmin-2 on the net page applies the algorithm to functions with up to 16 variables. It uses two-dimensional arrays, `cubes[m][j]` and `covered[m][j]`, to keep track of `MAX_VARS` *m*-cubes. The 0-cubes (minterms) are supplied by the user. Starting with the 0-cubes, the program examines every pair of cubes at each level and combines them when possible into cubes at the next level. Cubes that are combined into a next-level cube are marked as “covered”; cubes that are not covered are prime implicants.

Even though the program in Table Pmin-2 is short, an experienced programmer could become very pessimistic just looking at its structure. The inner `for` loop is nested four levels deep, and the number of times it might be executed is of the order of `MAX_VARS` · $\texttt{MAX\_CUBES}^3$. That’s right, that’s an exponent, not a footnote! We picked the value `maxCubes` = 1000 somewhat arbitrarily (in fact, too optimistically for many functions), but if you believe this number, then the inner loop can be executed *billions and billions* of times.

The maximum number of minterms of an *n*-variable function is 2^n, of course, and so by all rights the program in Table Pmin-2 should declare `maxCubes` to be 2^{16}, at least to handle the maximum possible number of 0-cubes. Such a declaration would not be overly pessimistic. If an *n*-variable function has a product term equal to a single variable, then 2^{n-1} minterms are in fact needed to cover that product term.

For larger cubes, the situation is actually worse. The number of possible *m*-subcubes of an *n*-cube is $\binom{n}{m} \times 2^{n-m}$, where the binomial coefficient $\binom{n}{m}$ is the number of ways to choose *m* variables to be x’s, and 2^{n-m} is the number of ways to assign 0s and 1s to the remaining variables. For 16-variable functions, the worst case occurs with $m = 5$; there are 8,945,664 possible 5-subcubes of a 16-cube. The total number of distinct *m*-subcubes of an *n*-cube, over all values of *m*, is 3^n. So a general minimization program might require a *lot* more memory than we’ve allocated in Table Pmin-2.

There are a few things that we can do to optimize the storage space and execution time required in Table Pmin-2 (see Exercises Pmin.3–Pmin.6), but they are piddling compared to the overwhelming combinatorial complexity of the problem. Thus, even with today’s fast computers and huge memories, direct application of the Quine-McCluskey algorithm for generating prime implicants is generally limited to functions with only a few variables (fewer than 15–20).

Table Pmin-2 A C program that finds prime implicants using the Quine-McCluskey algorithm.

```
#define TRUE    1
#define FALSE   0
#define MAX_CUBES 50

void main()
{
  CUBE cubes[MAX_VARS+1][MAX_CUBES];
  int covered[MAX_VARS+1][MAX_CUBES];
  int numCubes[MAX_VARS+1];
  int m;          /* Value of m in an m-cube, i.e., ‘‘level m.’’ */
  int j, k, p;    /* Indices into the cubes or covered array.    */
  CUBE tempCube;
  int found;

  /* Initialize number of m-cubes at each level m. */
  for (m=0; m<MAX_VARS+1; m++) numCubes[m] = 0;

  /* Read a list of minterms (0-cubes) supplied by the user, storing them    */
  /* in the cubes[0,j] subarray, setting covered[0,j] to false for each      */
  /* minterm, and setting numCubes[0] to the total number of minterms read.  */
ReadMinterms;

  for (m=0; m<MAX_VARS; m++)            /* Do for all levels except the last */
    for (j=0; j<numCubes[m]; j++)       /* Do for all cubes at this level    */
      for (k=j+1; k<numCubes[m]; k++)   /* Do for other cubes at this level  */
        if (Combinable(cubes[m][j], cubes[m][k])) {
          /* Mark the cubes as covered. */
          covered[m][j] = TRUE;  covered[m][k] = TRUE;
          /* Combine into an (m+1)-cube, store in tempCube. */
          Combine(cubes[m][j], cubes[m][k], &tempCube);
          found = FALSE;  /* See if we've generated this one before. */
          for (p=0; p<numCubes[m+1]; p++)
            if (EqualCubes(cubes[m+1][p],tempCube)) found = TRUE;
          if (!found) {  /* Add the new cube to the next level. */
            numCubes[m+1] = numCubes[m+1] + 1;
            cubes[m+1][numCubes[m+1]-1] = tempCube;
            covered[m+1][numCubes[m+1]-1] = FALSE;
          }
        }
  for (m=0; m<MAX_VARS; m++)       /* Do for all levels                 */
    for (j=0; j<numCubes[m]; j++)  /* Do for all cubes at this level */
      /* Print uncovered cubes -- these are the prime implicants. */
      if (!covered[m][j]) PrintCube(cubes[m][j]);
}
```

Pmin.3 Finding a Minimal Cover Using a Prime-Implicant Table

The second step in minimizing a combinational logic function, once we have a list of all its prime implicants, is to select a minimal subset of them to cover all the 1s of the function. The Quine-McCluskey algorithm uses a two-dimensional array called a *prime-implicant table* to do this. Figure Pmin-3(a) shows a small but representative prime-implicant table, corresponding to the Karnaugh-map minimization problem of Figure 4-36. There is one column for each minterm of the function, and one row for each prime implicant. Each entry is a bit that is 1 if and only if the prime implicant for that row covers the minterm for that column (shown in the figure as a check).

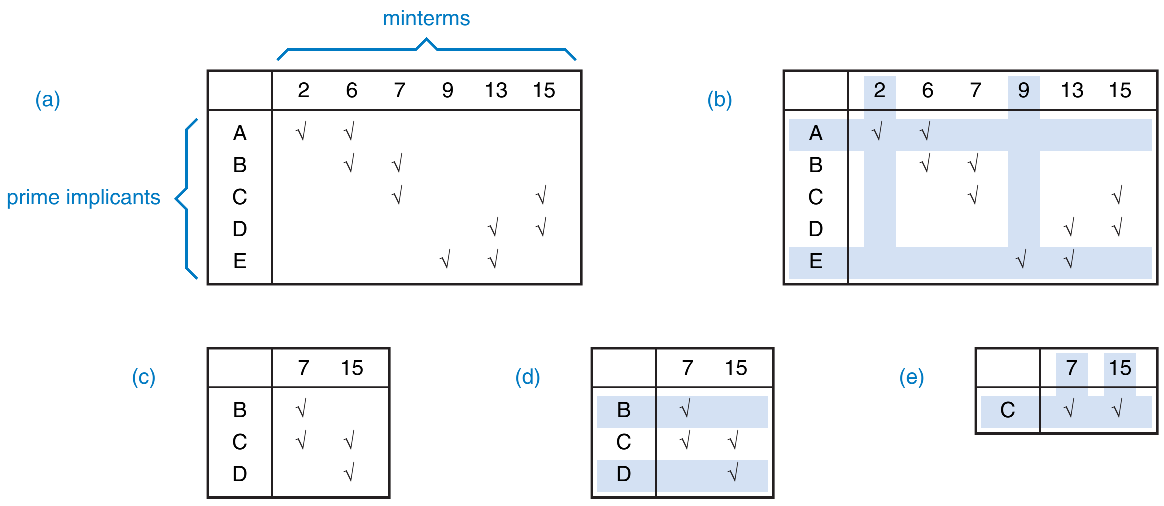
prime-implicant table

The steps for selecting prime implicants with the table are analogous to the steps that we used in Section 4.3.5 with Karnaugh maps:

1. Identify distinguished 1-cells. These are easily identified in the table as columns with a single 1, as shown in Figure Pmin-3(b).
2. Include all essential prime implicants in the minimal sum. A row that contains a check in one or more distinguished-1-cell columns corresponds to an essential prime implicant.
3. Remove from consideration the essential prime implicants and the 1-cells (minterms) that they cover. In the table, this is done by deleting the corresponding rows and columns, marked in color in Figure Pmin-3(b). If any rows have no checks remaining, they are also deleted; the corresponding prime implicants are *redundant*, that is, completely covered by essential prime implicants. This step leaves the reduced table shown in (c).

redundant prime implicant

Figure Pmin-3 Prime-implicant tables: (a) original table; (b) showing distinguished 1-cells and essential prime implicants; (c) after removal of essential prime implicants; (d) showing eclipsed rows; (e) after removal of eclipsed rows, showing secondary essential prime implicant.


minterms
prime implicants

(a)

	2	6	7	9	13	15
A	√	√				
B		√	√			
C			√			√
D					√	√
E				√	√	

(b)

	2	6	7	9	13	15
A	√	√				
B		√	√			
C			√			√
D					√	√
E				√	√	

(c)

	7	15
B	√	
C	√	√
D		√

(d)

	7	15
B	√	
C	√	√
D		√

(e)

	7	15
C	√	√

4. Remove from consideration any prime implicants that are "eclipsed" by others with equal or lesser cost. In the table, this is done by deleting any rows whose checked columns are a proper subset of another row's, and deleting all but one of a set of rows with identical checked columns. This is shown in color in (d) and leads to the further reduced table in (e).

   When a function is realized in a PLD, all of its prime implicants may be considered to have equal cost, because all of the AND gates in a PLD have all of the inputs available. Otherwise, the prime implicants must be sorted and selected according to the number of AND-gate inputs.
5. Identify distinguished 1-cells and include all secondary essential prime implicants in the minimal sum. As before, any row that contains a check in one or more distinguished-1-cell columns corresponds to a secondary essential prime implicant.
6. If all remaining columns are covered by the secondary essential prime implicants, as in (e), we're done. Otherwise, if any secondary essential prime implicants were found in the previous step, we go back to step 3 and iterate. Otherwise, the branching method must be used, as described in Section 4.3.5. This involves picking rows one at a time, treating them as if they were essential, and recursing (and cursing) on steps 3–6.

Although a prime-implicant table allows a fairly straightforward prime-implicant selection algorithm, the data structure required in a corresponding computer program is huge, since it requires on the order of $p \cdot 2^n$ bits, where p is the number of prime implicants and n is the number of input bits (assuming that the given function produces a 1 output for most input combinations). Worse, executing the steps that we so blithely described in a few sentences above requires a huge amount of computation.

References

The first algorithm for minimizing logic equations was described by W. V. Quine in "A Way to Simplify Truth Functions" (*Am. Math. Monthly*, Vol. 62, No. 9, 1955, pp. 627–631) and modified by E. J. McCluskey in "Minimization of Boolean Functions" (*Bell Sys. Tech. J.*, Vol. 35, No. 5, November 1956, pp. 1417–1444). The Quine-McCluskey algorithm is fully described in McCluskey's *Introduction to the Theory of Switching Circuits* (McGraw-Hill, 1965) and *Logic Design Principles* (Prentice Hall, 1986).

The huge number of prime implicants in some logic functions makes it impractical or impossible deterministically to find them all or select a minimal cover. However, efficient heuristic methods can find solutions that are close to minimal. The Espresso-II method is described in *Logic Minimization Algorithms for VLSI Synthesis* by R. K. Brayton, C. McMullen, G. D. Hachtel, and A. Sangiovanni-Vincentelli (Kluwer Academic Publishers, 1984). The slightly more recent Espresso-MV and Espresso-EXACT algorithms are described in "Multiple-Valued Minimization for PLA Optimization" by R. L. Rudell and A. Sangiovanni-Vincentelli (*IEEE Trans. CAD*, Vol. CAD-6, No. 5, 1987, pp. 727–750).

McCluskey's 1965 book also covers the iterative consensus algorithm for finding prime implicants and proves that it works. The starting point for this algorithm is a sum-of-products expression, or equivalently, a list of cubes. The product terms *need not* be minterms or prime implicants, but *may* be either or anything in between. In other words, the cubes in the list may have any and all dimensions, from 0 to n in an n-variable function. Starting with the list of cubes, the algorithm generates a list of all the prime-implicant cubes of the function, without ever having to generate a full minterm list.

The iterative consensus algorithm was first published by T. H. Mott, Jr., in "Determination of the Irredundant Normal Forms of a Truth Function by Iterated Consensus of the Prime Implicants" (*IRE Trans. Electron. Computers*, Vol. EC-9, No. 2, 1960, pp. 245–252). A generalized consensus algorithm was published by Pierre Tison in "Generalization of Consensus Theory and Application to the Minimization of Boolean Functions" (*IEEE Trans. Electron. Computers*, Vol. EC-16, No. 4, 1967, pp. 446–456). All of these algorithms are described by Thomas Downs in *Logic Design with Pascal* (Van Nostrand Reinhold, 1988).

Exercises

Pmin.1 There are $2n$ m-subcubes of an n-cube for the value $m = n - 1$. Show their text representations and the corresponding product terms. (You may use ellipses as required, e.g., 1, 2, …, n.)

Pmin.2 There is just one m-subcube of an n-cube for the value $m = n$; its text representation is xx…xx. Write the product term corresponding to this cube.

Pmin.3 The C program in Table Pmin-2 uses memory inefficiently because it allocates memory for a maximum number of cubes at each level, even if this maximum is never used. Redesign the program so that the `cubes` and `used` arrays are one-dimensional arrays, and each level uses only as many array entries as needed. (*Hint:* You can still allocate cubes sequentially, but keep track of the starting point in the array for each level.)

Pmin.4 As a function of m, how many times is each distinct m-cube rediscovered in Table Pmin-2, only to be found in the inner loop and thrown away? Suggest some ways to eliminate this inefficiency.

Pmin.5 The third `for`-loop in Table Pmin-2 tries to combine all m-cubes at a given level with all other m-cubes at that level. In fact, only m-cubes with x's in the same positions can be combined, so it is possible to reduce the number of loop iterations by using a more sophisticated data structure. Design a data structure that segregates the cubes at a given level according to the position of their x's, and determine the maximum size required for various elements of the data structure. Rewrite Table Pmin-2 accordingly.

Pmin.6 Estimate whether the savings in inner-loop iterations achieved in Exercise Pmin.5 outweighs the overhead of maintaining a more complex data structure. Try to make reasonable assumptions about how cubes are distributed at each level, and indicate how your results are affected by these assumptions.

Pmin.7 Optimize the `Oneone` function in Table Pmin-1. An obvious optimization is to drop out of the loop early, but other optimizations exist that eliminate the `for` loop entirely. One is based on table look-up and another uses a tricky computation involving complementing, Exclusive ORing, and addition.

Pmin.8 Extend the C program in Table Pmin-2 to handle don't-care conditions. Provide another data structure, `dc[MAX_VARS+1][MAX_CUBES]`, that indicates whether a given cube contains only don't-cares, and update it as cubes are read and generated.