Minimizing Boolean Sum of Products Functions

David Eigen

Table of Contents

Digital design is the design of hardware for computers. At the lowest level, computers are made up of switches and wires. Switches normally have two states and can only be in one state at a time. They may be manually controlled or may be controlled by the outputs of other parts of the computer. Wires also have two states, each corresponding to the level of voltage in the wire. Although wires may conduct any level of voltage, in digital design we restrict the amount of states to two: low voltage and high voltage.

In combinational design, one is given a truth table and must realize it into hardware. Depending on what the application of the hardware is, it may be optimal to have the hardware return the answer as quickly as possible (minimizing delay) or to use as little hardware as possible (minimizing cost). There are many techniques for doing this. Among them are Karnaugh Maps, the Quine-McCluskey algorithms, Simplify, and Espresso. In this paper, I will explore a few of these techniques, and compare and contrast them.

All of the techniques are derived from boolean algebra. Boolean algebra is a way of manipulating boolean variables. A boolean variable has exactly two states, just as the switches and wires at the lowest level of a computer have two states. Although the states of wires in a computer are always called HI and LO, the states of a boolean variable may be called true and false, on and off, or anything else. Usually, we use the states “1” and “0” because they are the easiest to work with.

A boolean function is a function that takes boolean parameters (inputs) and returns a boolean output. Primitive functions take one or two inputs. A two-input truth table has four lines: one line for each combination of 1s and 0s that can be assigned to the two inputs. There are four because for each 1 or 0 for the first input, the second input can be either 1 or 0. Therefore, there are 2*2 = 4 combinations. For an N-input truth table, the first input can be either 1 or 0; the second input can be either 1 or 0 for each 1 or 0 in the first input; the third input can be either 1 or 0 for any of the 2*2 different combinations of the first two inputs, doubling the number of combinations to 2*2*2; the fourth input can be either 1 or 0 for each of the different combinations of the first three inputs, doubling the number of combinations to 2*2*2*2, and so on. By the time we get to the *N*th input, there are 2^N combinations. Thus, an N-input truth table has 2^N lines.

There are sixteen different 2-input functions. This is because a 2-input truth table has 4 lines. The first line can correspond to an output of 1 or 0. For each of these possibilities, the second line can either be a 1 or a 0, making 2*2 combinations. The third line can correspond to an output of either 1 or 0 for each of the 2*2 combinations of the first two lines, making 2*2*2 possibilities. The fourth line can correspond to an output of either 1 or 0 for each of the 2*2*2 combinations of the first three lines, making 2*2*2*2 = 16 possibilities. Extending this to an *L*-line truth table, there are 2^L different possibilities. And since for an *N*-input truth table there are 2^N lines, $L = 2^N$. Therefore, an *N*-input truth table can have 2^{2^N} different functions.

Of the sixteen functions for a 2-input truth table, only seven are commonly used. I will now examine these seven crucial functions.

The invert (NOT) function takes one input, and its output is the opposite state of the input. Thus, NOT 1 = 0, and NOT 0 = 1. The truth table of the invert function is the following:

A	NOT A
0	1
1	0

NOT A is also called the complement of A, which is denoted $\overline{A}$.

The AND function's output is 1 when both of its two inputs are one. Thus, A AND B is 1 only when A and B are both one. The truth table of the AND function is the following:

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

Since anything times 0 equals 0, and 1 times 1 equals 1, A AND B looks like a multiplication table for A and B. Thus, A AND B is denoted as the product of A and B, A·B, or AB.

The OR function's output is 1 when any of its two inputs are 1. Thus, A OR B is 1 whenever A is 1 or B is 1. The truth table of the OR function is the following:

A	B	A OR B
0	0	0
0	1	1
1	0	1
1	1	1

Since 1 + 0 = 1 and 0 + 0 = 0, the truth table for A OR B looks somewhat like an addition table for A and B. Thus, A OR B is denoted as the sum of A and B, A+B.

The exclusive OR (XOR) function is 1 whenever either, but not both, of its two inputs is 1. Thus, A XOR B is 1 when only one of either A or B is 1. The truth table of the XOR function is the following:

A	B	A XOR B
0	0	0
0	1	1
1	0	1
1	1	0

A XOR B is denoted as $A \oplus B$. Note that $A \oplus B = (A + B) \cdot \overline{(A \cdot B)}$.

The other three primitive functions, NAND, NOR, and XNOR, are the complements of AND, OR , and XOR, respectively. A NAND B = $\overline{A \cdot B}$, A NOR B = $\overline{A + B}$, and A XNOR B = $\overline{A \oplus B}$.

There are four properties for boolean algebra. Often, they are called axioms. However, given the definitions of AND, OR, and NOT, they can be proved by making a truth table for each of them (this technique is called perfect induction). They are:

- Closure: $A + B \in \{0,1\}$ and $A \cdot B \in \{0,1\}$
- Associativity: $(A + B) + C = A + (B + C)$ and $(A \cdot B) \cdot C = A \cdot (B \cdot C)$
- Commutativity: $A + B = B + A$ and $A \cdot B = B \cdot A$
- Distributivity: $A + (B \cdot C) = (A + B) \cdot (A + C)$ and $A \cdot (B + C) = (A \cdot B) + (A \cdot C)$

There are also four basic identities. They are:

- $A + \overline{A} = 1$ and $A \cdot \overline{A} = 0$
- $A + 1 = 1$ and $A \cdot 0 = 0$
- $A + 0 = A$ and $A \cdot 1 = A$
- $A + A = A$ and $A \cdot A = A$

Note that each of the axioms and identities have two forms: each form can be created by switching the ORs and ANDs, and the 1s and 0s of the other form, called its “dual.” These identities can also be proved by perfect induction. For example, the truth tables and proof for the first identities are:

A	$\overline{A}$	$A + \overline{A}$	$A \cdot \overline{A}$
0	1	1	0
1	0	1	0

With these axioms and identities, we can prove the following theorems:

Absorption: $X + XY = X$ and its dual, $X(X+Y) = X$

Adsorption: $X + \overline{X}Y = X + Y$ and its dual, $X\left(\overline{X} + Y\right) = XY$, and

Adjacency: $XY + \overline{X}Y = Y$ and its dual, $(X+Y)\left(\overline{X} + Y\right) = Y$

Proof of Absorption:

By distribution, $X + XY = X(1+Y)$. 1 + Y = 1. Therefore, $X(1+Y) = X \cdot 1 = X$ (because X·1 = X). Therefore, by transitivity, $X + XY = X$.

By the distributive property, $X(X+Y) = XX + XY = X + XY$ (because X·X = X). By the form of absorption just proved, $X + XY = X$. Therefore, by transitivity, $X(X+Y) = X$.

Proof of Adsorption:

By distribution, $X + \overline{X}Y = \left(X + \overline{X}\right)(X+Y)$. Because $X + \overline{X} = 1$,

$\left(X + \overline{X}\right)(X+Y) = 1 \cdot (X+Y) = X + Y$ (because 1·X = X). Therefore, by transitivity, $X + \overline{X}Y = X + Y$.

By distribution, $X\left(\overline{X} + Y\right) = X\,\overline{X} + XY = 0 + XY$ (because $X\,\overline{X} = 0$). Because $0 + A = A$, $0 + XY = XY$. Therefore, by transitivity, $X\left(\overline{X} + Y\right) = XY$.[i]

Proof of Adjacency:

By the distributive property, $XY + \overline{X}Y = \left(X + \overline{X}\right)Y = 1 \cdot Y$ (because $X + \overline{X} = 1$). Because 1·A = A, $1 \cdot Y = Y$. Therefore, by transitivity, $XY + \overline{X}Y = Y$.

By the distributive property, $(X+Y)\left(\overline{X} + Y\right) = (X\,\overline{X}) + Y = 0 + Y$ (because $X\,\overline{X} = 0$). Because 0+A = A, $0 + Y = Y$. Therefore, by transitivity, $(X+Y)\left(\overline{X} + Y\right) = Y$.

De Morgan’s Laws state:

$\overline{A+B+C+\ldots}=\overline{A}\cdot\overline{B}\cdot\overline{C}\cdot\ldots$ and $\overline{A\cdot B\cdot C\cdot\ldots}=\overline{A}+\overline{B}+\overline{C}+\ldots$

Proof of De Morgan’s Laws:

Let $G = X + F$ and $H=\overline{X}\cdot\overline{F}$ where F is some boolean function. Then $G\cdot H=(X+F)\cdot(\overline{X}\cdot\overline{F})$. By distribution and commutativity, this equals $X\overline{X}\overline{F}+F\overline{F}\overline{X}$. And because $A\cdot\overline{A}=0$, this equals $0\cdot\overline{F}+0\cdot\overline{X}=0$. Also, $G+H=(X+F)+(\overline{X}\cdot\overline{F})$. By distribution and commutativity, this equals $(X+\overline{X}+F)\cdot(X+F+\overline{F})$. And because $A+\overline{A}=1$, this equals $(1+F)\cdot(X+1)=1$. Assume $G\neq\overline{H}$. Then there exists some ordered n-tuplet of the variables such that $G\cdot H=1$ or $G+H=0$. But, by transitivity, $G\cdot H=0$ and $G+H=1$. Therefore, $G=\overline{H}$ and, because of this, $H=\overline{G}$, proving both laws.

Duality is a direct implication of De Morgan’s Laws. Given a function $f=g+h$, by De Morgan’s Laws, $\overline{f}=\overline{g}\cdot\overline{h}$. By recursively performing De Morgan’s Laws on *g* and *h*, all ORs switch to ANDs, and all values become complemented: 1s to 0s and 0s to 1s. Of course, we can just rename all variables with their complements and have no loss of generality, since each form of the function (it and its dual) are independent. Similarly, if $f=g\cdot h$, $\overline{f}=\overline{g}+\overline{h}$, so all ANDs switch to ORs, all 1s to 0s, and all 0s to 1s.

When realizing a truth table, one must create a hardware circuit that takes in the appropriate inputs and produces the correct outputs. It is important to have as small a function as possible. Although it is always possible to arrive at an optimal solution solely using boolean algebra, it is not suitable for use with a function that has many inputs.

However, there are other techniques that may not always arrive at the most optimal solution, but are easier to use when dealing with large truth tables.

These techniques create a Sum of Products (SOP) solution. An SOP function consists of many inputs being ANDed together, and then the ANDed terms being ORed together. For example, an SOP function may look like: AB + CD + DEF. It consists of three terms (AB, CD, and DEF) all being ORed together. Each instance of an input is called a literal. In this expression, there are seven literals. A term is an ANDed group of literals. In this expression, there are three terms.

Any truth table can be realized with an SOP solution. For example, take the following truth table:

A	B	C	OUT
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

We want OUT to be 1 only when a “1” is listed under OUT in the truth table. One of the times this happens is when A is 0, B is 0, and C is 0. Therefore, we need OUT to be 1 whenever A is 0 AND B is 0 AND C is 0. An expression that is 1 *only* when this is 1 is $\overline{A} \cdot \overline{B} \cdot \overline{C}$. When A, B, and C are all 0, this expression is 1. In addition, we want OUT to be 1 whenever A is 0, B is 0, and C is 1 (the second line). An expression that is 1 *only* when this is 1 is $\overline{A} \cdot \overline{B} \cdot C$. So an expression that is 1 when *either* (or both) of these cases is 1 is:

$$\overline{A} \cdot \overline{B} \cdot \overline{C} + \overline{A} \cdot \overline{B} \cdot C$$

Similarly, we can generate terms for the other two times OUT is 1, ultimately creating a SOP expression that is 1 if and only if OUT is 1:

$$\text{OUT} = \overline{A} \cdot \overline{B} \cdot \overline{C} + \overline{A} \cdot \overline{B} \cdot C + A \cdot \overline{B} \cdot \overline{C} + A \cdot B \cdot \overline{C}$$

This can be generalized into an algorithm: First, look for lines in the truth table where OUT is 1. For each of these rows, AND together one literal for each input: an uninverted, normal literal if the input is 1, and an inverted input if the input is 0. Then, OR together all of these terms. An SOP solution like this is not at all reduced, and is called a canonical SOP solution. Each of the ORed terms in a canonical solution is called a minterm.

A technique to reduce an SOP solution is to use Karnaugh maps (K-maps). A Karnaugh map is a pictorial representation of an SOP expression. It is a grid with one box for each minterm. The boxes are arranged such that each box differs with an adjacent box by exactly one literal. For example, the K-map for the previous truth table is:

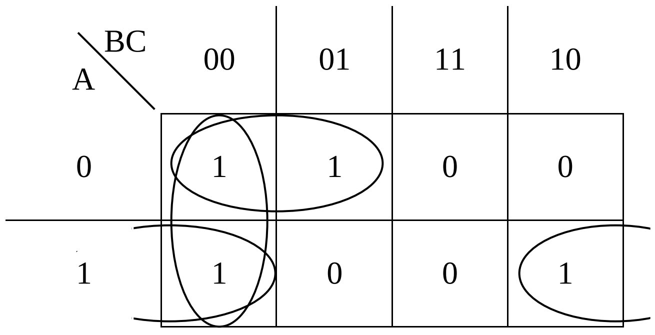
A \ BC	00	01	11	10
0	1	1	0	0
1	1	0	0	1

In this K-map, B and C label the columns and A labels the rows, as marked with a “A\BC” in the top left hand corner. For the column labels, B is the literal on the left, and C is the literal on the right. Notice that the columns are labeled 00, 01, 11, 10. In this sequence, each number is different from its two surrounding numbers by exactly one bit. Also, the numbers at the beginning and end of the sequence “wrap around:” they also differ by exactly one bit. Each entry has exactly 3 adjacent entries: one for each input variable.

Because of this, the minterms associated with adjacent boxes that contain 1s (horizontally and vertically only; not diagonally) differ by one literal. Thus, we can apply the adjacency theorem to two adjacent boxes and condense them to one smaller term. For example, both the 000 and 001 boxes contain 1s. Therefore, by adjacency, we can say:

$$\underbrace{\overline{A}\,\overline{B}\,\overline{C}+\overline{A}\,\overline{B}C}_{\text{minterms}}=\overline{A}\,\overline{B}$$

Notice that $\overline{A}\,\overline{B}$ corresponds to 00, which are the only two literals that the boxes have in common in the K-map. We group together the boxes in the K-map to denote that we are applying adjacency to them. The map with all the groups circled is:



Each of these groups represents a term in the SOP cover. To find the term, we can just look to see what literals all the boxes in each group have in common. The product of these literals is a term in the SOP cover. Thus, the SOP cover represented here is:

$$\overline{A}\,\overline{B}+A\overline{C}+\overline{B}\,\overline{C}$$

However, the $\overline{B}\,\overline{C}$ term is redundant. Graphically, that group overlaps the other two groups, and groups two boxes that are already circled. In boolean algebra, the theorem that states that this redundant term can be eliminated is the Law of Consensus.

The Law of Consensus states:

$$AB + \overline{A}C + BC = AB + \overline{A}C$$

It can be proved using Shannon's Expansion Theorem.

Shannon's Expansion Theorem states:

$$f(x_1, x_2, \ldots, x_n) = x_1 \cdot f(1, x_2, ..., x_n) + \overline{x_1} \cdot f(0, x_2, \ldots, x_n)$$

where $f(x_1,x_2,\ldots,x_n)$ is a boolean function with n parameters and x_i is a boolean variable.

Proof of Shannon's Expansion Theorem:

The proof of Shannon's Expansion Theorem is broken up into two cases: one case for when $x_1 = 1$ and one for when $x_1 = 0$.

Case 1: $x_1 = 1$

Since $x_1 = 1$, $f(x_1, x_2, \ldots, x_n) = f(1, x_2, \ldots, x_n)$. Because $X = 1 \cdot X$, $0 = 0 \cdot X$, and $X + 0 = X$, $f(1, x_2, \ldots, x_n) = f(1, x_2, ..., x_n) \cdot 1 + f(0, x_2, ..., x_n) \cdot 0$. Since $x_1 = 1$, $f(1, x_2, \ldots, x_n) \cdot 1 + f(0, x_2, \ldots, x_n) \cdot 0 = f(1, x_2, \ldots, x_n) \cdot x_1 + f(0, x_2, ..., x_n) \cdot \overline{x_1}$. Therefore, by transitivity, $f(x_1, x_2, \ldots, x_n) = x_1 \cdot f(1, x_2, ..., x_n) + \overline{x_1} \cdot f(0, x_2, \ldots, x_n)$.

Case 2: $x_1 = 0$

Since $x_1 = 0$, $f(x_1, x_2, \ldots, x_n) = f(0, x_2, \ldots, x_n)$. Because $X = 1 \cdot X$, $0 = 0 \cdot X$, and $0 + X = X$, $f(0, x_2, \ldots, x_n) = f(1, x_2, ..., x_n) \cdot 0 + f(0, x_2, ..., x_n) \cdot 1$. Since $x_1 = 0$, $f(1, x_2, \ldots, x_n) \cdot 0 + f(0, x_2, \ldots, x_n) \cdot 1 = f(1, x_2, \ldots, x_n) \cdot x_1 + f(0, x_2, ..., x_n) \cdot \overline{x_1}$. Therefore, by transitivity, $f(x_1, x_2, \ldots, x_n) = x_1 \cdot f(1, x_2, ..., x_n) + \overline{x_1} \cdot f(0, x_2, \ldots, x_n)$.

Proof of the Law of Consensus:

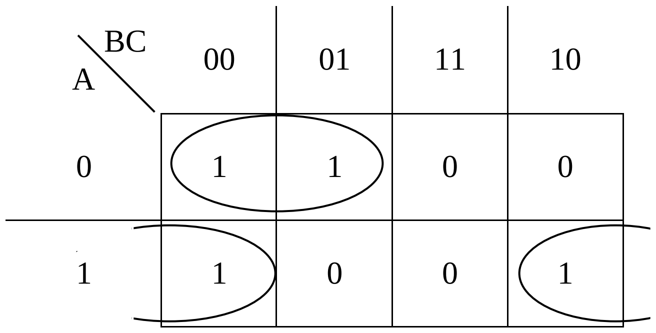
Now, with Shannon's Expansion Theorem, the Law of Consensus can be proved. Using Shannon's Expansion Theorem on $f(A, B, C) = AB + \overline{A}C + BC$, we get:

$AB+\overline{A}C+BC = A(B+BC)+\overline{A}(C+CB)$. By Absorption, we can combine both $B + BC$ and $C + CB$ to get: $A(B+BC)+\overline{A}(C+CB) = A \cdot B + \overline{A} \cdot C$. Therefore, by transitivity, $AB+\overline{A}C+BC = AB+\overline{A}C$.

Using this on the K-map example, the cover from the three circles can be simplified:

$$\overline{A}\overline{B}+A\overline{C}+\overline{B}\overline{C}=\overline{A}\overline{B}+A\overline{C}$$

This new cover has only two terms and four literals, much less than the original function with four terms and twelve literals. Thus, there is a 50% reduction in the number of terms and a 66 $\frac{2}{3}$% reduction in the number of literals. The new cover is graphically represented by two circles, namely the ones that are not completely overlapped:



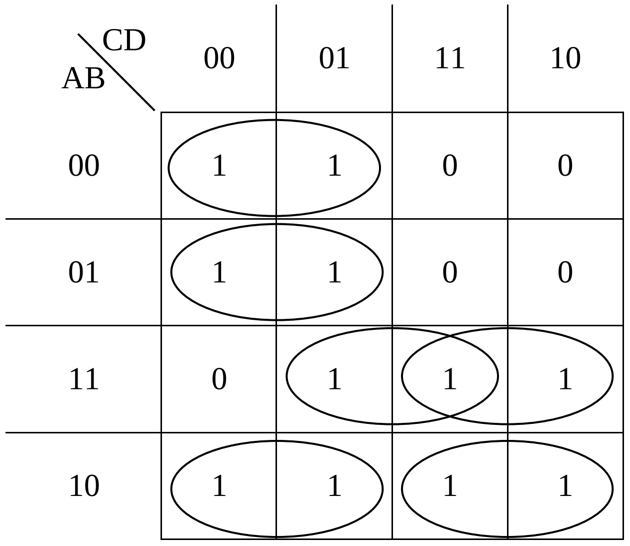
A \ BC	00	01	11	10
0	1	1	0	0
1	1	0	0	1

Whenever a circle is completely overlapped (every box it contains is also contained in another circle), the Law of Consensus can be applied. Thus, these circles do not even have to be circled to generate a cover of the function.

Adjacency can be applied to grouping not only single boxes, but also entire groups of boxes whose height and width are both powers of two. For example, take the following K-map:

AB \ CD	00	01	11	10
00	1	1	0	0
01	1	1	0	0
11	0	1	1	1
10	1	1	1	1

Circling every group of two boxes (and omitting those that can be eliminated by the Law of Consensus), we get:



AB \ CD	00	01	11	10
00	1	1	0	0
01	1	1	0	0
11	0	1	1	1
10	1	1	1	1

This produces a cover:

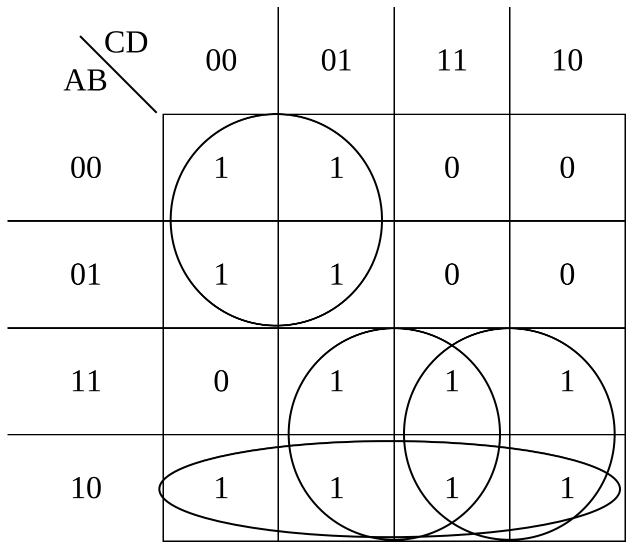
$$\bar{A}\bar{B}\bar{C} + \bar{A}B\bar{C} + ABD + ABC + A\bar{B}\bar{C} + A\bar{B}C$$

However, Adjacency can be applied to many of these terms (such as $\bar{A}\bar{B}\bar{C} + \bar{A}B\bar{C}$).

Applying adjacency to all possible terms and repeating, the cover becomes:

$$\bar{A}\bar{C} + AD + AC + A\bar{B}$$

Graphically, this is represented by combining adjacent circles:



AB \ CD	00	01	11	10
00	1	1	0	0
01	1	1	0	0
11	0	1	1	1
10	1	1	1	1

This cover of the function has only four terms and eight literals, reduced from eleven terms and forty-four literals. This is approximately a 64% reduction in terms and a 82% reduction in literals.

Thus, the algorithm for using a K-map is:

- Circle areas of the K-map where each box in the area contains a 1. Each area’s height and width (measured in number of boxes) must be a power of 2. Areas that are already covered by circles do not need to be circled, although it may be beneficial to do so when grouping areas that are not yet covered (since larger areas produce smaller terms).
- Each circle represents a term in the cover. Write down each term by ANDing the literals that are the same in all boxes contained by the circle. Do not complement literals that appear as a 1 in the column or row label and complement those that appear as a 0. Then, OR together all the terms.

K-maps are an intuitive and easy way to reduce covers of functions with four or less inputs, but they do not work well for functions with over four inputs. It is possible to use a K-map with more than two dimensions to represent these functions, but they are hard to deal with in three dimensions, and almost impossible to use in more than three. For a function with more than four inputs, other methods must be used. One such method is the Quine-McCluskey (Q-M) algorithms.

The Q-M algorithms are two algorithms that are used to minimize a cover of a boolean function. The first algorithm generates a cover of prime implicants (implicants (terms) that are fully reduced by adjacency). The second algorithm takes the prime implicants from the first algorithm and eliminates those that are not needed.

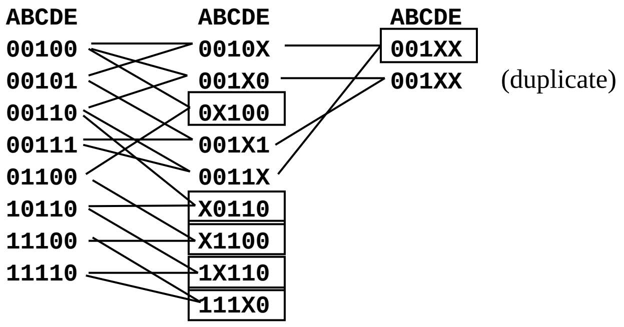
In the first Q-M algorithm, the minterms of the function are listed by using a 1 when a literal is not complemented and a 0 when a literal is complemented. For example, take the following 5-input function (1s are in bold to make it easier to see them):

A	B	C	D	E	OUT
0	0	0	0	0	0
0	0	0	0	1	0
0	0	0	1	0	0
0	0	0	1	1	0
0	0	1	0	0	**1**
0	0	1	0	1	**1**
0	0	1	1	0	**1**
0	0	1	1	1	**1**
0	1	0	0	0	0
0	1	0	0	1	0
0	1	0	1	0	0
0	1	0	1	1	0
0	1	1	0	0	**1**
0	1	1	0	1	0
0	1	1	1	0	0
0	1	1	1	1	0
1	0	0	0	0	0
1	0	0	0	1	0
1	0	0	1	0	0
1	0	0	1	1	0
1	0	1	0	0	0
1	0	1	0	1	0
1	0	1	1	0	**1**
1	0	1	1	1	0
1	1	0	0	0	0
1	1	0	0	1	0
1	1	0	1	0	0
1	1	0	1	1	0
1	1	1	0	0	**1**
1	1	1	0	1	0
1	1	1	1	0	**1**
1	1	1	1	1	0

The first step of the first Q-M algorithm yields the following list:

```
ABCDE
00100
00101
00110
00111
01100
10110
11100
11110
```

In the next part of the Q-M algorithm, the theorem of Adjacency is applied. To apply it, look at the implicants and find implicants that differ by exactly one literal. Combine the two implicants by putting an X (for don't-care) in place of the literal they differ by, and write the new implicant in the next column. Throw out any duplicate implicants that may have been generated. Draw a box around any implicants that could not be combined with other implicants. Repeat this process with the next column until no more combinations are possible. The boxed implicants are the ones that could not be combined: they are the prime implicants. In this example, the first algorithm produces:



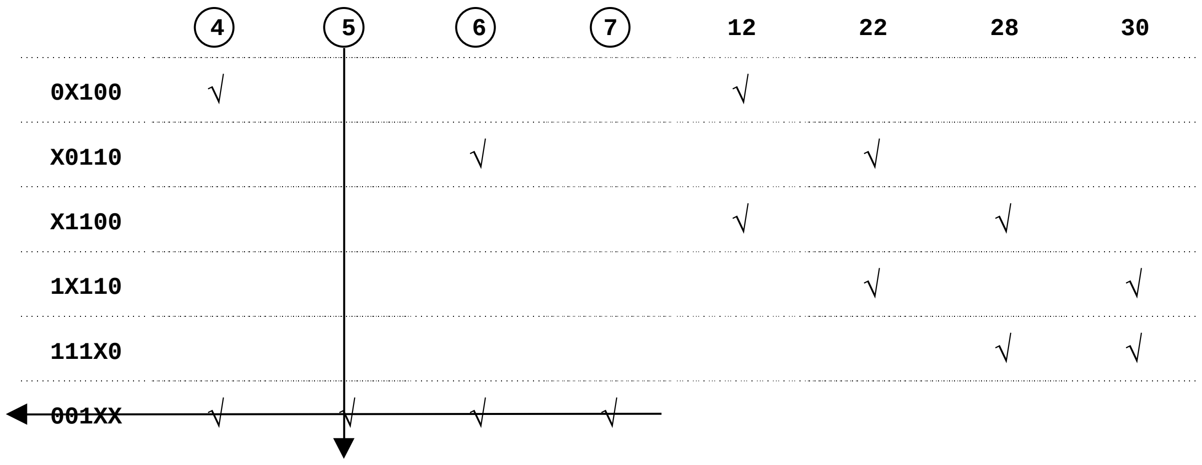
Therefore, the cover of the function using these prime implicants is:

$$\bar{A}C\bar{D}\,\bar{E}+\bar{B}CD\bar{E}+BC\bar{D}\,\bar{E}+ACD\bar{E}+A\,B\,C\,\bar{E}+\bar{A}\bar{B}C$$

The second Q-M algorithm can further minimize a cover generated by the first algorithm. The first part of the second algorithm is to make a table whose rows are labeled by the prime implicants and whose columns are labeled by the minterms of the function. When labeling the columns with the minterms, it is usually easier to write down the base 10 number represented by the binary (base 2) number that the minterm represents. In this case, E would be the value in the 1's place, and A would be the value in the 16's place. Then, put a check mark in each box where the minterm of that column is covered by the implicant of that row. The table for this function is:

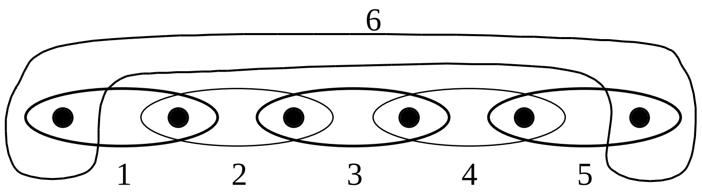
	4	5	6	7	12	22	28	30
0X100	√				√			
X0110			√			√		
X1100					√		√	
1X110						√		√
111X0							√	√
001XX	√	√	√	√				

The next part of the algorithm is to find all columns that contain exactly one check (all columns contain at least one check). Draw a line across the row that this check is in, and circle the minterms of all the checks that this line intersects to signify that that minterm has been covered. Do not do this process for any circled columns. After doing this, the table is:



	4	5	6	7	12	22	28	30
0X100	√				√			
X0110			√			√		
X1100					√		√	
1X110						√		√
111X0							√	√
~~001XX~~	~~√~~	~~√~~	~~√~~	~~√~~				

At this point, 12, 22, 28, and 30 are uncovered. Since they each have more than one check in their column, we must choose what checks to take first. Our choice will affect the number of implicants in our cover. There is always a way to find the minimum cover with the second algorithm[ii]. However, there are many intricacies involved in finding an algorithm to always find the correct choices. For example, take the following diagram with the dots representing minterms—which may be spread out in a random, nonconsecutive order in the list of minterms—and the circles representing prime implicants:



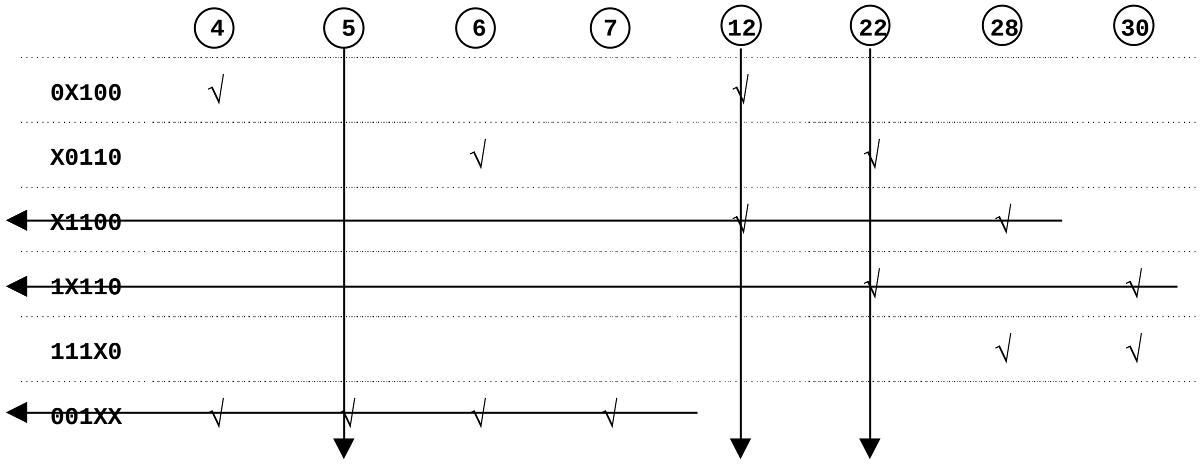
The essential prime implicants are the dark circles. The lighter circles are nonessential and can be thrown away. A perfect Q-M algorithm would pick the dark circles and not the light circles. Also, this picture is only in two dimensions. In a normal situation, these minterms may be spread out over many more dimensions, increasing the complexity of the problem.

I was not able to find the perfect second Q-M algorithm that always finds the minimum cover[*]. My Q-M algorithm may choose circle 1 and then, because the minterms in this diagram may not be in same order as they are in the algorithm’s list of minterms, choose circle 4 instead of circle 3. The best second Q-M algorithm I found is the following (on the next page):

[*] MacEspresso 1.0, a port of the Espresso algorithm by Mikhail Fridberg, was able to find a more minimal cover when running with the “–Dexact” parameter.

1. Pick the column with the least number of checks in it. If there is a tie, pick the first one.
2. In this column, pick the check whose row will get us the greatest number of *uncovered* minterms.

Using this algorithm on the above table yields:



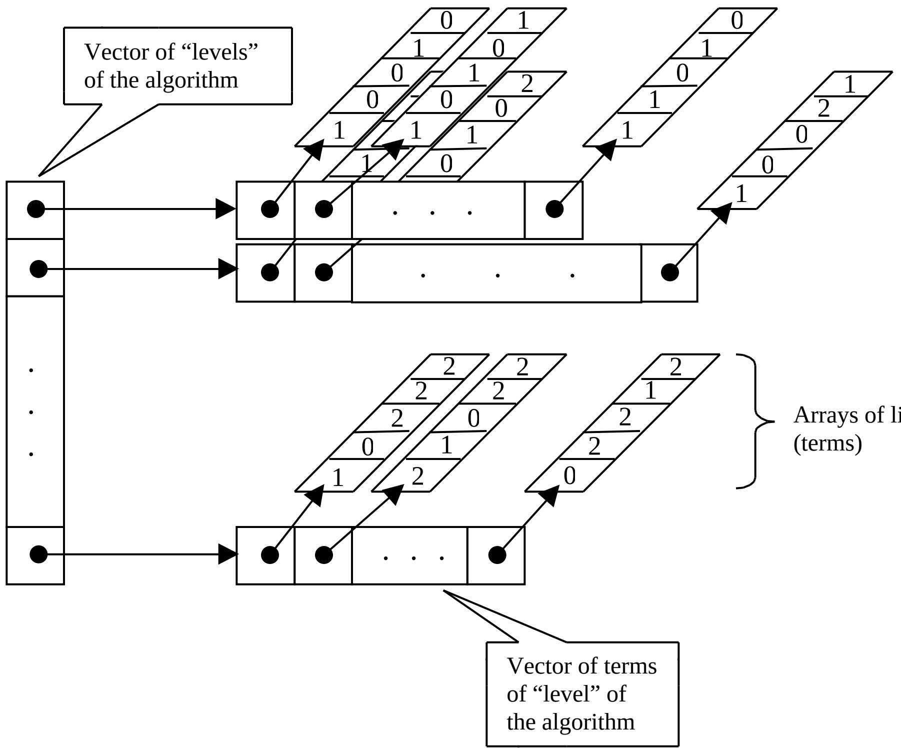
	4	5	6	7	12	22	28	30
0X100	√				√			
X0110			√			√		
~~X1100~~					√		√	
~~1X110~~						√		√
111X0							√	√
~~001XX~~	√	√	√	√				

Now, all the minterms are circled, so they are all covered. The algorithm stops, and we are left with three implicants in our cover:

$$BC\overline{D}\overline{E} + ACD\overline{E} + \overline{A}\overline{B}C$$

This is a 62.5% reduction in the number of terms and a 72.5% reduction in the number of literals.

This process may be tedious to go through by hand, especially if there are many inputs. However, because of the algorithmic, recursive nature of these algorithms, they are fit to be programmed into a computer. I programmed them in Java (see Appendix A for source code). I implemented the first Q-M algorithm using a 3-dimensional vector/array combination. In this case, a “vector” is a computer vector, not a math vector: it is an array without a preset, limited size. The first vector holds the different “levels” (iterations) of the first Q-M algorithm. Each element in the first vector is a vector of arrays. These arrays in the third and last level of the structure are the terms. Their size is the number of inputs, and they hold the literals. For the literals, I store a 0 for 0, a 1 for 1, and a 2 for don't-care ("X"). The following picture is a graphical representation of this data structure:


Vector of "levels" of the algorithm
Arrays of literals (terms)
Vector of terms of "level" of the algorithm

Note that no 2s are in any of the implicants in the first level, as no terms have yet been combined by adjacency. Each implicant in the second level contains exactly one 2, since exactly one iteration of the algorithm has been done, yielding one don't-care per new implicant. Each implicant in the third level contains two 2s, and so on. The first level (the canonical cover) is read in from a file. Each level after the first is created by performing the first Q-M algorithm on the previous level.

The implementation of the first Q-M algorithm consists of four nested for loops. The first (outer) loop goes through each level. The second loop goes through each term.

The third loop goes through all the terms that have not already been compared with that term (all the ones below it), and the fourth loop compares each literal of the two terms to see if there is exactly one difference in literals. If there is exactly one difference, the two terms are combined by adjacency: a new term with a 2 in the different literal’s place and all other literals the same as those in the two compared terms is put into the next level. Also, the two compared terms’ indices are put into a set of non-primes. After each term has been compared, the set of non-primes is used to find the terms that are prime. The prime terms are added to the set of primes, which is the final result of the algorithm. Pseudocode for this is on the next page (NUMINPUTS is the number of inputs).

```
Function QM1( levels ) returns primes

put { } (empty set) into primes
for each level level do
    irredundant level (remove duplicates from level)
    put { } (empty set) into nonprimes
    for every term term in level from 1 to the size of level –1 do
        for every term compterm in level from term + 1 to the size of level do
            put –1 into differentLiteral as a flag
            for every literal literal from 1 to NUMINPUTS do
                if literal literal of term ≠ literal literal of compterm then
                    if differentLiteral = –1 then
                        (there was no previous difference)
                        put literal into differentLiteral
                    else (there was a previous difference)
                        put –1 into differentLiteral as a flag
                        break (get out of this comparison loop)
                    end if
                end if
            end for of literal
            if differentLiteral ≠ –1 then (there was exactly one difference)
                add term to nonprimes
                add compterm to nonprimes
                add term with 2 in literal differentLiteral to level level + 1
        end for of compterm
    end for of term
    if the size of nonprimes > 0 then
        add all terms not in nonprimes to primes
    else
        break (get out of loop for levels)
    end if
end for of level
```

```
return primes
```

Because of its four nested loops, the first Q-M algorithm takes a long time to execute. For just one iteration of the outer loop (for only one level), there are approximately *L* iterations of the *term* loop, and each of those has *L – term* iterations of the *compterm* loop, where *L* is the size of the level. Thus, for each level, the total number of times the *literal* loop is called is approximately:

$$L + (L-1) + (L-2) + \ldots + (L-L) = \sum_{k=0}^{L}(L-k)$$

Simplifying this,

$$\sum_{k=0}^{L}(L-k) = \sum_{k=1}^{L+1}(L-k+1) = \sum_{k=1}^{L+1}L - \sum_{k=1}^{L+1}k + \sum_{k=1}^{L+1}1 = L^2 + L - \frac{(L+1)(L+2)}{2} + L + 1$$

$$= L^2 + 2L + 1 - \frac{L^2+3L+2}{2} = \frac{2L^2+4L+2-L^2-3L-2}{2} = \frac{L^2+L}{2}$$

Then, for each of these iterations, there is an iteration of the *literal* loop. The *literal* loop always has NUMINPUTS iterations. So, if *N* = NUMINPUTS, each level takes about $O\left(\left[\frac{L^2+L}{2}\right]N\right) = O(L^2N)$ time. This is a very long time considering that the number of inputs (*N*) and the number of terms in a level (*L*) can both commonly be in the hundreds.

To implement the second Q-M algorithm, I made a 2-dimensional array for the table of checks and a 1-dimensional array to store which minterms are “circled.” I then wrote several different versions of the algorithm, and put the best one in my final version. The best one is:

- Pick the column with the least number of checks in it. If there is a tie, pick the first one.
- In this column, pick the check whose row will get us the greatest number of *uncovered* minterms.

Pseudocode for this is on the next page.

```
Function QM2( minterms, implicants ) returns essentialPrimes

put a 2-dimensional array into checkTable (a value of false indicates there
        is no check. The first dimension is the minterms (columns) and the second
        dimension is the implicants (rows))
for every implicant implicant of implicants do
        for every minterm minterm of minterms do
                if implicant implies (covers) minterm then
                        put true into checkTable[minterm][implicant]
                else
                        put false into checkTable[minterm][implicant]
                end if
        end for of minterm
end for of implicant
put { } (empty set) into essentialPrimes
put a 1-dimensional array of all false into mintermsDone (to keep track of circles)
while not every index in mintermsDone contains true do
        put the column of checkTable with the least number of checks into minChecksCol
        put the row with a check in minChecksCol that contains the greatest number of
                checks in columns not in mintermsDone into maxChecksRow
        put true into every index of mintermsDone that corresponds to a column with a
                check in maxChecksRow
        add implicant maxChecksRow to essentialPrimes
end while
return essentialPrimes
```

In the worst case, the second Q-M algorithm can take longer to run than the first algorithm, but normally it takes much shorter. In the worst case, making *checkTable* takes about $O(I * M * N)$ time, where I is the number of prime implicants, M is the number of minterms, and N is the number of inputs. In the best case, it takes about $O(I * M)$ time.

Finding *minChecksCol* takes about $O(M * I)$ time at first, but then, as more inputs are covered, it decreases. Finding *maxChecksRow* takes about the same time. In the worst case scenario, where the prime implicants are the minterms, making *checkTable*, *minChecksCol*, and *maxChecksRow* all take the greatest amount of time. Worse still, the amount of time to find *minChecksCol* and *maxChecksRow* in each iteration of the while loop only decreases by 1 each time, since only 1 minterm gets covered by the chosen implicant. This makes the total amount of time to find all *minChecksCol*'s values in all the iterations of the while loop about:

$$(M * I) + (M * I - 1) + (M * I - 2) + (M * I - 3) + \ldots + (M * I - M) = \sum_{k=0}^{M}(M * I - k).$$

Using the fact that $I = M$ (because it's the worst case) and simplifying,

$$\sum_{k=0}^{M}(M * I - k) = \sum_{k=1}^{M+1}(M^2 - k + 1) = \sum_{k=1}^{M+1}(M^2) - \sum_{k=1}^{M+1}k + \sum_{k=1}^{M+1}1$$

$$= M^3 + M^2 - \frac{(M+1)(M+2)}{2} + M + 1 = M^3 + \frac{2M^2 - M^2 - 3M - 2 + 2M + 2}{2}$$

$$= M^3 + \frac{M^2 - M}{2}$$

This is also about the amount of time it takes to find all *maxChecksRow*'s values. Therefore, in the worst case, the second Q-M algorithm takes about $O\left(2M^3 + M^2 - M + M^2N\right) = O(M^3 + M^2N)$ time to run (the M^2N is from making *checkTable*). However, in a normal scenario, there are much fewer implicants than there are minterms. Also, there are not normally $M + 1$ iterations of the while loop, since most of the implicants cover more than one minterm. This makes the second algorithm normally run in just more than a tenth of the time taken by the first algorithm.

Simplify is another algorithm for minimizing a SOP function[iii]. I programmed it in Java (see Appendix B for source code). It does not reduce a given function as much as the Q-M algorithms, but it runs in much less time. Simplify uses Shannon's Expansion Theorem to break up the given function into functions that are can be easily reduced, and then merges the results. Shannon's Expansion Theorem (on page 10) states:

$$f(x_1, x_2, \ldots, x_n) = x_1 \cdot f(1, x_2, \ldots, x_n) + \overline{x_1} \cdot f(0, x_2, \ldots, x_n)$$

where *f* is a boolean function with *n* inputs. This can be efficiently turned into computer code by making use of the cofactor of a function with respect to a variable.

The cofactor operation makes use of matrix representations of implicants and SOP functions. An implicant can be represented by a one-dimensional matrix of 0s, 1s, and 2s. The *n*th place in the matrix is the state of the *n*th literal of the implicant: 0 for complemented, 1 for uncomplemented, and 2 for don't-care (the variable does not appear in the implicant). For example, the implicant $x_1 x_2 \overline{x_4}$ is represented as [1 1 2 0]. A set of implicants (a cover of a SOP function) can be represented by a two-dimensional matrix whose rows are the representations of the implicants that collectively cover the function. For example, the function $f = x_1 x_2 \overline{x_4} + x_2 \overline{x_3} + x_1 x_2 x_3 x_4$ is represented as:

$$f = \begin{bmatrix} 1 & 1 & 2 & 0 \\ 2 & 1 & 0 & 2 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

The cofactor of a SOP function *f* with respect to an implicant *x*, f_x, is defined such that for every column index *i* between 1 and the number of inputs, inclusive, and every row index *j* between 1 and the number of rows in *f*, inclusive:

$$\left(f_x\right)_i^j = \begin{cases} \varnothing \text{ if } (f_i^j = 0 \text{ and } x_i = 1) \text{ or } (f_i^j = 1 \text{ and } x_i = 0) & \text{(Case 1)} \\ 2 \text{ (don't-care) if } x_i \neq 2 \text{ and Case 1 is false} & \text{(Case 2)} \\ f_i^j \text{ if } x_i = 2 & \text{(Case 3)} \end{cases}$$

For example, f_i^j is the number in the ith column and jth row of *f*. It is the ith literal of the jth implicant of *f*. x_i is number in the ith column (place) of *x*. It is the ith literal of *x*. The cofactor operation forms a function f_x, which has *i* columns and *j* or less rows. If Case 1 ever happens ($\left(f_x\right)_i^j$ is ∅), then $\left(f_x\right)^j$—the jth row of f_x—is deleted, and the jth row would become the (j+1)th row, the (j+1)th row the (j+2)th row, and so on. For example, the cofactor of the following function *f* with respect to the following implicant *x* is the following function f_x:

$$f = \begin{bmatrix} 1 & 1 & 2 & 0 \\ 2 & 1 & 0 & 2 \\ 1 & 1 & 1 & 1 \end{bmatrix}, \quad x = \begin{bmatrix} 2 & 1 & 1 & 2 \end{bmatrix}, \quad f_x = \begin{bmatrix} 1 & 2 & 2 & 0 \\ 1 & 2 & 2 & 1 \end{bmatrix}$$

The cofactor of a function *f* with respect to a variable x_i of *f* is defined as the cofactor of *f* with respect to an "implicant" whose literals are all 2s except for the ith literal, whose value is x_i. Effectively, it produces a function f_{x_i} whose implicants all have a 2 in the ith literal and the literals of all the implicants in *f* that have a value of 2 or x_i in their ith literal in the other literals. For example, using the function *f* from the previous example, $f_{\overline{x_3}}$ is found by taking the cofactor of *f* with respect to the implicant representing the literal $\overline{x_3}$:

$$f = \begin{bmatrix} 1 & 1 & 2 & 0 \\ 2 & 1 & 0 & 2 \\ 1 & 1 & 1 & 1 \end{bmatrix}, \quad \overline{x_3} = \begin{bmatrix} 2 & 2 & 0 & 2 \end{bmatrix}, \quad f_{\overline{x_3}} = \begin{bmatrix} 1 & 1 & 2 & 0 \\ 2 & 1 & 2 & 2 \end{bmatrix}$$

Shannon’s Expansion Theorem can be stated as:

$$f = x_i f_{x_i} + \overline{x_i} f_{\overline{x_i}}$$

for any *i* between 1 and the number of inputs of *f*, inclusive. This is because the each of the cofactors of *f* do not contain the implicants that would evaluate to 0 if an algebraic application of Shannon’s Expansion Theorem were used. The cofactor operation removes those implicants with its Case 1. In addition, the cofactor operation turns the *i*th literal of each implicant into 2 (don’t-care). This corresponds to algebraically plugging in a 1 or 0 into x_i, thus removing it from each term. Because x_i is the variable “splitting” *f*, x_i is called the splitting variable.

Simplify makes use of Shannon’s Expansion Theorem to recursively break up the given function *f* until it comes across a unate function. A unate function is a function that is monotone increasing or monotone decreasing in each of its variables. A function is monotone increasing in a variable x_i if changing x_i from 0 to 1 makes the output of the function 1 (although the output of the function need not be 0 beforehand). A function is monotone decreasing in a variable x_i if changing x_i from a 0 to a 1 makes the output of the function 0 (although the output of the function need not be 1 beforehand). Note that this says nothing about what happens if x_i is changed from a 1 to a 0. Also, a non-unate function is called binate.

A cover of a function is unate if the *i*th literal of every implicant of the cover of the function is a 2 or a 1, or a 2 or a 0. That is, if each column in the matrix representation of the cover of the function contains only 1s and 2s or only 0s and 2s. For example, the cover of the following function *U* defined by its matrix representation is unate, and the cover of the following function *B* is not unate:

$$U = \begin{bmatrix} 1 & 2 & 0 & 1 \\ 1 & 0 & 2 & 2 \\ 1 & 0 & 0 & 1 \\ 1 & 2 & 2 & 2 \\ 1 & 0 & 2 & 1 \end{bmatrix} \qquad B = \begin{bmatrix} 0 & 2 & 1 & 1 \\ 0 & 2 & 0 & 1 \\ 2 & 0 & 1 & 2 \\ 0 & 1 & 1 & 2 \\ 2 & 0 & 0 & 1 \end{bmatrix}$$

The cover of *B* is binate because its 2nd and 3rd columns contain both 1s and 0s. The cover of *U* is unate because no column of *U* contains both 1s and 0s. If a cover of a function is unate, then the function must be unate. However, if a cover is not unate, it is possible that the function is unate. For example, the following function *F* is unate, but its cover is not:

$$F = \begin{bmatrix} 1 & 1 & 0 \\ 2 & 0 & 2 \end{bmatrix}$$ [iv]

In Simplify, however, we only deal with covers of functions, and it would not be beneficial to the algorithm to check to see if the function is unate if the cover is not. Therefore, from this point forward in the paper, I will not differentiate between unate covers and unate functions.

In Simplify, a unate function is simplified by removing the implicants of the function that are covered by other implicants of the function. Since each column of the unate function can only contain either 1s and 2s or 0s and 2s, it is more likely that this simplification can happen than if the function were binate. For example, the following function *F* gets reduced to the function F':

$$F = \begin{bmatrix} 1 & 0 & 2 & 2 \\ 2 & 0 & 1 & 1 \\ 1 & 0 & 2 & 2 \\ 2 & 2 & 1 & 2 \\ 1 & 2 & 1 & 1 \\ 2 & 0 & 1 & 2 \end{bmatrix} \text{ is reduced to } F' = \begin{bmatrix} 1 & 0 & 2 & 2 \\ 2 & 2 & 1 & 2 \end{bmatrix}$$

Note that an implicant *a* covers another implicant *b* if and only if each literal *i* of *a* is a 2 or equals the *i*th literal of *b*. That is, the *i*th literal of *b* cannot be a 1 when the *i*th literal of *a* is a 0, a 0 when the *i*th literal of *a* is a 1, or a 2 when the *i*th literal of *a* is not 2. If it were, then *b* would not be covered by *a*. The following is pseudocode for Unate Simplify:

```
Function unate_simplify( f ) returns f′ (f is a cover of a function)
put f into f′
for every implicant implicant of f from 1 to the number of implicants in f – 1 do
    for every implicant compImplicant of f from the number of implicants in f down
        to the index of implicant + 1 do
        if implicant contains compImplicant then
            remove compImplicant from f′
        end if
    end for of compImplicant
end for of implicant
return f′
```

After Simplify simplifies the unate functions, it merges the functions back together. The Merge algorithm puts together the two functions formed by performing Shannon’s Expansion Theorem on a function to form a new function logically equivalent to the original function. Merge not only splices the two functions into one function, but also performs the AND operation needed on each of the functions to complete Shannon’s Expansion Theorem. However, if Merge just performed the AND operation and put the two resulting functions together (effectively ORing them), many redundancies would result. Thus, before Merge performs the AND and OR operations, it checks to see if it can take out any redundancies.

Merge takes three parameters: $h_1 = f_{x_i}$, $h_0 = f_{\overline{x_i}}$, and *splittingVar* = i (the index of the splitting variable). It returns a function h created by merging h_1 and h_0. To check for redundancies, Merge first checks to see if any implicants of h_1 and h_0 are identical. Those that are identical are put into a set of implicants h_2 and removed from h_1 and h_0. Then, Merge checks to see if any implicants in h_1 cover any of those in h_0, and vice versa. Implicants in h_0 that are covered by h_1 are removed from h_0 and put into h_2. Implicants in h_1 that are covered by h_0 are removed from h_1 and put into h_2. Because the implicants in h_2 are those that were covered by both h_1 and h_0, they should make $h = 1$ no matter x_i is. Thus, they are not ANDed with the splitting variable. Since the implicants of h_1 should make $h = 1$ only when the splitting variable is 1, they should be ANDed with x_i, and because the implicants of h_0 should make $h = 1$ only when the splitting variable is 0, they should be ANDed with $\overline{x_i}$. Therefore, $h = h_2 + x_i h_1 + \overline{x_i} h_0$. Pseudocode for Merge is on the next page.

```
Function merge( $h_0$, $h_1$, splittingVar ) returns $h$

put { } (empty set) into $h_2$
for every implicant $i$ of $h_0$ do
    for every implicant $l$ of $h_1$ do
        if $i = l$ then
            add $i$ to $h_2$
            remove $i$ from $h_0$
            remove $l$ from $h_1$
        end if
    end for of $l$
end for of $i$
for every implicant $i$ of $h_0$ do
    for every implicant $l$ of $h_1$ do
        if $i$ covers $l$ then
            add $l$ to $h_2$
            remove $l$ from $h_1$
        else if $l$ covers $i$ then
            add $i$ to $h_2$
            remove $i$ from $h_0$
        end if
    end for of $l$
end for of $i$
put $h_2 + x_{splittingVar} \cdot h_1 + \overline{x_{splittingVar}} \cdot h_0$ into $h$
return $h$
```

Because Simplify breaks a function down into unate functions, we want to make each of the two functions resulting from the application of Shannon's Expansion Theorem as "unate" as possible. Thus, we choose the splitting variable to be the "most" binate variable. This is done by the algorithm Binate Select.

Binate Select chooses the index of the splitting variable, *splittingVar*, such that the column in the matrix representation of *f* corresponding to the variable $x_{splittingVar}$ contains both 1s and 0s and the greatest number of 1s and 0s (the greatest sum of the number of 1s and the number of 0s). This is done by the following pseudocode:

```
Function binate_select( f ) returns splittingVar
put a 1-dimensional array into numZeros (numZeros[k] holds the number of zeros in the
    kth column of the matrix representation of f)
put a 1-dimensional array into numOnes (numOnes[k] holds the number of ones in the
    kth column of the matrix representation of f)
for each column col of the matrix representation of f do
    put the number of 0s in col into numZeros[col]
    put the number of 1s in col into numOnes[col]
end for of col
put { } (empty set) into binateColumns
for each column col of the matrix representation of f do
    if numZeros[col] > 0 and numOnes[col] > 0 then
        there are both 1s and 0s in col, so col is binate:
        add col to binateColumns
    end if
end for of col
put –1 into splittingVar as an initial value and flag saying f is unate
put 0 into maxVal as an initial value of the sum of 0s and 1s in a column
for each column col of the matrix representation of f do
    if numZeros[col] + numOnes[col] > maxVal and col ∈ binateColumns then
        put numZeros[col] + numOnes[col] into maxVal
        put col into splittingVar (that is, the index of col)
    end if
end for of col
return splittingVar
```

Simplify breaks down its given function *f* using the following line of pseudocode:

$$\textbf{\textit{merge}}(\ \textbf{\textit{simplify}}(\ f_{\overline{x_{splittingVar}}}\ ),\ \textbf{\textit{simplify}}(\ f_{x_{splittingVar}}\ ),\ splittingVar\ )$$

It does this recursively until it encounters a unate function. Then, it calls *unate_simplify* on the unate function. Note that this is, in effect, simply applying Shannon's Expansion Theorem. That is why the function generated by Simplify is equivalent to the given function. Pseudocode for Simplify is the following:

```
Function simplify( f ) returns $f'$
if f is unate then
    return unate_simplify( f )
else
    put binate_select( f ) into splittingVar
    if ( splittingVar = -1 ) then
        return unate_simplify( f )
    else
        put merge( simplify( $f_{\overline{x_{splittingVar}}}$ ), simplify( $f_{x_{splittingVar}}$ ), splittingVar) into $f'$
        if the number of implicants in f < the number of implicants in $f'$ then
            return f
        else
            return $f'$
        end if
    end if
end if
```

For an example, let us call Simplify on the following function:

$$f = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 2 \end{bmatrix}$$

First, Simplify calls *binate_select*, which selects column 1 as the splitting variable. Simplify then cofactors *f* with respect to the splitting variable and to the complement of the splitting variable. This breaks up *f*:

$$\begin{tikzcd}[ampersand replacement=\&]
\& \begin{bmatrix}1 \& 0 \& 1\\ 0 \& 0 \& 1\\ 1 \& 1 \& 1\\ 1 \& 1 \& 2\end{bmatrix} \arrow[dl, "\overline{x_1} = \begin{bmatrix}0 \& 2 \& 2\end{bmatrix}"'] \arrow[dr, "x_1 = \begin{bmatrix}1 \& 2 \& 2\end{bmatrix}"] \& \\
\begin{bmatrix}2 \& 0 \& 1\end{bmatrix} \& \& \begin{bmatrix}2 \& 0 \& 1\\ 2 \& 1 \& 1\\ 2 \& 1 \& 2\end{bmatrix}
\end{tikzcd}$$

Simplify is then recursively called on each of these "subfunctions." Because the function on the left is unate, *unate_simplify* is called on it. Since this subfunction only has one implicant, *unate_simplify* cannot simplify it further. For the right subfunction, column 2 is selected as the splitting variable, and the process repeats recursively:

$$\begin{tikzcd}[ampersand replacement=\&]
\& \begin{bmatrix}2 \& 0 \& 1\\ 2 \& 1 \& 1\\ 2 \& 1 \& 2\end{bmatrix} \arrow[dl, "\overline{x_2} = \begin{bmatrix}2 \& 0 \& 2\end{bmatrix}"'] \arrow[dr, "x_2 = \begin{bmatrix}2 \& 1 \& 2\end{bmatrix}"] \& \\
\begin{bmatrix}2 \& 2 \& 1\end{bmatrix} \& \& \begin{bmatrix}2 \& 2 \& 1\\ 2 \& 2 \& 2\end{bmatrix}
\end{tikzcd}$$

Because both the left and right subfunctions are unate, *unate_simplify* is called on each of them. Since there is only one implicant in the left subfunction, *unate_simplify* cannot simplify it further. In the right subfunction, [2 2 2] covers [2 2 1], so *unate_simplify* removes [2 2 1].

Then, *merge* is called to merge together the left branch and the right branch, [2 2 1] and [2 2 2], respectively. Because [2 2 2] contains [2 2 1], [2 2 1] is put into h_2 and removed from h_0, and is not ANDed with the splitting variable. Thus, *merge* yields:

$$\begin{tikzcd}
{\begin{bmatrix}2 & 2 & 1\end{bmatrix}} \arrow[dr, "\overline{x_2}"'] & & {\begin{bmatrix}2 & 2 & 2\end{bmatrix}} \arrow[dl, "x_2"] \\
& {\begin{bmatrix}2 & 2 & 1 \\ 2 & 1 & 2\end{bmatrix}} &
\end{tikzcd}$$

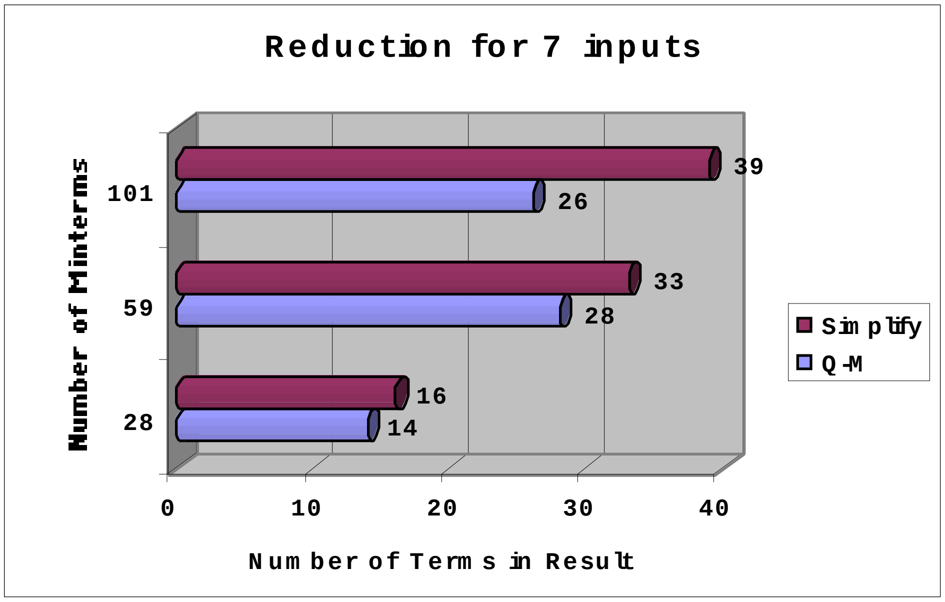
This is then merged with the subfunction obtained by cofactoring *f* with respect to $\overline{x_1}$ ([2 0 1] is not ANDed with $\overline{x_1}$ because it is covered in [2 2 1]):

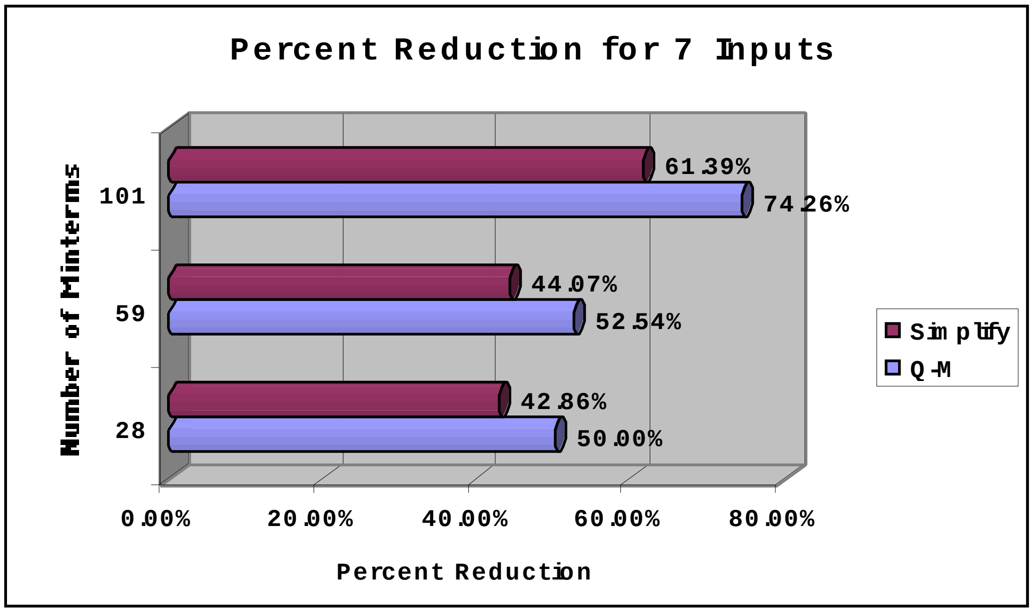
$$\begin{tikzcd}
{\begin{bmatrix}2 & 0 & 1\end{bmatrix}} \arrow[dr, "\overline{x_1}"'] & & {\begin{bmatrix}2 & 2 & 1 \\ 2 & 1 & 2\end{bmatrix}} \arrow[dl, "x_1"] \\
& {\begin{bmatrix}2 & 0 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2\end{bmatrix}} &
\end{tikzcd}$$

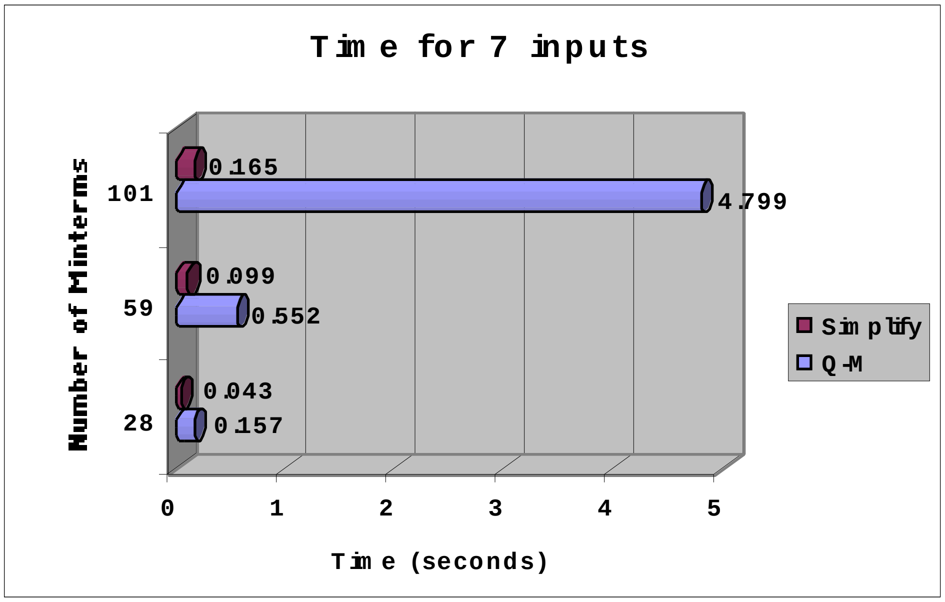
This is the final result by Simplify. It has three terms and six literals: a 25% reduction in terms and a 50% reduction in literals.

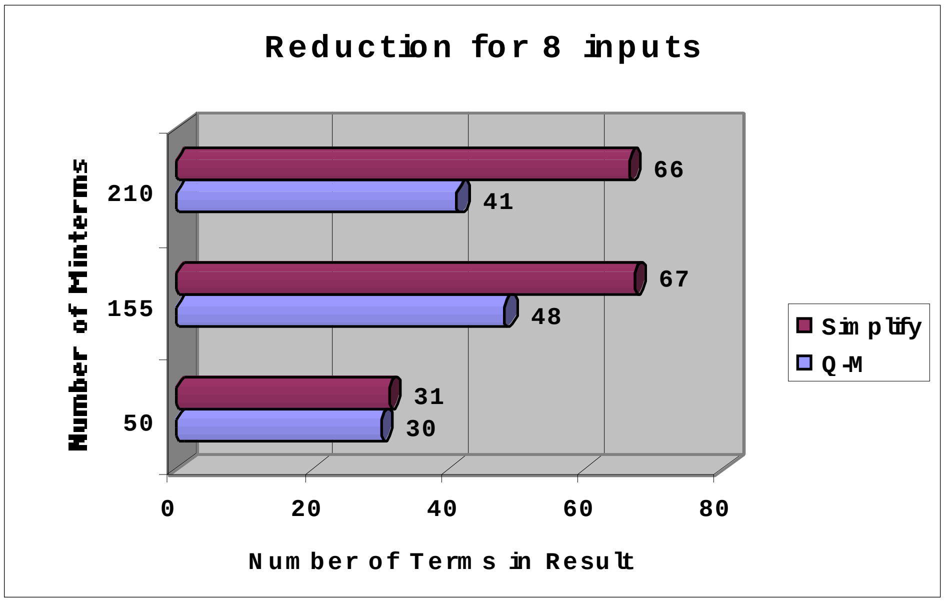
Because Simplify breaks up a function into easy-to-reduce unate functions with a binary recursion tree, it is very efficient in terms of time. However, it is not guaranteed to even come up with a set of prime implicants, much less find a minimum cover. To compare Q-M and Simplify, I created a program that generated random functions by making lists of random minterms. I then ran each on a variety of functions under controlled conditions*. The results are shown in the following charts (on the next four pages). The reduction results are measured in terms instead of literals because these algorithms are used for large functions, and the hardware used to realize large functions, programmable logic arrays (PLAs), is structured in such a way that it does not matter how many literals are in each term, but it does matter how many terms there are.

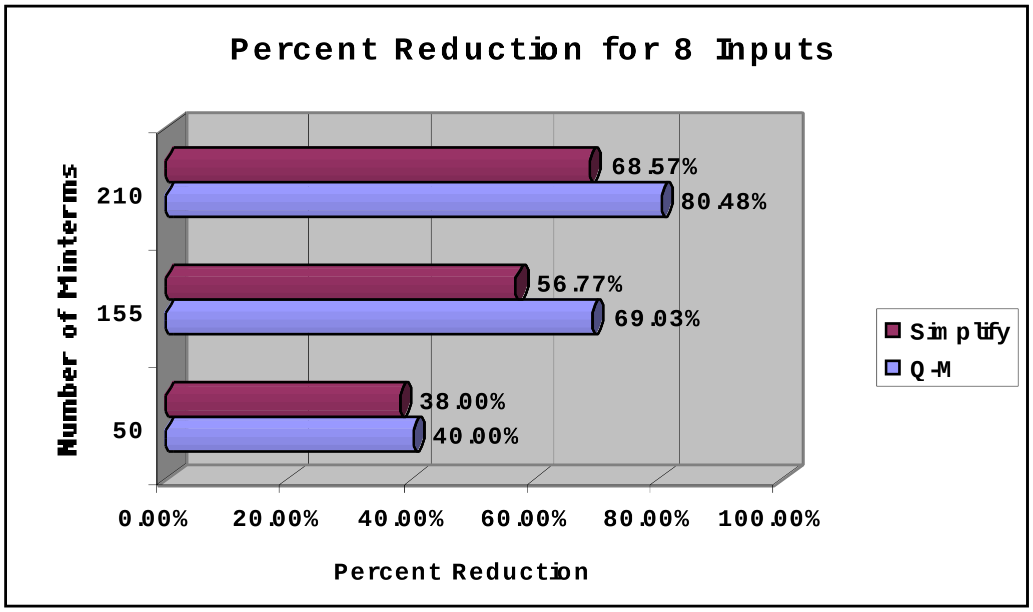
* PowerTower 180 with 64MB RAM running Mac OS 8.5.1 and MRJ 2.0. 5MB of RAM was allocated each to Simplify and Q-M as applications generated by JBindery 2.0. Each ran with extensions off and no background applications (besides the Finder) running. Three consecutive trials were taken for each combination of minterms. Times are averages; other results were constant.

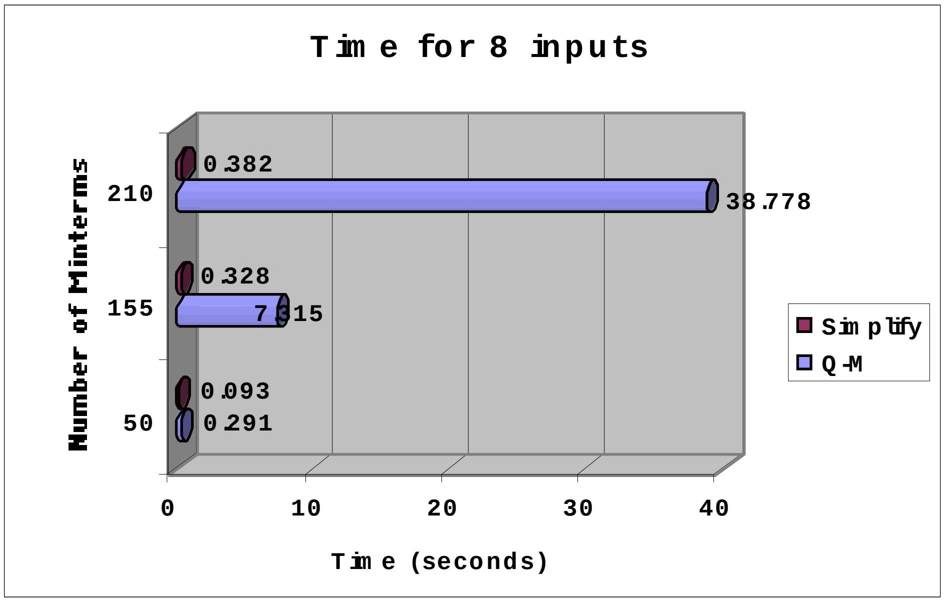

Reduction for 7 inputs
Number of Minterms
101
59
28
39
26
33
28
16
14
0
10
20
30
40
Number of Terms in Result
Simplify
Q-M

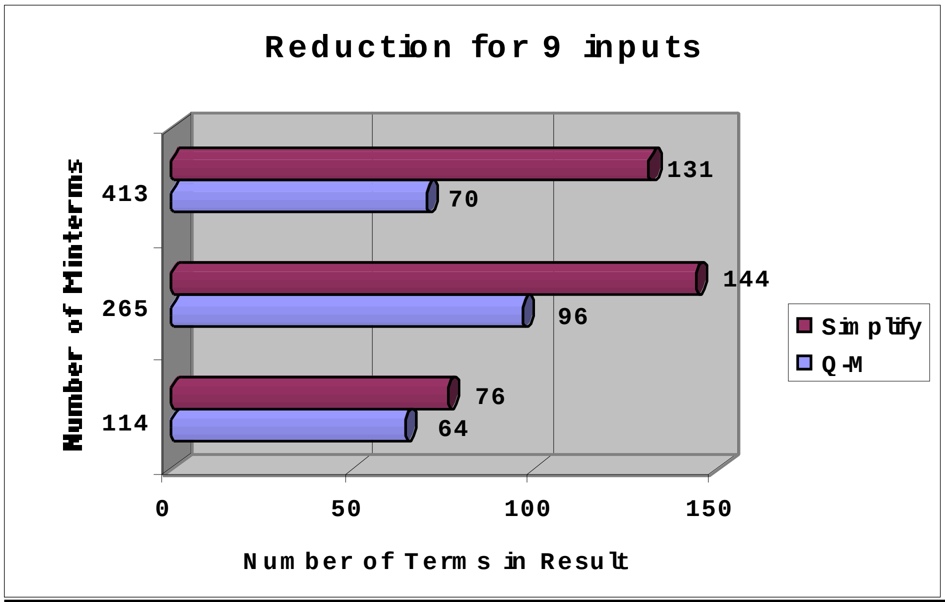

Percent Reduction for 7 Inputs
Number of Minterms
101
59
28
61.39%
74.26%
44.07%
52.54%
42.86%
50.00%
0.00%
20.00%
40.00%
60.00%
80.00%
Percent Reduction
Simplify
Q-M

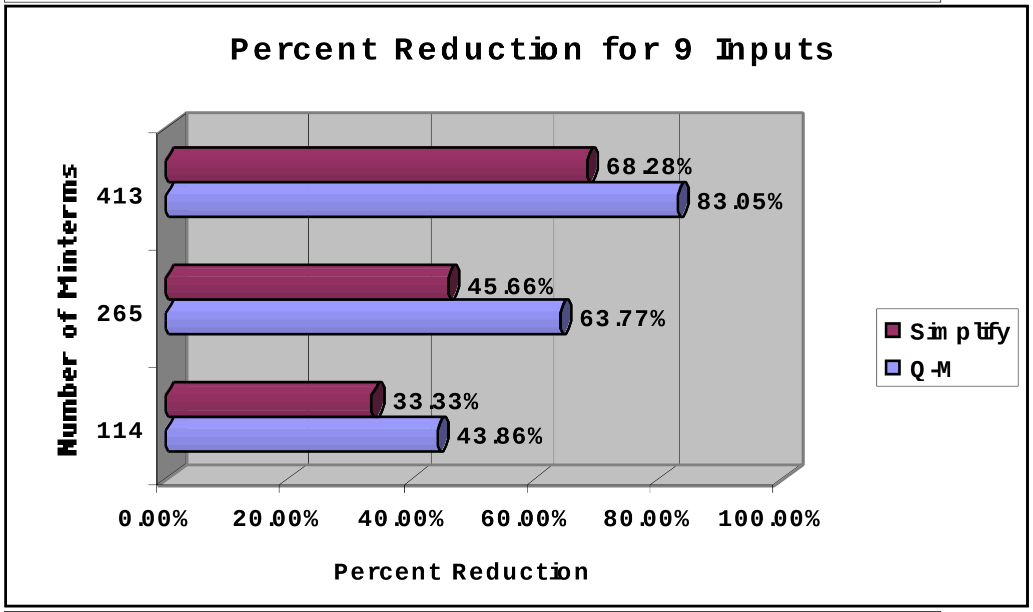

Time for 7 inputs
Number of Minterms
101
59
28
0.165
4.799
0.099
0.552
0.043
0.157
0
1
2
3
4
5
Time (seconds)
Simplify
Q-M

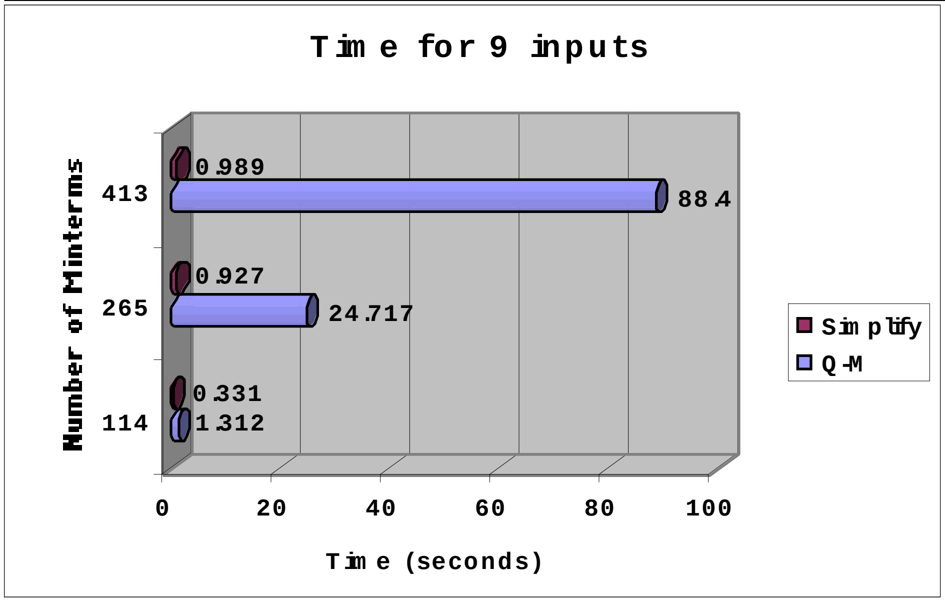

Reduction for 8 inputs
Number of Minterms
210
155
50
66
41
67
48
31
30
0
20
40
60
80
Number of Terms in Result
Simplify
Q-M

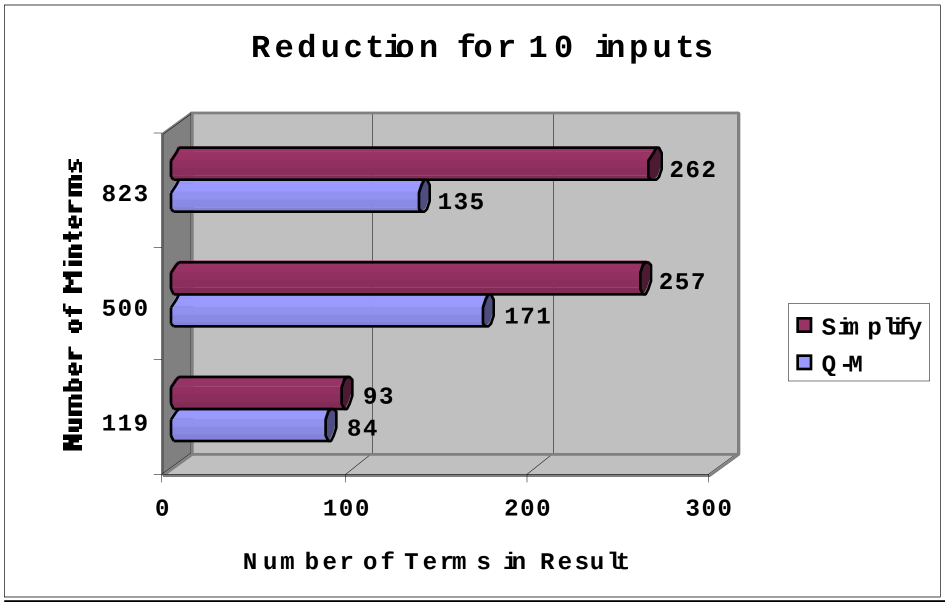

Percent Reduction for 8 Inputs
Number of Minterms
210
155
50
68.57%
80.48%
56.77%
69.03%
38.00%
40.00%
0.00%
20.00%
40.00%
60.00%
80.00%
100.00%
Percent Reduction
Simplify
Q-M

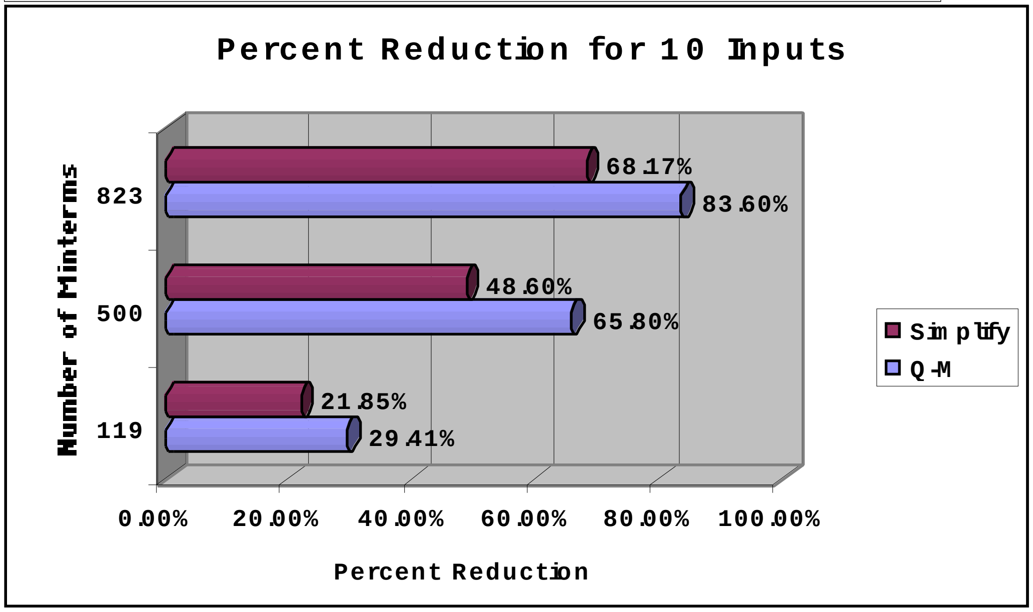

Time for 8 inputs
Number of Minterms
210
155
50
0.382
38.778
0.328
7.615
0.093
0.291
0
10
20
30
40
Time (seconds)
Simplify
Q-M

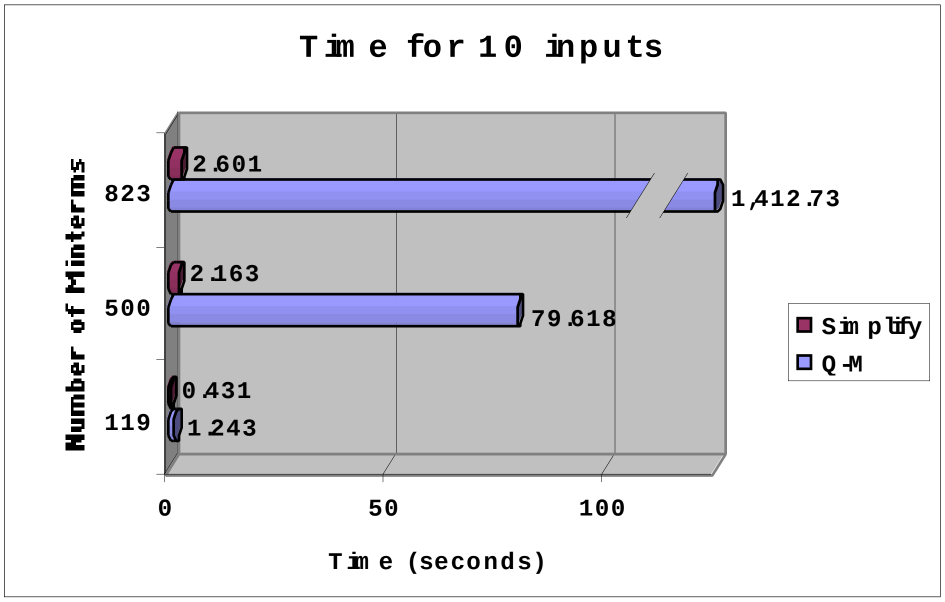

Reduction for 9 inputs
Number of Minterms
413
265
114
131
70
144
96
76
64
0
50
100
150
Number of Terms in Result
Simplify
Q-M


Percent Reduction for 9 Inputs
Number of Minterms
413
265
114
68.28%
83.05%
45.66%
63.77%
33.33%
43.86%
0.00%
20.00%
40.00%
60.00%
80.00%
100.00%
Percent Reduction
Simplify
Q-M


Time for 9 inputs
Number of Minterms
413
265
114
0.989
88.4
0.927
24.717
0.331
1.312
0
20
40
60
80
100
Time (seconds)
Simplify
Q-M


Reduction for 10 inputs
823
500
119
262
135
257
171
93
84
0
100
200
300
Number of Terms in Result
Number of Minterms
Simplify
Q-M


Percent Reduction for 10 Inputs
823
500
119
68.17%
83.60%
48.60%
65.80%
21.85%
29.41%
0.00%
20.00%
40.00%
60.00%
80.00%
100.00%
Percent Reduction
Number of Minterms
Simplify
Q-M


Time for 10 inputs
823
500
119
2.601
1,412.73
2.163
79.618
0.431
1.243
0
50
100
Time (seconds)
Number of Minterms
Simplify
Q-M

The Espresso-II algorithm performs the following steps:

1. Complement: Compute the complement of the given function (it will be needed in the rest of the algorithm).
2. Expand: Make each implicant of the function contain as many don’t-cares as possible without covering the complement of the function.
3. Essential Primes: Find the essential primes and make a list of them.
4. Irredundant: Remove redundancies from the current cover
5. Reduce: Put 1s and 0s back into the 2s of the expanded implicants. The benefit of this is that implicants with fewer 2s can be expanded in more directions when the loop is iterated again.
6. Repeat steps 2-5 until there is no improvement.
7. Lastgasp: Try the same process in a slightly different way. If there is improvement, repeat.
8. Makesparse: After the algorithm has completed, different parts of the function may be in different places. Put together the function and simplify it as much as possible.[v]

Unfortunately, this is a very intricate and technical algorithm, and I did not have enough time to research it past the Expand step.

In the amount of time I had to complete this paper, I was not able to accomplish several things. I was not able to explore the second Q-M algorithm enough to find the algorithm that guarantees a minimum cover. Also, I was not able to find out enough about the Espresso algorithm in order to program it. If I did have enough time to program the Espresso algorithm, I would compare it with the Q-M and the Simplify algorithms.

Since I was able to obtain a program of Espresso written in native Macintosh code (this makes it faster than programs in Java), MacEspresso 1.0, I am able to tell that Espresso normally produces results with more literals than the results of the Q-M algorithms, and fewer literals than the results of the Simplify algorithm. However, I cannot tell how it compares in terms of time. I believe that it takes significantly less time than the Q-M algorithms and more time than Simplify.

Notes

[i] Proofs of Absorption and Adsorption from Jerry D. Daniels, *Digital Design from Zero to One*, New York: John Wiley & Sons, Inc., 1996 p. 103

[ii] *Digital Design from Zero to One* p. 178

[iii] Simplify adapted from Robert K. Brayton, Gary D. Hachtel, Curtis T. McMullen, Alberto L. Sangiovanni-Vincentielli, *Logic Minimization Algorithms for VLSI Synthesis*, Boston: Kluwer Academic Publishers, 1997

[iv] *Logic Minimization Algorithms for VLSI Synthesis* p. 38

[v] *Logic Minimization Algorithms for VLSI Synthesis* p. 13

Glossary

absorption: the theorem stating $X + XY = X$ and $X(X+Y) = X$

adjacency: the theorem stating $XY + \overline{X}Y = Y$ and $(X+Y)(\overline{X}+Y) = Y$

adsorption: the theorem stating $X + \overline{X}Y = X + Y$ and $X(\overline{X}+Y) = XY$

algebra, boolean: a way to manipulate boolean expressions using theorems, axioms, and definitions.

binate function: a function that is not unate. This can also be used to mean a cover of a function that is not unate.

canonical cover/solution: the SOP cover of a function that contains only minterms, and thus has not at all been reduced.

cofactor: the operation used by Simplify to break up and merge a function. See pp. 26-28.

cube: a one-dimensional matrix in the form of an implicant.

De Morgan's Laws: two laws used to evaluate the complement of a function. They state $\overline{A+B+C+\ldots} = \overline{A}\cdot\overline{B}\cdot\overline{C}\cdot\ldots$ and $\overline{A\cdot B\cdot C\cdot\ldots} = \overline{A}+\overline{B}+\overline{C}+\ldots$

duality: the property of identities stating that the dual of any identity is the identity with all ANDs switched to ORs, ORs switched to ANDs, 1s switched to 0s, and 0s switched to 1s.

Espresso algorithm: an algorithm that minimizes SOP functions.

essential prime implicant: a prime implicant that the cover of a function must contain in order to cover the function

function, boolean: a function that takes boolean variables as inputs and creates a boolean output dependant on the inputs

implicant: an ANDed string of literals. It is a term in an SOP function.

Karnaugh map (K-map): a graphical representation of a function that makes it easy to apply adjacency

Law of Consensus: the theorem stating $AB + \overline{A}C + BC = AB + \overline{A}C$. It is used to eliminate implicants in a Karnaugh map.

literal: an instance of a boolean variable. It may be the variable complemented, uncomplemented, or ignored (don’t-care). In matrix representations or the Q-M algorithm, it may have a value of 0, 1, or 2/X, corresponding to complemented, uncomplemented, and don’t-care, respectively.

matrix representation of a function or implicant: The rows of a two-dimensional matrix representation of a function are the implicants of the function. The columns of a one-dimensional matrix representation of an implicant are the literals of the implicant.

minterm: an implicant that contains exactly one literal for each variable. It is not at all simplified.

monotone decreasing: A function is monotone decreasing in a variable if changing the value of the variable from 0 to 1 results in the output of the function being 0.

monotone increasing: A function is monotone increasing in a variable if changing the value of the variable from 0 to 1 results in the output of the function being 1.

prime implicant: an implicant that cannot be further reduced by adjacency

primitive function: a function that takes one or two parameters and returns one output. The most common primitive functions are NOT, AND, OR, XOR, NAND, NOR, and XNOR.

Quine-McCluskey (Q-M) algorithms: two algorithms that minimize a boolean function. The first algorithm finds all prime implicants, and the second algorithm eliminates nonessential prime implicants.

Shannon's Expansion Theorem: the theorem stating $f(x_1, x_2, \ldots, x_n) = x_1 \cdot f(1, x_2, \ldots, x_n) + \overline{x_1} \cdot f(0, x_2, \ldots, x_n)$. It also has another, logically equivalent form that can only be applied to SOP functions: $f = x_i f_{x_i} + \overline{x_i} f_{\overline{x_i}}$. It is used, among other things, to break up and merge a function in the Simplify algorithm.

Simplify algorithm: an algorithm that minimizes a boolean function. It does this by recursively breaking up a function with Shannon's Expansion Theorem until it is easy to simplify.

sum of products (SOP): a form of a function consisting of two levels of operations: First, combinations of literals are ANDed together, forming terms. Then, these terms are ORed together. All boolean functions can be expressed as SOP functions.

unate function: a function that is either monotone increasing or monotone decreasing in all of its variables.

variable, boolean: a variable that can be in one of two states: 0 or 1.

Bibliography

Brayton, Robert K., Hachtel, Gary D., McMullen, Curtis T., Sangiovanni-Vincentelli, Alberto L., *Logic Minimization Algorithms for VLSI Synthesis*, Boston: Kluwer Academic Publishers, 1997

Daniels, Jerry D., *Digital Design from Zero to One*, New York: John Wiley & Sons, Inc., 1996

Fridberg, Mikhail, MacEspresso 1.0, 1996 [computer program

Appendix A

Java Source Code of the Qune-McCluskey Algorithms

```
// global variables:
int NUMINPUTS; // the number of inputs
int NUMTERMS; // the number of minterms

/*
 * irredundant removes duplicate terms from the parameter “terms”
/*
void irredundant( Vector terms ) {
        for (int term = terms.size() - 1; term > 0; term--) {
                for (int check = term - 1; check >= 0; check--) {
                        boolean duplicate = true;
                        for (int literalIndex = 0; literalIndex < NUMINPUTS; literalIndex++) {
                                if ( ((int[]) terms.elementAt(term))[literalIndex]
                                     != ((int[]) terms.elementAt(check))[literalIndex] ) {
                                        // a literal is different: term and check are not duplicates
                                        duplicate = false;
                                        break;
                                }
                        }
                        if (duplicate) {
                                terms.removeElementAt(term);
                                break;
                        }
                }
        }
}

/*
 * The first Quine-McCluskey algorithm:
 * data is a vector containing the first level
 * of terms (the minterms). The structure of
 * data is a 3-dimensional vector/array combination
```

Appendix A
Java Source Code of the Qune-McCluskey Algorithms
A1

```
 * where the first “tier” is a vector of terms.
 * Each term in the first “tier” holds an array of literals.
 * QM1 returns a vector of terms containing arrays of literals.
 */
Vector QM1( Vector data ) {
       // initialize variables:
       // initialize variables
       int diff = -1; /* the index of the literal that the terms being compared have a difference at.
                        * diff is -1 if there is no difference.
                        */
       boolean termIsPrime = false; // is the term “term” (loop index) prime?
       int nextLevelTermCount; /* the number of terms in the next “level” of data - 1.
                                  * corresponds to the index of the last term in the next level.
                                  */
       Vector primes = new Vector(); // a vector containing all prime implicants. It will be returned at the end.

       /* repeat for each level:
        * start on the first level to create the second level, then
        * go to the second level to create the third level, and so on.
        * when there is no more reduction possible, break.
        */
       for (int level = 0; level <= NUMINPUTS; level++) {
              // initialize variables:
              nextLevelTermCount = -1;
              irredundant( (Vector) data.elementAt(level) ); // remove duplicates from the list of terms
              data.addElement(new Vector()); // initialize the next level
              Hashtable nonprimes = new Hashtable(); /* a hashtable containing the indeces of the terms
                                                      * that are not prime. It will be used to find
                                                      * the terms that are prime.
                                                      */
              // 2 nested loops to compare a term term with a term compterm to see if they can be combined
              for (int term = 0; term < ((Vector) data.elementAt(level)).size() - 1; term++) {
                     for (int compterm = term + 1; compterm < ((Vector) data.elementAt(level)).size(); compterm++) {
                            // check literals in term and compterm to ultimately see if the terms can be combined
                            for (int literal = 0; literal < NUMINPUTS; literal++) {
```

```
            if ( ((int[]) ((Vector) data.elementAt(level)).elementAt(term))[literal]
                   != ((int[]) ((Vector) data.elementAt(level)).elementAt(compterm))[literal] ) {
                    // there is a difference between literals
                    if (diff == -1) { //there was no previous difference
                        diff = literal;
                    }
                    else { //there was a previous difference
                        diff = -1;
                        break; //there is more than one difference: get out of this comparison
                    }
            }
        }// end of literal for loop
        if (diff != -1) { // term and compterm can be combined
            nonprimes.put( new Integer(term), new Integer(term) ); // term is not prime
            nonprimes.put( new Integer(compterm), new Integer(term) ); // compterm is not prime
                // initialize a space to put the combined term in the next level
            ((Vector) data.elementAt(level + 1)).addElement(new int[NUMINPUTS]);
            nextLevelTermCount++; // we are adding a term to the next level
            System.arraycopy( (int[]) ((Vector) data.elementAt(level)).elementAt(term), 0,
                        (int[]) ((Vector) data.elementAt(level + 1)).elementAt(nextLevelTermCount), 0,
                        NUMINPUTS);
                // the different literal is a don't care:
            ((int[]) ((Vector) data.elementAt(level + 1)).elementAt(nextLevelTermCount))[diff] = 2;
            diff = -1; // initialize diff for the next comparison
        }
    }// end of compterm loop
}// end of term loop

// check to see if there were any primes:
for ( int i = 0; i < ((Vector) data.elementAt(level)).size(); i++ ) {
    if (!nonprimes.containsKey(new Integer(i))) {
        // the i_th term of this level is prime
        primes.addElement(new int[NUMINPUTS]);
        System.arraycopy( (int[]) ((Vector) data.elementAt(level)).elementAt(i), 0,
                        (int[]) primes.elementAt( primes.size()-1 ), 0, NUMINPUTS);
                }
            }
            if ( nonprimes.size() == 0 ) {
                // there was no reduction in this level
                break;
            }
    }// end of level loop

    return primes;
}


/*
 * The second Quine-McCluskey algorithm
 * minterms is a vector containing the minterms.
 * Each minterm is an array of 0 to NUMINPUTS – 1 literals.
 * implicants is a vector containing the implicants.
 * Each implicant is an array of 0 to NUMINPUTS – 1 literals.
 */
Vector QM2( Vector minterms, Vector implicants ) {
    if ( implicants.size() > 1 ) {
        /* Create a 2D array “checkMap” containing the checks in the table.
         * The first index is the minterm (column) and the second is the implicant (row)
         */
        boolean checkMap[][] = new boolean[minterms.size()][implicants.size()];
        // put "checks" into checkMap: checks have a value of true
        for (int implicant = 0; implicant < implicants.size(); implicant++) {
            for (int minterm = 0; minterm < minterms.size(); minterm++) {
                boolean impliesMinterm = true; // does the implicant imply the minterm? Init. to true.
                for (int literal = 0; literal < NUMINPUTS; literal++) {
                    if ( !(( ((int[]) implicants.elementAt(implicant))[literal]
                            == ((int[]) minterms.elementAt(minterm))[literal] )
                            || ((int[]) implicants.elementAt(implicant))[literal] == 2 ) ) {
                        // implicant does not imply minterm
                        impliesMinterm = false;
                    break; // We know the minterm is not implied: get out of the comparison.
                }
            }
            checkMap[minterm][implicant] = impliesMinterm; // put the check (or no check) into checkMap
        }
    }
    Vector essentialPrimes = new Vector(); // essential primes: to be returned
    boolean[] mintermsDone = new boolean[minterms.size()]; // what minterms are "circled"
    // initialize mintermsDone to all false
    for (int initMintermsDone = 0; initMintermsDone < minterms.size(); initMintermsDone++) {
        mintermsDone[initMintermsDone] = false;
    }
    // Checking columns and taking rows (implicants):
    while ( !allTermsDone( mintermsDone ) ) {
        // do columns in order of how many checks they have (least first)
        // find the col. with the least number of checks:
        int minChecksCol = 0;
        int minChecksVal = implicants.size() + 1;
        for (int col = 0; col < minterms.size(); col++) {
            int numChecksInCol = 0;
            if ( !mintermsDone[col] ) { // don’t check any minterms that are circled
                for (int row = 0; row < implicants.size(); row++) {
                    if (checkMap[col][row]) {
                        // there is a check. Add 1 to the number of checks in col.
                        numChecksInCol++;
                    }
                }
                if ( numChecksInCol < minChecksVal ) {
                    // There are fewer checks in this column than the current choice.
                    minChecksVal = numChecksInCol;
                    minChecksCol = col;
                }
            }
        }
        // find row with most additional checks from col minChecksCol
        int maxChecksRow = 0;
        int maxChecksVal = 0;
        for (int row = 0; row < implicants.size(); row++) {
            if ( checkMap[minChecksCol][row] ) {
                // There is a check in this row that is in minChecksCol.
                // Count the number of additional checks in this row.
                int checksInRow = 0;
                for (int col = 0; col < minterms.size(); col++) {
                    if ( checkMap[col][row] && !mintermsDone[col] ) {
                        // There is a check, and its minterm is not circled.
                        checksInRow++;
                    }
                }
                if ( checksInRow > maxChecksVal ) {
                    // There are more additional checks in this row than the current choice.
                    maxChecksVal = checksInRow;
                    maxChecksRow = row;
                }
            }
        }
        // Circle all minterms with checks in maxChecksRow:
        for ( int col = 0; col < minterms.size(); col++ ) {
            if ( checkMap[col][maxChecksRow] ) {
                mintermsDone[col] = true;
            }
        }
        // The implicant at maxChecksRow is essential. Take it:
        essentialPrimes.addElement(new int[NUMINPUTS]);
        System.arraycopy( (int[]) implicants.elementAt( maxChecksRow ), 0,
                    (int[]) essentialPrimes.elementAt( essentialPrimes.size() - 1 ), 0, NUMINPUTS );
    }
    return essentialPrimes; // done. Return essentialPrimes.
}
else {
    // There is only one implicant. No elimination is possible.
```

```
                return implicants;
        }
}

/*
 * allTermsDone checks to see if all the minterms in QM2 are done.
 */
boolean allTermsDone( boolean[] mintermsDoneArray ) {
        for ( int ix = 0; ix < NUMTERMS; ix++) {
                if ( mintermsDoneArray[ix] == false ) {
                        return false;
                }
        }
        return true;
}
```

Appendix B

Java Source Code of Simplify

```
// global variables:
int NUMINPUTS; // the number of inputs
int NUMTERMS; // the number of minterms

/*
 * The main meathod for simplify. Simplify breaks up
 * a function f recursively to simplify it.
 * f is a vector of implicants. It is a representation
 * of a function. Each implicant (or “cube”) is an
 * array of literals.
 */
Vector simplify( Vector f ) {

	if ( unate(f) ) {
		// f is unate. Use unate_simplify:
		return unate_simplify( f );
	}
	else {
		// f is binate.
		int j = binate_select(f); // j is the splitting variable
		if ( j == -1 ) { // check the error flag
			// some error occurred in binate_select. f is unate? use
			// unate_simplify to see if any containment removal can occur.
			return unate_simplify( f );
		}
		else {
			// no error occurred. Use Shannon’s Expansion Theorem:
			Vector f_prime = merge( simplify( cofactor(f,j,true) ),
				simplify( cofactor(f,j,false) ), j );
			// double check to make sure f actually was simplified:
			if ( f.size() > f_prime.size() ) {
				return f_prime;
			}
			else {
				return f;
			}
		}
	}
}

/*
 * unate_simplify removes any implicant in f that
 * is contained by another implicant in f.
 */
Vector unate_simplify( Vector f ) {
	for ( int cube = 0; cube < f.size() - 1; cube++ ) {
		for ( int compCube = f.size() - 1; compCube > cube; compCube-- ) {
			if ( contains( (int[]) f.elementAt(cube), (int[])
				f.elementAt(compCube) ) ) {
```

```
                        // compCube is contained in cube.
                        f.removeElementAt( compCube );
                    }
                }
            }
            return f;
        }

        /*
         * c and d are cubes (implicants). contains
         * returns true if c contains (covers) d and
         * false if c does not contain d.
         */
        boolean contains( int[] c, int[] d ) {
            boolean toReturn = true; // init toReturn to true.
            for ( int test = 0; test < NUMINPUTS; test++ ) {
                if ( (c[test] != d[test]) && (c[test] != 2) ) {
                    // literals disagree and the literal in c is not 2.
                    // c does not contain d.
                    toReturn = false;
                    break;
                }
            }
            return toReturn;
        }

        /*
         * binate_select returns the index of the most binate
         * column in f. The most binate column is the one with
         * the greatest number of ones and zeros.
         */
        int binate_select( Vector f ) {
            // finds the most binate variable (column) for splitting
            int[] p_zero = new int[NUMINPUTS]; // array holding the number of 0s in each col
            int[] p_one = new int[NUMINPUTS]; // array holding the number of 1s in each col
            // count the number of 0s and 1s in each row:
            for ( int j = 0; j < NUMINPUTS; j++ ) { // j is index for columns
                for ( int i = 0; i < f.size(); i++ ) { // i is index for rows
                    if ( ((int[]) f.elementAt(i))[j] == 0 ) {
                        p_zero[j]++;
                    }
                    else {
                        if ( ((int[]) f.elementAt(i))[j] == 1 ) {
                            p_one[j]++;
                        }
                    }
                }
            }
            boolean unate = true; // make sure f is not unate. Init value of unate is true.
            Hashtable J = new Hashtable(); // the set of binate columns.
            for ( int j = 0; j < NUMINPUTS; j++ ) {
                if ( (p_zero[j] > 0) && (p_one[j] > 0) ) {
                    // there are both 0s and 1s in this column.
                    unate = false;
                    J.put(new Integer(j), new Integer(j));
                }
            }
            if ( unate ) {
                return -1;
            }
            else {
                int splittingJ = -1; // the choice of the most binate column.
                int maxVal = 0; // the maximum value of the sum of 1s and 0s in a column
                for ( int j = 0; j < NUMINPUTS; j++ ) {
                    if ( p_zero[j] + p_one[j] > maxVal && J.containsKey(new Integer(j))){
                        // There are more 1s and 0s in this column than the chosen col.
                        // Also, this column is binate. It becomes the new chosen col.
                        maxVal = p_zero[j] + p_one[j];
                        splittingJ = j;
                    }
                }
                return splittingJ;
            }
        }

        /*
         * merge merges the two functions (sets of implicants) H1 and H0.
         * H1 is the “subfunction” gotten from using Shannon’s Expansion
         * Theorem with the splitting variable splittingVar = 1.
         * H0 is the “subfunction” gotten from using Shannon’s Expansion
         * Theorem with the splitting variable splittingVar = 0.
         * merge ANDs the splitting var with H1 and H0, and then ORs the
         * result to complete Shannon’s Expansion Theorem. While it is
         * doing this, it checks to see if it can create a more simple
         * result by checking for identical implicants and implicants
         * that contain other implicants between  H1 and H0.
         */
        Vector merge( Vector H1, Vector H0, int splittingVar ) {
            // H2 is the set of implicants that are not ANDed with the splitting variable
            Vector H2 = new Vector();
            // check to see if any implicants in H1 and H0 are identical:
            for ( int i = H0.size() - 1; i >= 0; i-- ) {
                for ( int l = H1.size() - 1; l >= 0; l-- ) {
                    boolean equal = true; // are the implicants in H1 and H0 identical?
                    // compare by checking each literal:
                    for ( int compare = 0; compare < NUMINPUTS; compare++ ) {
                        if ( ((int[]) H0.elementAt(i))[compare] != ((int[])
                        H1.elementAt(l))[compare] ) {
                            // the implicants are not identical.
                            equal = false;
                            break; // get out of this comparison.
                        }
                    }
                    if (equal) {
                        // they are identical, and thus do not have to be
                        // ANDed with the splitting variable.
```

```
                        H2.addElement(((int[]) H0.elementAt(i)).clone());
                        H0.removeElementAt(i);
                        H1.removeElementAt(l);
                        break;
                }
        }
}
// check to see if any implicants in H1 contain
// any implicants in H0 and vice versa.
for ( int i = H0.size() - 1; i >= 0; i-- ) {
        for ( int l = H1.size() - 1; l >= 0; l-- ) {
                if ( contains((int[]) H0.elementAt(i), (int[]) H1.elementAt(l)) ) {
                        // The implicant in H0 contains the one in H1.
                        // The one in H1 does not have to be ANDed with splittingVar.
                        H2.addElement( H1.elementAt(l) );
                        H1.removeElementAt(l);
                }
                else {
                        if ( contains((int[]) H1.elementAt(l),
                                (int[]) H0.elementAt(i)) ) {
                                // The implicant in H1 contains the one in H0.
                                // The one in H0 does not have to be ANDed
                                // with splittingVar.
                                H2.addElement( H0.elementAt(i) );
                                H0.removeElementAt(i);
                                break;
                        }
                }
        }
}
try {
        Vector H = new Vector(); // H is the final function to be returned.
        // AND all implicants in H0 with the splitting var = 0:
        H0 = intersect( H0, splittingVar, false );
        // Put everything in H0 into H:
        for ( int i = 0; i < H0.size(); i++ ) {
                H.addElement(((int[]) H0.elementAt(i)).clone());
        }
        // And all implicants in H1 with the splitting var = 1:
        H1 = intersect( H1, splittingVar, true );
        // Put everything in H1 into H:
        for ( int l = 0; l < H1.size(); l++ ) {
                H.addElement(((int[]) H1.elementAt(l)).clone());
        }
        // Put everything in H2 into H:
        for ( int t = 0; t < H2.size(); t++ )
                H.addElement(H2.elementAt(t));
        return H;
} catch (CloneNotSupportedException ex) {
        // this will never happen, but he compiler wants it.
        System.out.println(ex);
        return null;
}
```

```
}

/*
 * intersect ANDs a function f with a splitting variable.
 * It ANDs each implicant in f with the “implicant” that is
 * all 2s except for the index of splittingVar, which is
 * 0 when value is false and 1 when value is true.
 * Effectively, this makes a function whose value in
 * the splittingVar column is 1 or 0 (depending on what
 * value is). The implicants in this function come from
 * the implicants in f that do not “disagree” with the
 * splitting value: that is, the ones that do not have a
 * 0 in the splittingVar literal when splittingVar is 1,
 * and vice versa.
 */
Vector intersect( Vector f, int splittingVar, boolean value ) {
       Vector intersection = (Vector) f.clone(); // intersection will be returned.
       // remove the implicants from intersection that “disagree”
       // with the splitting variable:
       int notVal = 0, loweredVal = 1; // default (init): value is true
       /* notVal is the opposite value of value. It is 0 when value is true
        * and 1 when value is false.
        * loweredVal is the value of value. It is what the 2s in the
        * splittingVar column of intersection will be changed to.
        */
       if (!value) { // change notVal and loweredVal if value is false
              notVal = 1;
              loweredVal = 0;
       }
       for ( int cube = intersection.size() - 1; cube >= 0; cube-- ) {
              if ( ((int[]) intersection.elementAt(cube))[splittingVar] == 2 ) {
                     // AND this cube (implicant) by changing the 2 to loweredVal:
                     ((int[]) intersection.elementAt(cube))[splittingVar] = loweredVal;
              }
              else {
                     if (((int[]) intersection.elementAt(cube))[splittingVar] == notVal) {
                            // there is a “disagreement:” remove this cube (implicant)
                            intersection.removeElementAt(cube);
                     }
              }
       }
       return intersection;
}

/*
 * cofactor returns the cofactor of f with respect
 * to the splitting variable j, whose value is value.
 * value is true for uncomplemented and false for
 * complemented.
 */
Vector cofactor( Vector f, int j, boolean value ) {
       Vector result = new Vector(); // result is the cofactor; it will be returned.
       try {
            for ( int i = 0; i < f.size(); i++ ) {
                  // i is the index of implicants (rows)
                  if ( ( (((int[]) f.elementAt(i))[j] != 0) && value )
                        || ( (((int[]) f.elementAt(i))[j] != 1) && !value ) ) {
                        // Case 2 of the cofactor operation
                        int[] newCube = (int[]) ((int[]) f.elementAt(i)).clone();
                        newCube[j] = 2;
                        result.addElement(newCube);
                  }
            }
      }
      catch (CloneNotSupportedException ex) {
            // this will never happen, but the compiler wants it.
            System.out.println(ex);
            return null;
      }
      return result;
}

/*
 * unate checks to see if a function f is unate
 */
boolean unate( Vector f ) {
      boolean isUnate = true; // initialize isUnate, which will be returned.
      for ( int col = 0; ( col < NUMINPUTS ) && isUnate; col++ ) {
            // check each column to make sure it does not contain both 1s and 0s:
            int somethingInCol = 2; // if a 1 or 0 is found, it's put here.
            for ( int row = 0; row < f.size(); row++ ) {
                  if ( ( ((int[]) f.elementAt(row))[col] != somethingInCol )
                        && ( ((int[]) f.elementAt(row))[col] != 2 ) ) {
                        if (somethingInCol == 2) {
                              // only 2s were seen before in this column.
                              somethingInCol = ((int[]) f.elementAt(row))[col];
                        }
                        else {
                              // both 1s and 2s are in this column.
                              isUnate = false;
                              break;
                        }
                  }
            }
      }
      return isUnate;
}
```