12 Boolean Algebra

Introduction

In this chapter we will use *Mathematica* to model Boolean algebra. In the first section, we demonstrate the basic functions that will be used in this chapter. In the second section, we will focus on the disjunctive normal form of a logical expression. We will see how to use *Mathematica*'s functions for finding a disjunctive normal form expression for a Boolean function and for finding a representation for a function defined by a table of values. In Section 3, we will see how *Mathematica* can be used to model logical circuits, including how to go about transforming a circuit diagram into a *Mathematica* expression. We also provide a function that will transform a logical expression into a model of a circuit. In the final section of the chapter, we consider simplification of logical expressions, and we develop an implementation of the Quine-McCluskey method.

12.1 Boolean Functions

In this section we will see how to work with Boolean expressions and how to create Boolean functions. We will also use *Mathematica* to verify identities in Boolean algebra and to compute the dual of an expression.

Preliminaries

In Chapter 1 of this manual, we discussed *Mathematica*'s logical expressions. The Boolean values true and false are represented by the symbols `True` and `False`. To *Mathematica*, these are constant values, like the numbers **2** or `Pi`.

We also saw in Chapter 1 the logical operators `And` (**&&**), `Or` (**||**), and `Not` (**!**). These can be used in functional form or as operators. The two expressions below both compute $T \wedge \neg (T \vee F)$.

```
In[1]:= True && ! (True || False)

Out[1]= False

In[2]:= And[True, Not[Or[True, False]]]

Out[2]= False
```

Note that *Mathematica* obeys the usual order of precedence for logical operators, namely negation followed by conjunction, then disjunction, and finally implication.

Other logical operators supported by *Mathematica* include: the exclusive or, `Xor`, implication, `Implies`, and the biconditional, `Equivalent`. Each logical connective can be entered as the usual mathematical symbol by using an escape sequence, as shown in the table below.

And	ESCandESC	⋀
Or	ESCorESC	⋁
Not	ESCnotESC	¬
Xor	ESCxorESC	⊻
Implies	ESC=>ESC	⇒
Equivalent	ESCequivESC	⇔

In this manual, we will typically enter operators using either the simple keyboard character operators for `And` (`&&`), `Or` (`||`), and `Not` (`!`) or in functional form, rather than using the escape sequences.

For Boolean algebra, the textbook uses the objects 0 and 1 with operators +, ·, and ¯ instead of their logical counterparts. It is tempting to use *Mathematica*'s bit functions, `BitAnd`, `BitOr`, and `BitNot`, in order to replicate the 0-1 form of Boolean expressions. However, the bit functions behave differently than the corresponding Boolean operators would, in particular `BitNot` does not switch between 0 and 1.

```
In[3]:=  BitNot[0]

Out[3]=  -1

In[4]:=  BitNot[1]

Out[4]=  -2
```

In the remainder of this manual, we will stick to the logical forms of Boolean expressions. However, some readers may be interested to know that it is possible to create operators to mirror the kinds of Boolean algebra expressions used in the text. To do so, we use symbols that *Mathematica* will interpret as operators but which have no built-in definition. For example, we could use circle times (⊗, entered ESCc*ESC) and circle plus (⊕, entered ESCc+ESC) for the **and** and **or** operators, and the unary minus-plus (∓, entered ESC-+ESC) for negation. Then the expression $1 \cdot 0 + \overline{(0 + 1)}$ would be entered as shown below.

```
In[5]:=  1⊗0⊕∓(0⊕1)

Out[5]=  1 ⊗ 0 ⊕ ∓ (0 ⊕ 1)
```

To get *Mathematica* to evaluate such expressions properly, you just need to make definitions to the symbols. By setting the `Flat` and `Listable` attributes first, the operator will be associative. For example, ⊗ can be defined by setting values for `CircleTimes`.

```
In[6]:=  SetAttributes[CircleTimes, {Flat, Listable}];
         CircleTimes[1, 1] = 1;
         CircleTimes[1, 0] = 0;
         CircleTimes[0, 1] = 0;
         CircleTimes[0, 0] = 0;
```

Now *Mathematica* will automatically simplify ⊗, so if we enter the previous expression again:

```
In[11]:=  1⊗0⊕∓(0⊕1)

Out[11]=  0 ⊕ ∓ (0 ⊕ 1)
```

Definitions of the other operations are left to the interested reader. In this manual, we will not use this approach, since the logical form of Boolean expressions is more naturally supported by *Mathematica*.

Boolean Expressions and Boolean Functions

Consider Example 1 from the text, which asks that we compute the value of $1 \cdot 0 + \overline{(0+1)}$. To perform this computation in *Mathematica*, we first translate it into a logical statement. We do this by changing 1 into `True`, 0 into `False`, the multiplication into `And` (**&&**), the addition into `Or` (**||**), and the bar into `Not` (**!**).

```
In[12]:= True && False || ! (False || True)

Out[12]= False
```

Of course, you can enter Boolean expressions involving variables, assuming the symbols have not previously been assigned values.

```
In[13]:= Implies[p && q, r]

Out[13]= p && q ⇒ r
```

And, just as with arithmetic expressions, you can evaluate these expressions for specific values by applying the `ReplaceAll` (**/.**) operator.

```
In[14]:= Implies[p && q, r] /. {p → True, q → True, r → False}

Out[14]= False
```

Representing Boolean Functions

You define a Boolean function in *Mathematica* in the same way as any other function.

Consider, for example, the Boolean function $f(x, y, z) = x\,y + y\,z + z\,x$ (written in the 0-1 notation).

This can be modeled in *Mathematica* by the function defined below.

```
In[15]:= f[x_, y_, z_] := Or[And[x, y], And[y, z], And[z, x]]
```

You can work with **f** in the usual way. The following applies **f** to **p**, **q**, and **r**.

```
In[16]:= f[p, q, r]

Out[16]= (p && q) || (q && r) || (r && p)
```

When **f** is applied to truth values, it is evaluated.

```
In[17]:= f[True, False, True]

Out[17]= True
```

You can also mix truth values and symbols. In this case, *Mathematica* will simplify the expression, given the partial information.

```
In[18]:= f[True, q, r]

Out[18]= q || (q && r) || r
```

You may notice that this expression is logically equivalent to $q \vee r$. Applying the `Simplify` function will ask *Mathematica* to more fully simplify the output.

```
In[19]:= f[True, q, r] // Simplify

Out[19]= q || r
```

Values of Boolean Functions

Examples 4 and 5 of Section 12.1 illustrate how the values of a Boolean function, in the 0-1 format, can be displayed in a table. In the logical form, this is equivalent to a truth table for the Boolean function. In Chapter 1 of this manual, we illustrated the use of the `BooleanTable` function for creating truth tables.

Recall that `BooleanTable` accepts two arguments: a Boolean expression and a list of the variables. The output is a list of the truth values for the expression obtained by substituting every possible combination of truth values into the variables.

For example, we will display the table of values for the Boolean function **f** defined above. The first argument to `BooleanTable` will be a list containing the three variables and the function applied to them. Giving the first argument as this list means that the output will indicate the values of the individual variables, and not just the result. The second argument will be the list of variables.

```
In[20]:= BooleanTable[{p, q, r, f[p, q, r]}, {p, q, r}] // TableForm

Out[20]//TableForm=
         True     True     True     True
         True     True     False    True
         True     False    True     True
         True     False    False    False
         False    True     True     True
         False    True     False    False
         False    False    True     False
         False    False    False    False
```

If you wish, you can use this function to produce output in the 0-1 form by applying the `Boole` function. `Boole` is a built-in function that transforms the truth values `True` and `False` into the values 1 and 0. Since it threads over lists, it can be applied to the output from `BooleanTable`.

```
In[21]:= Boole[BooleanTable[{p, q, r, f[p, q, r]}, {p, q, r}]] // TableForm

Out[21]//TableForm=
         1    1    1    1
         1    1    0    1
         1    0    1    1
         1    0    0    0
         0    1    1    1
         0    1    0    0
         0    0    1    0
         0    0    0    0
```

We can further refine the output by using the `TableHeadings` option for `TableForm`. `TableHeadings` is assigned to a pair representing the row and column headings, with `None` used when a group of headings is not wanted.

```
In[22]:= TableForm[Boole[BooleanTable[{p, q, r, f[p, q, r]}, {p, q, r}]],
          TableHeadings → {None, {p, q, r, f}}]

Out[22]//TableForm=
         p    q    r    f
         1    1    1    1
         1    1    0    1
         1    0    1    1
         1    0    0    0
         0    1    1    1
         0    1    0    0
         0    0    1    0
         0    0    0    0
```

Operations on Boolean Functions

As with functions on real numbers, Boolean functions can be combined using basic operations. The complement of a Boolean function and the Boolean sum and product of functions are defined in the text.

To compute complements, sums, and products of Boolean functions, you must define a new function in terms of the original. For example, consider the function $G(x, y) = x \cdot y$. In logical notation, this is $G(x, y) = x \bigwedge y$.

```
In[23]:= G[x_, y_] := x && y
```

The complement of **G**, which we'll call **notG**, is created as follows. The arguments of **notG** are the same as **G**. The formula that defines **notG** is `! G[x, y]`.

```
In[24]:= notG[x_, y_] := ! G[x, y]
```

Observe that if we evaluate **notG** at a pair of variables, *Mathematica* returns the expected result.

```
In[25]:= notG[x, y]

Out[25]= ! (x && y)
```

Let us define another function, $H(x, y) = x \cdot \overline{y}$.

```
In[26]:= H[x_, y_] := x && ! y
```

To compute the Boolean sum $G + H$, we combine the functions with the Or (||) operator. More precisely, we define a function **GpH** with the formula `G[x,y]||H[x,y]`.

```
In[27]:= GpH[x_, y_] := G[x, y] || H[x, y]
```

Applying this to a pair of variables and simplifying, we obtain the following formula for $G + H$.

```
In[28]:= GpH[x, y] // Simplify

Out[28]= x
```

This result indicates that $x \cdot y + x \cdot \overline{y} = x$. This can also be verified using the identities in Table 5 of Section 12.1.

Identities of Boolean Algebra

We can check identities, equivalence of Boolean expressions, and equality of Boolean functions using the `Equivalent` and `TautologyQ` functions.

We will use the distributive law $x(y+z) = x \cdot y + x \cdot z$ as an example. First we must translate the statement into a logical equivalence: $x \bigwedge (y \bigvee z) \equiv (x \bigwedge y) \bigvee (x \wedge z)$.

Now we will assign the expressions on either side of the equivalence to symbols. This is not necessary, but it will make later expressions easier to read.

```
In[29]:= distributiveL = x && (y || z);
         distributiveR = (x && y) || (x && z);
```

To confirm the equivalence of the two Boolean expressions, we combine them into a biconditional using the `Equivalent` function. We then apply the `TautologyQ` function to the biconditional and a list of the Boolean variables appearing in the expression.

```
In[31]:= TautologyQ[Equivalent[distributiveL, distributiveR], {x, y, z}]

Out[31]= True
```

This verifies the given distributive law.

In the case that the two expressions are not equivalent, you can use the `SatisfiabilityInstances` function to find a list of assignments of truth values to the variables in the expression that demonstrates that the expressions are not equivalent.

Consider the non-equivalence $x + x \cdot y \neq y$. In logical form, this is $x \bigvee (x \bigwedge y) \not\equiv y$. First observe that `TautologyQ` returns false.

```
In[32]:= TautologyQ[Equivalent[x || (x && y), y], {x, y}]

Out[32]= False
```

Now apply `SatisfiabilityInstances` to the negation of the equivalence.

```
In[33]:= SatisfiabilityInstances[! Equivalent[x || (x && y), y], {x, y}]

Out[33]= {{True, False}}
```

This output means that setting x equal to **true** and y equal to **false** provides a demonstration, by counterexample, that $x \bigvee (x \bigwedge y) \not\equiv y$. Indeed, substituting $x = \text{true}$ and $y = \text{false}$ on the left hand side produces true $\bigvee$ (true $\bigwedge$ false) $\equiv$ true $\bigvee$ false $\equiv$ true. That is not the same as the right hand side, y, which is assigned **false**.

Note that the output from `SatisfiabilityInstances` is a list of assignments. Ordinarily only one truth value assignment will be returned. But if you provide a positive integer as an optional third argument, *Mathematica* will attempt to find that number of different assignments. Below, we ask for three assignments, but only two exist and so two are returned.

```
In[34]:= SatisfiabilityInstances[! Equivalent[x || (x && y), y], {x, y}, 3]

Out[34]= {{True, False}, {False, True}}
```

Equality of Boolean functions can also be checked with the `Equivalent` and `TautologyQ` functions.

Consider the following Boolean functions.

$$f_1(x, y) = \overline{(x \cdot y)}$$

$$f_2(x, y) = \overline{x} + \overline{y}$$

Define the corresponding functions:

```
In[35]:= f1[x_, y_] := ! (x && y);
         f2[x_, y_] := ! x || ! y;
```

We can test the assertion that $f_1(x, y) = f_2(x, y)$ by applying the `Equivalent` and `TautologyQ` functions as shown below.

```
In[37]:= TautologyQ[Equivalent[f1[x, y], f2[x, y]], {x, y}]

Out[37]= True
```

Duality

We conclude this section by showing how *Mathematica* can be used to compute the dual of an expression. We will define a function, **dual**, to achieve this.

Recall that the dual of a Boolean expression is the expression obtained by interchanging conjunctions and disjunctions and interchanging trues and falses. We can achieve this in *Mathematica* by applying `ReplaceAll` (`/.`) with a list of rules effecting the interchanges.

```
In[38]:= dual[expr_] :=
          expr /. {And → Or, Or → And, False → True, True → False}
```

Note that this will not produce an infinite loop, since `ReplaceAll` (`/.`) operates by looking at each part of the expression only once applying the first rule in the list that is valid.

For example, consider the expression $x \cdot \overline{y} + y \cdot \overline{z} + \overline{x} \cdot z$. As a logical expression, this can be written as $(x \bigwedge \neg\, y) \bigvee (y \bigwedge \neg\, z) \bigvee (\neg\, x \wedge z)$. We calculate the dual by applying the **dual** function.

```
In[39]:= dual1 = dual[(x && ! y) || (y && ! z) || (! x && z)]

Out[39]= (x || ! y) && (y || ! z) && (! x || z)
```

Similarly, the dual of $\overline{x} \cdot y + \overline{y} \cdot z + x \cdot \overline{z}$ can be computed by

```
In[40]:= dual2 = dual[(! x && y) || (! y && z) || (x && ! z)]

Out[40]= (! x || y) && (! y || z) && (x || ! z)
```

Exercise 13 of Section 12.1 asks you to prove that the expressions $x \cdot \overline{y} + y \cdot \overline{z} + \overline{x} \cdot z$ and $\overline{x} \cdot y + \overline{y} \cdot z + x \cdot \overline{z}$ are equivalent. The duality principle implies that the duals calculated above are also equivalent. This can be verified by the `Equivalent` and `TautologyQ` functions.

```
In[41]:= TautologyQ[Equivalent[dual1, dual2], {x, y, z}]

Out[41]= True
```

12.2 Representing Boolean Functions

In this section we will see how to use *Mathematica* to express Boolean functions in the disjunctive normal form (also called sum-of-products expansion). We will first look at the *Mathematica* function for turning an expression in Boolean algebra into the disjunctive normal form. Then we will see the function for finding an expression based on a table of values.

Disjunctive Normal Form from an Expression

Given an expression written using the logical connectives, the `BooleanConvert` function can be used to transform the expression into disjunctive normal form.

Consider Example 3: $(x+y)\,\overline{z}$. In logical form, this is $(x \bigvee y) \bigwedge \neg z$. We assign this logical expression to a symbol.

```
In[42]:= example3 = (x || y) && ! z

Out[42]= (x || y) && ! z
```

The most common way to apply the `BooleanConvert` function involves two arguments: the expression to be converted and a string representing the desired form of the output. There are several possible forms, as detailed by the help page, but the two we will be using are **"DNF"** for disjunctive normal form (or equivalently **"SOP"** meaning sum of products) and **"CNF"** for conjunctive normal form (or equivalently **"POS"** meaning product of sums). Be sure to include the quotation marks.

Below, we convert **example3** to disjunctive normal form using **"DNF″** in the second argument.

```
In[43]:= BooleanConvert[example3, "DNF"]

Out[43]= (x && ! z) || (y && ! z)
```

Note that this expression is different from the solution to Example 3 in the text. *Mathematica* produces a reduced disjunctive normal form in which terms are not required to contain every variable.

`BooleanConvert` can also be applied with a single argument, in which case it defaults to disjunctive normal form.

```
In[44]:= BooleanConvert[example3]

Out[44]= (x && ! z) || (y && ! z)
```

To produce conjunctive normal form, **"CNF"** (or **"POS"**) is required as the second argument.

```
In[45]:= BooleanConvert[example3, "CNF"]

Out[45]= (x || y) && ! z
```

Disjunctive Normal Form from a Table

Example 1 of Section 12.2 describes how to find an expression for a Boolean function represented by a table of values. We will illustrate the built-in *Mathematica* functions that accomplish this task.

To define a Boolean function using truth values, we will use `BooleanFunction`. This function has a variety of uses, including the ability to specify a Boolean function in terms of the number variables and an index into the list of all Boolean functions on that number of variables, and the ability to specify a Boolean function in terms of truth value assignments. We will focus on the latter.

For example, consider the function defined by the following table.

x	y	z	F (x, y, z)
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	1
0	1	1	0
0	1	0	1
0	0	1	1
0	0	0	0

There are two ways we can represent this function in *Mathematica* that will be usable as input to `BooleanFunction`. The first is as rules that assign the truth value assignments to the value of the function. For example, the third row in the table would correspond to the rule {1, 0, 1} → 0. Note that `BooleanFunction` allows for the use of 0-1 notation for true and false. It will also accept the symbols `True` and `False`, as in {True, False, True} → False. We will use the 0-1 notation as it makes for shorter input sequences.

Given a list of such rules, the `BooleanFunction` will output a *Mathematica* function object.

```
In[46]:= BFexample = BooleanFunction[
           {{1, 1, 1} → 0, {1, 1, 0} → 1, {1, 0, 1} → 0, {1, 0, 0} → 1,
            {0, 1, 1} → 0, {0, 1, 0} → 1, {0, 0, 1} → 1, {0, 0, 0} → 0}]

Out[46]= BooleanFunction[ < 3 >]
```

Applying the function to truth values yields the appropriate result.

```
In[47]:= BFexample[True, False, True]

Out[47]= False
```

With a list of variable names as a second argument, `BooleanFunction` will output an expression for the Boolean function instead of a `BooleanFunction` object.

```
In[48]:= BooleanFunction[{{1, 1, 1} → 0, {1, 1, 0} → 1,
           {1, 0, 1} → 0, {1, 0, 0} → 1, {0, 1, 1} → 0,
           {0, 1, 0} → 1, {0, 0, 1} → 1, {0, 0, 0} → 0}, {x, y, z}]

Out[48]= (x && ! z) || (! x && ! y && z) || (y && ! z)
```

A third argument can be used to specify the form, e.g., **"DNF"** or **"CNF"**, for the output.

```
In[49]:= BooleanFunction[{{1, 1, 1} → 0, {1, 1, 0} → 1,
           {1, 0, 1} → 0, {1, 0, 0} → 1, {0, 1, 1} → 0, {0, 1, 0} → 1,
           {0, 0, 1} → 1, {0, 0, 0} → 0}, {x, y, z}, "CNF"]

Out[49]= (! x || ! z) && (x || y || z) && (! y || ! z)
```

You can simplify the input to `BooleanFunction` slightly by entering only those rules corresponding to one of the possible function values, say true, and then use a `BlankSequence` (`__`) to assert that all others have the other value. This is illustrated below for the same function,

```
In[50]:= BooleanFunction[{{1, 1, 0} → 1, {1, 0, 0} → 1,
           {0, 1, 0} → 1, {0, 0, 1} → 1, {__} → 0}, {x, y, z}, "CNF"]

Out[50]= (! x || ! z) && (x || y || z) && (! y || ! z)
```

Observe that this is identical to the output above.

You can simplify the input even further by entering only the output values of the function, provided you enter them in standard order. The order the values must be entered in is the same as displayed in the table above.

```
In[51]:= BooleanFunction[{0, 1, 0, 1, 0, 1, 1, 0}, {x, y, z}]

Out[51]= (x && ! z) || (! x && ! y && z) || (y && ! z)
```

If you are in doubt of the order in which to enter the function's values, it is identical to the order used by `BooleanTable`, the function used to display truth tables. Below we use `BooleanTable` to display the canonical ordering of truth value assignments for two variables.

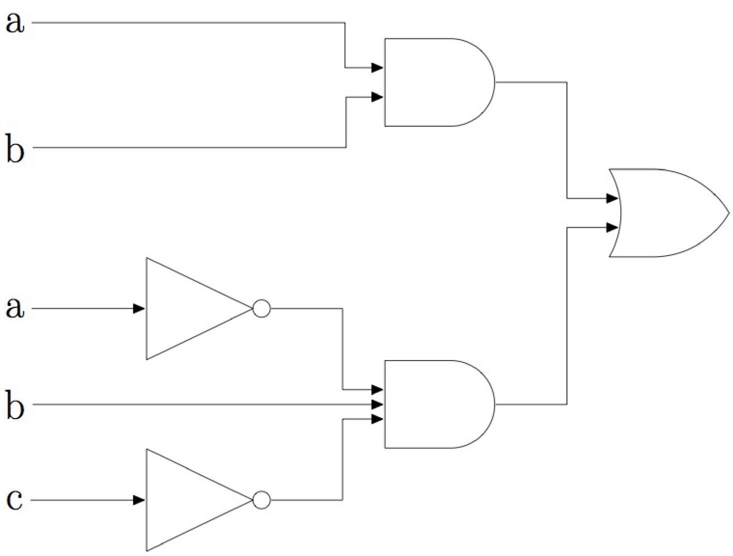
```
In[52]:= BooleanTable[{x, y}, {x, y}] // TableForm

Out[52]//TableForm=
         True     True
         True     False
         False    True
         False    False
```

12.3 Logic Gates

In this section, we will use *Mathematica* to work with logic gates, particularly circuit diagrams. First, we will see how to use *Mathematica* to translate a circuit diagram into a Boolean expression. Then we will do the reverse and see how to transform a logical expression into a circuit diagram (modeled as a tree diagram).

Circuit Diagram to Logical Expression

Consider the circuit diagram shown below.

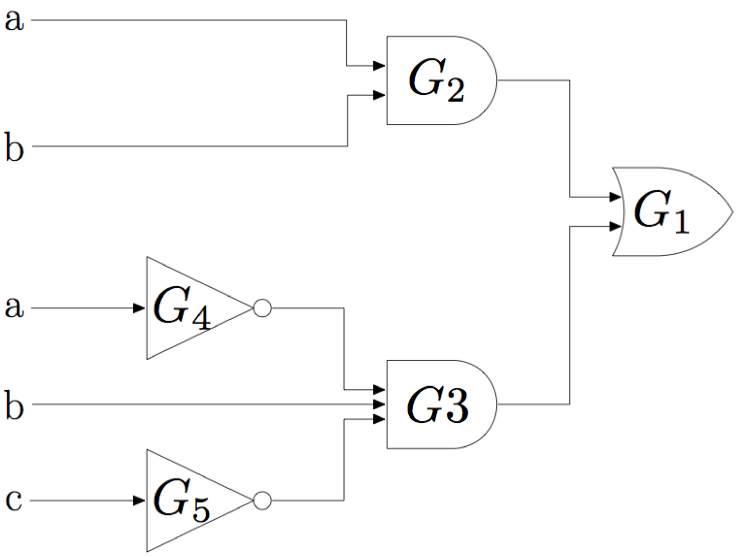


Our goal in this subsection is to use *Mathematica* to produce a logical expression for the output of this diagram.

To do this, we use the fact that *Mathematica* has functional forms for each of the logical expressions. That is, `Not`, `And`, and `Or` can all be used as functions applied to expressions. For example, the following forms the disjunction of **`x`**, **`y`**, and **`z`**.

```
In[53]:=  Or[x, y, z]

Out[53]=  x || y || z
```

We give each gate in the diagram a label. The specific names are not important. We chose to label the gates using the capital letter G with subscripts numbered from the right to the left.



Interpret the labels as names for both the gates themselves and for their outputs. Note G_1 is the name for both the output of the final gate and also names the output of the circuit. The input of G_1 is the outputs from gates G_2 and G_3. That is to say, $G_1 = G_2$ or G_3.We can write that in *Mathematica* as shown below.

```
In[54]:=  G1 = Or[G2, G3]

Out[54]=  G2 || G3
```

For each gate, do the same. Note that the order in which the gates are specified is irrelevant. The gate G_2 is a and b.

```
In[55]:=  G2 = And[a, b]

Out[55]=  a && b
```

The output of G_3 is the conjunction of G_4, b, and G_5.

```
In[56]:=  G3 = And[G4, b, G5]

Out[56]=  G4 && b && G5
```

And G_4 and G_5 are inversions on a and c, respectively.

```
In[57]:=  G4 = Not[a]

Out[57]=  ! a

In[58]:=  G5 = Not[c]

Out[58]=  ! c
```

Once all of the gates have been specified, inspect the value for the final gate, G_1.

```
In[59]:=  G1

Out[59]=  (a && b) || (! a && b && ! c)
```

This tells us that the circuit's result is $(a \bigwedge b) \bigvee (\neg\, a \bigwedge b \bigwedge \neg\, c)$. In 0-1 form, this is $\mathrm{ab} + \overline{a}\, b\, \overline{c}$.

The reason this works is that when we define the output of a gate in terms of unassigned names, such as G_2, *Mathematica* accepts the definition. When G_2 is later assigned its own value and then the expression for G_1 is evaluated, *Mathematica* resolves all assigned names into their definitions so that the expression for G_1 is in terms of unassigned names (a, b, and c) only.

Logical Expression to Circuit Diagram

We have just seen how to use *Mathematica* to transform a circuit diagram into a logical expression for the result of the circuit. Now we consider the reverse. Given a logical expression, such as that for G_1, we will use *Mathematica* to transform the expression into a circuit diagram.

We will model a circuit diagram as a binary tree. While circuit diagrams are generally not necessarily binary, this will serve for our purposes.

Recall that a binary tree has a number of vertices and directed edges. Vertices in the tree will correspond to gates in the circuit. One of the vertices is distinguished as the root, which will correspond to the output of the circuit. Each vertex has at most two children vertices. The edges between the vertex and its child correspond to the inputs to the gate. Each vertex other than the root has a parent, and the edge from the vertex to the parent corresponds to the output from the gate.

The assumption that a circuit can be modeled as a binary tree requires that the circuit satisfy the following properties. First, the circuit has only one output. Second, each gate has only one output. Third, each gate has at most two inputs.

Recall that in Chapter 11, we wrote the function **`expressionTree`** for converting an algebraic expression in terms of the binary arithmetic operators into a tree representation. We will make use of the functions from Chapter 11 here, so we load them. If you place the file Chapter11.mx from the website in the same directory as this notebook is stored, the executing the following expression will load the functions from Chapter 11.

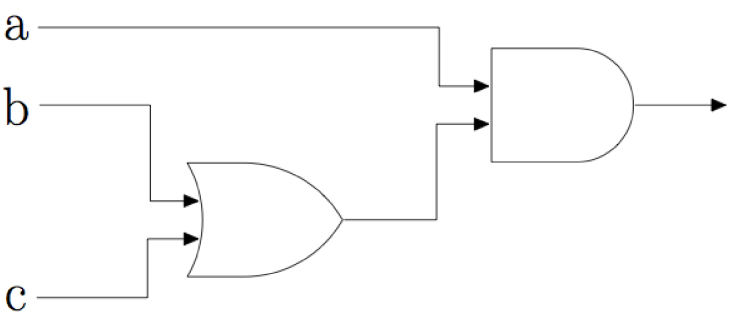
```
In[60]:= << (NotebookDirectory[] <> "Chapter11.mx")

         Get::noopen : Cannot open
              /Users/Dan/Dropbox/RosenMathematica/Chapter12/Chapter11.mx. ≫

Out[60]= $Failed
```

Recall that the functions written in each chapter of this manual are collected in packages. Please refer to the Introduction if you need instructions on how to use the packages on your system.

Observe what happens if we apply the **`expressionTree`** function to the logical expression $a \wedge (b \vee c)$.

```
In[61]:= expressionTree[a && (b || c), ImagePadding → 5]

Out[61]= expressionTree[a && (b || c), ImagePadding → 5]
```

Compare the tree above to the circuit diagram below.



Observe that the diagram and the tree have the same structure. After reversing the arrows, rotating by 90°, and exchanging the symbols with the functions labeling the internal nodes, the two are identical.

As you can see, we nearly have a function for turning logical expressions into trees that correspond to circuit diagrams. The only problem is that the **`expressionTree`** function does not allow for unary operators such as `Not`.

In order to create a function like this that will work with expressions using the Boolean operators, we need to allow for the possibility that there is only one operand.

The inner `If` statement below tests the number of operands of the expression using `Length`. This will function similarly to the original function, except using the left hand side for the sole operand. We also need to create a new function, **`addLeftBranch`**, to take the place of **`joinTrees`** for unary operators.

```
In[62]:= SetAttributes[logicTree, HoldFirst];
         logicTree[expr_, opts___] :=
          Module[{e, lhs, rhs, operator, lhsTree, rhsTree, result},
           If[expr[[0]] === Hold,
            e = expr,
            e = Hold[expr]
           ];
           If[e[[1, 0]] === Integer || e[[1, 0]] === Symbol,
            result = newExpressionTree[ReleaseHold[e], opts],
            operator = Extract[e, {1, 0}];
            If[Length[e[[1]]] == 1,
             lhs = Extract[e, {1, 1}, Hold];
             lhsTree = logicTree[lhs];
             result = addLeftBranch[operator, lhsTree, opts],
             (* else there are 2 operands *)
             lhs = Extract[e, {1, 1}, Hold];
             rhs = Extract[e, {1, 2}, Hold];
             lhsTree = logicTree[lhs];
             rhsTree = logicTree[rhs];
             result = joinTrees[operator, lhsTree, rhsTree, opts]
            ]
           ];
           drawBinaryTree[result]
          ]
```

To implement **`addLeftBranch`**, we essentially repeat the definition of **`joinTrees`** with the elements related to the right child removed.

```
In[64]:= addLeftBranch[newR_, A_?binaryTreeQ, opts___] :=
          Module[{newRI, newT, newVerts, Aroot, newEdges, v, e, p, w},
           newRI = ToString[newRI];
           newVerts = Join[{newRI}, VertexList[A]];
           Aroot = findRoot[A];
           newEdges = Join[EdgeList[A], {DirectedEdge[newRI, Aroot]}];
           newT = Graph[newVerts, newEdges, opts];
           Do[PropertyValue[{newT, v}, "order"] =
             PropertyValue[{A, v}, "order"]
            , {v, VertexList[A]}];
           PropertyValue[{newT, Aroot}, "order"] = 1;
           PropertyValue[{newT, newRI}, "order"] = 0;
           Do[PropertyValue[{newT, v}, VertexLabels] =
             PropertyValue[{A, v}, VertexLabels]
            , {v, VertexList[A]}];
           PropertyValue[{newT, newRI}, VertexLabels] = newR;
           drawBinaryTree[newT]
          ]
```

With these in place, let's look at the result for an example. Consider $(x+y)\overline{x}$, the subject of Example 1(a) from the text.

```
In[65]:= logicTree[(x || y) && ! x, ImagePadding → 10]

Out[65]= drawBinaryTree[joinTrees[And, drawBinaryTree[
             joinTrees[Or, drawBinaryTree[newExpressionTree[x]],
              drawBinaryTree[newExpressionTree[y]]]]], drawBinaryTree[
             addLeftBranch[Not, drawBinaryTree[newExpressionTree[x]]]]],
           ImagePadding → 10]]
```

Compare this diagram to Figure 4(a) in the text.

Note that the function above will not work correctly on expressions, such as the expression **G1**, that *Mathematica* interprets as applications of `And` or `Or` with more than two arguments. Observe that the `FullForm` of **G1** reveals that *Mathematica* considers it to include an `And` applied to three arguments.

```
In[66]:= FullForm[G1]

Out[66]//FullForm=
         Or[And[a, b], And[Not[a], b, Not[c]]]
```

It is left as an exercise for the reader to alter the **logicTree** function to handle this case.

12.4 Minimization of Circuits

In this section we will discuss the use of the `BooleanMinimize` function for minimizing circuits, and we will provide an implementation of the Quine-McCluskey method.

The `BooleanMinimize` Function

The `BooleanMinimize` function, applied to a Boolean expression, finds a minimal representation of the expression in disjunctive normal form.

For example, we apply `BooleanMinimize` to the expression we obtained for the output of the circuit diagram at the beginning of Section 12.3.

```
In[67]:= BooleanMinimize[(a && b) || (! a && b && ! c)]

Out[67]= (a && b) || (b && ! c)
```

The result indicates that $\neg a$ can be removed as an input to the second AND gate.

The result of `BooleanMinimize` is guaranteed to be of minimal length among all possible disjunctive normal form representations of the input, however, such a minimal expression is not unique. Note that if you don’t care that the expression be in disjunctive normal form, the `Simplify` function will produce a shorter expression.

```
In[68]:= Simplify[(a && b) || (! a && b && ! c)]

Out[68]= (a || ! c) && b
```

The `BooleanMinimize` function can also accept, as a second argument, all the same forms as `BooleanConvert` in order to produce minimal expressions of different forms. For example, to find a minimal conjunctive normal form expression for G_1, you enter the following.

```
In[69]:= BooleanMinimize[(a && b) || (! a && b && ! c), "CNF"]

Out[69]= (a || ! c) && b
```

Don’t Care conditions

Informally, a set of don't care conditions for a Boolean function F is a set of points in the domain of F whose images do not concern us.

If F is a function on n variables, then its domain is $\{\text{true, false}\}^n$. Let A be the subset of $\{\text{true, false}\}^n$ for which the value of F is specified. If we think of F as fully defined on this subset A, then we are interested in the family of all extensions of F to all of $\{\text{true, false}\}^n$. In other words, the set of all G defined on $\{\text{true, false}\}^n$ that agree with F on A. The goal is to choose the particular G that is simplest. That is, the G that has the smallest sum of products expansion.

We should pause to consider the size of this problem. If there are d don't care points, then there are 2^d possible extensions G. Considering every possible extension can become rather time consuming.

Consider the Boolean function F defined by the following table of values, in which “d” in the final column indicates a don't care condition.

x	y	z	F(x,y,z)
true	true	true	true
true	true	false	false
true	false	true	false
true	false	false	true
false	true	true	d
false	true	false	d
false	false	true	false
false	false	false	true

The points that must evaluate to true are: {(false, false, false), (true, false, false), (true, true, true)} and the don't care conditions are {(false, true, false), (false, true, true)}.

In Section 12.2, we showed how to use `BooleanFunction` to define a Boolean function in terms of a table. We also saw above that you can specify a default output by using a `BlankSequence` (__). For example, the following returns the Boolean expression that is true on (true, false, true, false) and false otherwise.

```
In[70]:= BooleanFunction[
           {{True, False, True, False} → True, {__} → False}, {x, y, z, w}]

Out[70]= ! w && x && ! y && z
```

We can specify a don't care condition within a call to `BooleanFunction` by identifying a don't care condition, e.g., (false, true, false), with a blank.

So we can determine a Boolean function defined by the table above as follows. (Note that we could use a `BlankSequence` (__) to simplify the input, but in this example we list all the elements of the domain for clarity.)

```
In[71]:= BooleanFunction[{{True, True, True} → True,
            {True, True, False} → False,
            {True, False, True} → False,
            {True, False, False} → True,
            {False, True, True} → _,
            {False, True, False} → _,
            {False, False, True} → False,
            {False, False, False} → True},
           {x, y, z}]

Out[71]= (x && y && z) || (! x && ! z) || (! y && ! z)
```

Quine-McCluskey

We conclude with an implementation of the Quine-McCluskey method. This method is fairly involved and it will take considerable effort to implement it correctly, but understanding this algorithm is worthwhile.

It will be helpful to have an example that we can use to illustrate the method as we build the function. The expression we use for the example is

$$w\,x\,\overline{y}\,\overline{z}+w\,\overline{x}\,y\,z+w\,\overline{x}\,y\,\overline{z}+w\,\overline{x}\,\overline{y}\,\overline{z}+\overline{w}\,x\,y\,z+\overline{w}\,x\,\overline{y}\,z+\overline{w}\,x\,\overline{y}\,\overline{z}+\overline{w}\,\overline{x}\,y\,z+\overline{w}\,\overline{x}\,\overline{y}\,z+\overline{w}\,\overline{x}\,\overline{y}\,\overline{z}$$

We assign this to the symbol **`F`**.

```
In[72]:= F = (w && x && ! y && ! z) || (w && ! x && y && z) ||
           (w && ! x && y && ! z) || (w && ! x && ! y && ! z) ||
           (! w && x && y && z) || (! w && x && ! y && z) ||
           (! w && x && ! y && ! z) || (! w && ! x && y && z) ||
           (! w && ! x && ! y && z) || (! w && ! x && ! y && ! z);
```

Let us begin by (very) briefly outlining the approach. More details will be given as we proceed.

1. Transform the minterms into bit strings.
2. Group the bit strings by the number of 1s.
3. Combine bit strings that differ in exactly one location.
4. Repeat steps 2 and 3 until no additional combinations are possible.
5. Identify the prime implicants (those bit strings not used in a simplification) and form the coverage table.
6. Identify the essential prime implicants and update the table.
7. Process the remaining prime implicants using a heuristic approach in order to achieve complete coverage.

Implementing this will require several different functions that will come together to achieve the goal of minimizing the expression for F.

Modifying Arguments

Before we begin implementing the method, we take a moment to explain the `HoldRest` attribute. Earlier in this manual, we have seen how to use a held argument to allow modification of an argument to a function. We will need to do this again here in order to avoid the need to copy data structures that must be modified by a function.

Holding parameters to a function means that, when you call the function on a symbol, instead of applying the function to the object stored in that symbol, the function is given the name of the symbol itself. This allows the symbol name to be reassigned and otherwise modified within the function.

The `HoldRest` attribute causes all but the first argument to a function to be held. For example, the following function updates the symbol given as the second argument to be the sum of what it previously stored and the first argument, and appends the result to the list associated to the symbol given as the third argument.

```
In[73]:= exampleHold1 = 5;
         exampleHold2 = 12;
         exampleHold3 = {1, 2, 3};
         SetAttributes[exampleHoldFunction, {HoldRest}];
         exampleHoldFunction[a_, b_, c_] := Module[{},
           b = a + b;
           AppendTo[c, b]
          ]
```

```
In[78]:= exampleHoldFunction[exampleHold1, exampleHold2, exampleHold3]

Out[78]= {1, 2, 3, 17}
```

Observe that the values stored in the last two arguments have changed. Without the `HoldRest` attribute, both expressions in the body of the `Module` would have produced errors.

```
In[79]:= exampleHold2

Out[79]= 17

In[80]:= exampleHold3

Out[80]= {1, 2, 3, 17}
```

Other attributes that can be used to hold arguments are `HoldFirst` and `HoldAll`.

Transforming Minterms Into Bit Strings

The first task is to process the input. That is, *F* must be transformed into a list of bit strings. This is not strictly necessary, but it makes working with the minterms more convenient. We represent bit strings as lists of 0s and 1s.

We begin by creating a function to transform a single minterm into a bit string. We assume that the input to this function will be a properly formed minterm, that is, a conjunction of variables and negations of variables. We require that a list of variables be provided to the function, so that the bit string can be formed in the proper order.

Consider the following minterm, which is the fourth minterm in our example *F*.

```
In[81]:= minterm = w && ! x && y && ! z

Out[81]= w && ! x && y && ! z
```

Fortunately, *Mathematica* automatically transforms a series of conjunctions into a single application of the `And` (**`&&`**) function with multiple arguments. Applying `FullForm` illustrates.

```
In[82]:= FullForm[minterm]

Out[82]//FullForm=
         And[w, Not[x], y, Not[z]]
```

`MemberQ`’s first argument can have any head, not just a `List`. So, we can determine that *w* is part of the conjunction but that *x* is not by applying `MemberQ` to **`minterm`** and the variables.

```
In[83]:= MemberQ[minterm, w]

Out[83]= True

In[84]:= MemberQ[minterm, x]

Out[84]= False
```

But of course, the negation of *x* is part of **`minterm`**.

```
In[85]:= MemberQ[minterm, Not[x]]

Out[85]= True
```

To transform the minterm into a bit string, we only need to check, for each variable, whether the variable or its negation is in the list. Recall that we will insist that the function be given the list of variables as an argument to maintain the proper order of the variables.

We first assign the list of variables to a symbol.

```
In[86]:= variableList = {w, x, y, z}

Out[86]= {w, x, y, z}
```

Now create a list, initialized to the proper length, for the bit string.

```
In[87]:= bitstring = ConstantArray[Null, 4]

Out[87]= {Null, Null, Null, Null}
```

Finally, we use a `For` loop to check, for each variable in the variable list, whether the variable is in the minterm. If the variable is a member of **minterm**, then we change the bit to 1. If the negation is in **minterm**, we set the value in the bit string to 0. Otherwise, we place the character “-“ in the list, to indicate the absence of the variable in the string.

```
In[88]:= For[i = 1, i ≤ Length[variableList], i++,
          Which[MemberQ[minterm, variableList[[i]]],
           bitstring[[i]] = 1,
           MemberQ[minterm, Not[variableList[[i]]]],
           bitstring[[i]] = 0,
           True, bitstring[[i]] = "-"]
         ]
```

This has created the bit string associated to **minterm**.

```
In[89]:= bitstring

Out[89]= {1, 0, 1, 0}
```

We condense this process into a single function.

```
In[90]:= mtToBitstring[minterm_, variableList_] :=
          Module[{i, bitstring},
           bitstring = ConstantArray[Null, Length[variableList]];
           For[i = 1, i ≤ Length[variableList], i++,
            Which[MemberQ[minterm, variableList[[i]]],
             bitstring[[i]] = 1,
             MemberQ[minterm, Not[variableList[[i]]]],
             bitstring[[i]] = 0,
             True,
             bitstring[[i]] = "-"
            ]
           ];
           bitstring
          ]
```

```
In[91]:= mtToBitstring[minterm, {w, x, y, z}]

Out[91]= {1, 0, 1, 0}
```

Transforming the Original Expression Into Bit Strings

Now that we have the means for transforming a single minterm into a bit string, we are ready to transform an expression in disjunctive normal form into a list of bit strings.

Observe that an expression in disjunctive normal form is the Or (||) function applied to minterms. Again, FullForm reveals this convenient structure.

```
In[92]:= FullForm[F]
Out[92]//FullForm=
         Or[And[w, x, Not[y], Not[z]],
           And[w, Not[x], y, z], And[w, Not[x], y, Not[z]],
           And[w, Not[x], Not[y], Not[z]], And[Not[w], x, y, z],
           And[Not[w], x, Not[y], z], And[Not[w], x, Not[y], Not[z]],
           And[Not[w], Not[x], y, z], And[Not[w], Not[x], Not[y], z],
           And[Not[w], Not[x], Not[y], Not[z]]]
```

Our goal is to produce a list of the bit strings obtained from the minterms. We can transform the disjunction into a list by using the Apply (@@) function to replace the Or (||) head with a List head.

```
In[93]:= List @@ F
Out[93]= {w && x && ! y && ! z, w && ! x && y && z,
          w && ! x && y && ! z, w && ! x && ! y && ! z, ! w && x && y && z,
          ! w && x && ! y && z, ! w && x && ! y && ! z, ! w && ! x && y && z,
          ! w && ! x && ! y && z, ! w && ! x && ! y && ! z}
```

Then we just need to apply the **mtToBitstring** function to each member of the list. We can do this by using the Map (/@) function, which applies a function (given as the first argument) to a list (given as the second argument) and returns the list obtained by applying the function to each element of the list. Since the **mtToBitstring** requires two arguments, not just one, the first argument to Map (/@) will be a pure Function (&) obtained by calling **mtToBitstring** on a Slot (#) and the list of variables.

```
In[94]:= Map[mtToBitstring[#, {w, x, y, z}] &, List @@ F]
Out[94]= {{1, 1, 0, 0}, {1, 0, 1, 1}, {1, 0, 1, 0}, {1, 0, 0, 0}, {0, 1, 1, 1},
          {0, 1, 0, 1}, {0, 1, 0, 0}, {0, 0, 1, 1}, {0, 0, 0, 1}, {0, 0, 0, 0}}
```

We define a function based on this model.

```
In[95]:= dnfToBitList[dnfExpr_, variableList_] :=
          Map[mtToBitstring[#, variableList] &, List @@ dnfExpr]
```

We now apply this function to the example expression and store the result as the symbol **Fbits**.

```
In[96]:= Fbits = dnfToBitList[F, {w, x, y, z}]
Out[96]= {{1, 1, 0, 0}, {1, 0, 1, 1}, {1, 0, 1, 0}, {1, 0, 0, 0}, {0, 1, 1, 1},
          {0, 1, 0, 1}, {0, 1, 0, 0}, {0, 0, 1, 1}, {0, 0, 0, 1}, {0, 0, 0, 0}}
```

Transforming Bit Strings Into Minterms

At the conclusion of the Quine-McCluskey process, we will want to display the result in disjunctive normal form. This will require that we turn bit strings back into minterms.

Note that since this function will be applied at the end of the process, it may be that some of the variables have been removed. We will be using the string “-“ in a bit string to indicate the elimination of a variable.

This function will require the bit string and a list of variable names as its input. It operates in two stages. First, it processes the variable list based on the content of the bit string. It initializes an empty list and, for each variable, appends the variable or its negation or does nothing, depending on the content of the bit string.

```
In[97]:= bitstr = {0, 1, "-", 0}

Out[97]= {0, 1, -, 0}

In[98]:= outList = {}

Out[98]= {}

In[99]:= For[i = 1, i ≤ Length[variableList], i++,
          Switch[bitstr[[i]],
           1, AppendTo[outList, variableList[[i]]],
           0, AppendTo[outList, Not[variableList[[i]]]],
           "-", Null]
         ]

In[100]:= outList

Out[100]= {! w, x, ! z}
```

Note the use of the `Switch` function. Recall that `Switch` evaluates its first argument, and then compares that result to the arguments with even index, executing the argument following the first of the even-indexed arguments that matches the result of the first argument.

Once this list is formed, we form the conjunction of the elements by using `Apply` (**`@@`**) to change the `List` head into `And` (**`&&`**).

```
In[101]:= And @@ outList

Out[101]= ! w && x && ! z
```

Here is the function based on this process.

```
In[102]:= bitStringToMT[bitstring_, variableList_] :=
           Module[{outList, i},
            outList = {};
            For[i = 1, i ≤ Length[variableList], i++,
             Switch[bitstring[[i]],
              1, AppendTo[outList, variableList[[i]]],
              0, AppendTo[outList, Not[variableList[[i]]]],
              "-", Null]
            ];
            And @@ outList
           ]
```

Applied to {0, 1, 0, 1} and {w, x, y, z}, we see that **`bitStringToMT`** reproduces the original minterm example.

```
In[103]:= bitStringToMT[{0, 1, 0, 1}, {w, x, y, z}]
```

```
Out[103]= ! w && x && ! y && z
```

And applied to {0, 1, "−", 1}, it removes the *y* and negates *w*.

```
In[104]:= bitStringToMT[{0, 1, "-", 1}, {w, x, y, z}]
```

```
Out[104]= ! w && x && z
```

The final result of our Quine-McCluskey process will be a list of bit strings. To produce the associated disjunctive normal form expression, we only need to apply **bitStringToMT** to each element of the list and then join the elements of the list in a disjunction.

```
In[105]:= bitListToDNF[bitList_, variableList_] :=
           Map[bitStringToMT[#, variableList] &, Or @@ bitList]
```

Initializing the Source Table

In order to form the coverage table in the second part of the method, we need to know which of the original minterms are covered by which of the prime implicants. Refer to Tables 3 and 6 in the text. Notice that each bit string in those tables is associated with either a single number, in the case of the original minterms, or lists of numbers, for the derived products.

We will store this information as an indexed symbol whose indices are the bit strings and whose values are lists of integers. Given the **Fbits** list, we initialize this indexed symbol with the elements of **Fbits** as the indices. The corresponding entries will be the whose sole element is the bit string's position in **Fbits**.

We will refer to this as the "coverage dictionary," since it allows us to look up any bit string and determine all of the original minterms covered by it. The following function accepts the name of a symbol and the **Fbits** list as arguments and assigns the coverage dictionary to the symbol. Note that we must set the `HoldFirst` attribute in order to pass the symbol name as an argument. We apply `Clear` so as to remove anything already associated to the symbol.

```
In[106]:= SetAttributes[initCoverDict, {HoldFirst}];
          initCoverDict[symbol_, L_] := Module[{i},
            Clear[symbol];
            For[i = 1, i ≤ Length[L], i++,
             symbol[L[[i]]] = {i}
            ]
           ]
```

Applying this function to a symbol and **Fbits** produces the initial coverage dictionary.

```
In[108]:= initCoverDict[coverageDict, Fbits]
```

We can check what is stored in the indexed symbol **coverageDict** as shown below. Remember that the indices are the elements of the list of bit strings **Fbits**.

```
In[109]:= coverageDict[{0, 1, 0, 1}]
```

```
Out[109]= {6}
```

The value, after initialization, is the location within **Fbits** where the bit string is stored.

```
In[110]:= Fbits[[6]]

Out[110]= {0, 1, 0, 1}
```

Grouping by the Number of 1s

Step 2 in our outline is to group the bit strings by the number of 1s.

The reason for this step is to improve the efficiency of finding simplifications to make. Since two bit strings can be combined only when they are identical except for one location, the only possible combinations are when one bit string has n 1s and the other has $n-1$.

After step 1 is concluded, we have a list of bit strings. That will be the starting point for the function we create for this step. The result of this step will be to turn the list of bit strings into a list of lists of bit strings, which we'll call **groups**. In location **i** of **groups** will be the set of all bit strings with $i-1$ 1s. So location 1 will have the bit strings with no 1s, location 2 will contain the bit strings with a single 1, etc.

We know that the number of 1s in any bit string must be between 0 and the length of the bit string. We initialize **groups** to be the list of empty lists. The maximum number of 1s in equal to the length of a bit string, which we can obtain from the size of the first element of **Fbits**.

```
In[111]:= groups = ConstantArray[{}, Length[First[Fbits]] + 1]

Out[111]= {{}, {}, {}, {}, {}}
```

Since the bit strings had four entries, **groups** now consists of five copies of the empty list.

For each member of **Fbits**, we need to count the number of 1s. We will create a small function to do this.

```
In[112]:= count1s[bitstring_] := Module[{c = 0, i},
            Do[If[i == 1, c++]
              , {i, bitstring}];
            c
           ]
```

We test this function on a small example.

```
In[113]:= count1s[{1, 0, 1, 1, 0, 0, 1}]

Out[113]= 4
```

We use **count1s** to sort the members of **Fbits** into **groups**. Using a loop to step through the **Fbits** list, we apply **count1s** and add 1 to the result (since the bit strings with no 1s are in the first position) to obtain the correct location for that bit string in **groups**. We then add that bit string to the correct sublist in **groups**.

Here is the function implementing this.

```
In[114]:= sortGroups[bitstringList_] := Module[{groups, bitstring, c},
            groups = ConstantArray[{}, Length[First[bitstringList]] + 1];
            Do[c = count1s[bitstring];
             groups[[c + 1]] = Append[groups[[c + 1]], bitstring]
             , {bitstring, bitstringList}];
            groups
           ]
```

Here is the result of sorting **Fbits**.

```
In[115]:= groups = sortGroups[Fbits]

Out[115]= {{{0, 0, 0, 0}}, {{1, 0, 0, 0}, {0, 1, 0, 0}, {0, 0, 0, 1}},
           {{1, 1, 0, 0}, {1, 0, 1, 0}, {0, 1, 0, 1}, {0, 0, 1, 1}},
           {{1, 0, 1, 1}, {0, 1, 1, 1}}, {}}
```

Applying `TableForm` will make the output easier to read. The `TableDepth` option prevents `TableForm` from splitting the sublists into subtables.

```
In[116]:= TableForm[groups, TableDepth → 1]

Out[116]//TableForm=
          {{0, 0, 0, 0}}
          {{1, 0, 0, 0}, {0, 1, 0, 0}, {0, 0, 0, 1}}
          {{1, 1, 0, 0}, {1, 0, 1, 0}, {0, 1, 0, 1}, {0, 0, 1, 1}}
          {{1, 0, 1, 1}, {0, 1, 1, 1}}
          {}
```

Combining Bit Strings

Step 3 is to combine all of the bit strings that differ in exactly one location. We first write a function that takes as input two bit strings and either combines them if, in fact, they do differ in exactly one location, or returns `False` if they do not.

This function needs to do two tasks. First, it has to check to see whether or not the two bit strings differ in more than one location. Second, it needs to combine them if they are allowed to be combined.

Combining two bit strings is easy, provided we know the one location in which they differ. For example,

```
In[117]:= bit1 = {1, "-", 0, 1, 1}

Out[117]= {1, -, 0, 1, 1}

In[118]:= bit2 = {1, "-", 0, 0, 1}

Out[118]= {1, -, 0, 0, 1}
```

You can see that these are identical except in position 4.

To merge them, we take either one and replace position 4 with “-“.

```
In[119]:= bit1[[4]] = "-";
          bit1

Out[120]= {1, -, 0, -, 1}
```

We determine that they differ only in position 4 using a `Catch` and `Throw` with a `For` loop. Inside a `Catch` block, initialize a symbol **pos**, for position, to 0. Now begin a `For` loop to compare each pair of entries in the two bit strings. If we find a difference, check the value of **pos**. If it is still 0, then this is the first difference that has been encountered, so set **pos** to the position of this difference and continue the loop. If **pos** is not 0, however, then we know that this is the second time a difference was found. In this case, we immediately `Throw` `False`, terminating the loop. Once the loop is complete, if **pos** is still 0, then there was no difference, so again we `Throw` `False`. Otherwise, **pos** stores the location of the sole difference, and we modify one of the bit strings and return it.

Here is the function

```
In[121]:= mergeBitstrings[bit1_, bit2_] := Module[{i, pos, result},
            Catch[
             pos = 0;
             For[i = 1, i ≤ Length[bit1], i++,
              If[bit1[[i]] ≠ bit2[[i]],
               If[pos == 0, pos = i, Throw[False]]
              ]
             ];
             If[pos == 0, Throw[False]];
             result = bit1;
             result[[pos]] = "-";
             Throw[result]
            ]
           ]
```

We see that it works correctly on our two example bit strings.

```
In[122]:= mergeBitstrings[{1, "-", 0, 1, 1}, {1, "-", 0, 0, 1}]

Out[122]= {1, -, 0, -, 1}
```

Searching for Combinations to Make

The **mergeBitstrings** function will do the work of checking to see if bit strings can be merged and returning the result if they can. However, we need to give **mergeBitstrings** the bit strings to test.

Recall that, in our example, we have successfully grouped the minterms by the number of 1s they contain.

```
In[123]:= TableForm[groups, TableDepth → 1]

Out[123]//TableForm=
          {{0, 0, 0, 0}}
          {{1, 0, 0, 0}, {0, 1, 0, 0}, {0, 0, 0, 1}}
          {{1, 1, 0, 0}, {1, 0, 1, 0}, {0, 1, 0, 1}, {0, 0, 1, 1}}
          {{1, 0, 1, 1}, {0, 1, 1, 1}}
          {}
```

Next we will produce a list containing all the bit strings formed by merging two members of **groups**.

Note there may be multiple ways to obtain the same bit string, so we will think of this collection as a set. We initialize to the empty set.

```
In[124]:= Fbits1 = {}

Out[124]= {}
```

Also recall that it is only possible to merge bit strings that are in successive locations in **groups**. In other words, we only need to check bit strings when one has *n* 1s and one has $n-1$ 1s. This suggests a loop with *n* ranging from 1 to one less than the size of **groups**. Within the body of the loop, we will consider the lists with $n-1$ 1s (index *n*) and with *n* 1s (index $n+1$). (Remember that **groups[[1]]** is the set of bit strings with 0 1s.)

The loop is structured as follows.

```
For[n = 1, n ≤ Length[groups] - 1, n++,
 A = groups[[n]];
 B = groups[[n + 1]];
  (* search for bit strings from A and B to merge *)
]
```

After **A** and **B** have been defined, we need to compare every possible pair. We use a Do loop with indices for the members of **A** and another for members of **B**. Within the Do loop, we use **mergeBitstrings** and store the result. If it is not false, we add it to the new list of bit strings, **Fbits1**.

```
In[125]:= For[n = 1, n ≤ Length[groups] - 1, n++,
            A = groups[[n]];
            B = groups[[n + 1]];
            Do[m = mergeBitstrings[a, b];
             If[m =!= False, Fbits1 = Union[Fbits1, {m}]]
             , {a, A}, {b, B}
            ]
          ]

In[126]:= Fbits1

Out[126]= {{0, 0, 0, -}, {0, 0, -, 1}, {0, 1, 0, -}, {0, 1, -, 1}, {0, -, 0, 0},
            {0, -, 0, 1}, {0, -, 1, 1}, {1, 0, 1, -}, {1, 0, -, 0},
            {1, -, 0, 0}, {-, 0, 0, 0}, {-, 0, 1, 1}, {-, 1, 0, 0}}
```

This is close to the function we want, but we need to think ahead a bit. Recall from the description of the Quine-McCluskey process in the text that, in order to proceed with the second half of the method, we need to know which of the bit strings are prime implicants. That is, which bit strings are never used in a simplification.

We will track which bit strings are used as follows. Before the first loop, we create a set consisting of all of the bit strings in **groups**. We can do this by applying `Flatten` to **groups**, removing the sublist structure. To obtain the elements of the sublists of groups rather than just all the 0s and 1s, we need to use the option second argument of `Flatten` to specify that we wish to flatten only the first level.

```
In[127]:= Flatten[groups, 1]

Out[127]= {{0, 0, 0, 0}, {1, 0, 0, 0}, {0, 1, 0, 0}, {0, 0, 0, 1}, {1, 1, 0, 0},
           {1, 0, 1, 0}, {0, 1, 0, 1}, {0, 0, 1, 1}, {1, 0, 1, 1}, {0, 1, 1, 1}}
```

Then each time **`mergeBitstrings`** is successful, we remove the pair of bit strings from this set, using the `Complement` set operator. For example, to remove {0, 0, 0, 0} and {1, 0, 0, 0}, we would execute the following expression.

```
In[128]:= Complement[Flatten[groups, 1], {{0, 0, 0, 0}, {1, 0, 0, 0}}]

Out[128]= {{0, 0, 0, 1}, {0, 0, 1, 1}, {0, 1, 0, 0}, {0, 1, 0, 1},
           {0, 1, 1, 1}, {1, 0, 1, 0}, {1, 0, 1, 1}, {1, 1, 0, 0}}
```

The function will return the list consisting of the next level of bit strings and the prime implicants from this stage. Here is our next attempt at the function.

```
In[129]:= nextBitList1[lastgroups_List] :=
           Module[{nextL = {}, primeImps, n, A, B, a, b, m},
            primeImps = Flatten[lastgroups, 1];
            For[n = 1, n ≤ Length[lastgroups] - 1, n++,
             A = lastgroups[[n]];
             B = lastgroups[[n + 1]];
             Do[m = mergeBitstrings[a, b];
              If[m =!= False,
               nextL = Union[nextL, {m}];
               primeImps = Complement[primeImps, {a, b}]
              ]
              , {a, A}, {b, B}
             ]
            ];
            {nextL, primeImps}
           ]
```

This still isn't sufficient, however, because we also need to update the coverage dictionary as we create new bit strings. Recall that “coverage dictionary” is the name we gave to the table that records, for each bit string, which of the original minterms are covered by that bit string. The coverage dictionary was initialized with the bit strings formed from the minterms.

We can inspect the values stored with the `Definition` function.

```
In[130]:= Definition[coverageDict]

Out[130]= coverageDict[{0, 0, 0, 0}] = {10}

          coverageDict[{0, 0, 0, 1}] = {9}

          coverageDict[{0, 0, 1, 1}] = {8}

          coverageDict[{0, 1, 0, 0}] = {7}
```

```
        coverageDict[{0, 1, 0, 1}] = {6}

        coverageDict[{0, 1, 1, 1}] = {5}

        coverageDict[{1, 0, 0, 0}] = {4}

        coverageDict[{1, 0, 1, 0}] = {3}

        coverageDict[{1, 0, 1, 1}] = {2}

        coverageDict[{1, 1, 0, 0}] = {1}
```

Within our function, we need to update the coverage dictionary. To do this, we will pass it as an argument to the function. The function will be able to update the indexed variable as if it were global without it needing to be held. Consider the following, for example.

```
In[131]:= indexedExample["a"] = 1;
          indexedExample["b"] = 2;

In[133]:= Definition[indexedExample]

Out[133]= indexedExample[a] = 1

          indexedExample[b] = 2

In[134]:= functionToChangeIndexed[
            indexedVar_Symbol, i_, v_] := Module[{},
            indexedVar[i] = v;
           ]
```

The intent of this function is to add the index *i* with value *v* to the indexed variable passed as the first argument. If the first argument were a list, for example, this function would not have the desired effect, as the list would be passed to the function, modified within the `Module`, but remain unchanged outside the function. For an indexed symbol, however, when we call this function with first argument **`indexedExample`**, the argument variable **`indexedVar`** is assigned to the symbol **`indexedExample`**. It is like assigning **`indexedVar`** to the name of the indexed variable. Inside the function, the variable **`indexedVar`** is resolved to the symbol **`indexedExample`**, making the assignment within the function a “global” assignment.

```
In[135]:= functionToChangeIndexed[indexedExample, "c", 3]
```

```
In[136]:= Definition[indexedExample]

Out[136]= indexedExample[a] = 1

          indexedExample[b] = 2

          indexedExample[c] = 3
```

We update the dictionary within the **`m=!=False`** `If` statement. When we form a new bit string **`m`**, we obtain the set of minterms it covers by taking the union of the sets of minterms covered by the two bit strings that were merged. That is, **`coverDict[m]`** is the union of **`coverDict[a]`** and **`coverDict[b]`**. Note that bit strings formed beyond the first step are typically generated multiple times. However, each time they are generated they always cover the same set of original minterms.

Here is the final version of **`nextBitList`**.

```
In[137]:= nextBitList[lastgroups_List, coverDict_Symbol] :=
           Module[{nextL = {}, primeImps, n, A, B, a, b, m},
            primeImps = Flatten[lastgroups, 1];
            For[n = 1, n ≤ Length[lastgroups] - 1, n++,
             A = lastgroups[[n]];
             B = lastgroups[[n + 1]];
             Do[m = mergeBitstrings[a, b];
              If[m =!= False,
               nextL = Union[nextL, {m}];
               primeImps = Complement[primeImps, {a, b}];
               coverDict[m] = Union[coverDict[a], coverDict[b]];
              ]
              , {a, A}, {b, B}
             ]
            ];
            {nextL, primeImps}
           ]
```

We apply it to **`groups`** to obtain **`Fbits1`** and **`primes1`**.

```
In[138]:= {Fbits1, primes1} = nextBitList[groups, coverageDict]

Out[138]= {{{0, 0, 0, -}, {0, 0, -, 1}, {0, 1, 0, -},
            {0, 1, -, 1}, {0, -, 0, 0}, {0, -, 0, 1},
            {0, -, 1, 1}, {1, 0, 1, -}, {1, 0, -, 0}, {1, -, 0, 0},
            {-, 0, 0, 0}, {-, 0, 1, 1}, {-, 1, 0, 0}}, {}}
```

We see that there are

```
In[139]:= Length[Fbits1]

Out[139]= 13
```

bit strings in the second level, but no prime implicants coming from the first pass.

```
In[140]:= Length[primes1]
Out[140]= 0
```

Also, almost as a side effect, the function has updated **`coverageDict`**.

```
In[141]:= Definition[coverageDict]
Out[141]= coverageDict[{0, 0, 0, 0}] = {10}

          coverageDict[{0, 0, 0, 1}] = {9}

          coverageDict[{0, 0, 0, -}] = {9, 10}

          coverageDict[{0, 0, 1, 1}] = {8}

          coverageDict[{0, 0, -, 1}] = {8, 9}

          coverageDict[{0, 1, 0, 0}] = {7}

          coverageDict[{0, 1, 0, 1}] = {6}

          coverageDict[{0, 1, 0, -}] = {6, 7}

          coverageDict[{0, 1, 1, 1}] = {5}

          coverageDict[{0, 1, -, 1}] = {5, 6}

          coverageDict[{0, -, 0, 0}] = {7, 10}

          coverageDict[{0, -, 0, 1}] = {6, 9}

          coverageDict[{0, -, 1, 1}] = {5, 8}

          coverageDict[{1, 0, 0, 0}] = {4}

          coverageDict[{1, 0, 1, 0}] = {3}

          coverageDict[{1, 0, 1, 1}] = {2}

          coverageDict[{1, 0, 1, -}] = {2, 3}

          coverageDict[{1, 0, -, 0}] = {3, 4}
```

```
coverageDict[{1, 1, 0, 0}] = {1}

coverageDict[{1, -, 0, 0}] = {1, 4}

coverageDict[{-, 0, 0, 0}] = {4, 10}

coverageDict[{-, 0, 1, 1}] = {2, 8}

coverageDict[{-, 1, 0, 0}] = {1, 7}
```

Repeating

Step 4 is to repeat steps 2 and 3.

The **`Fbits1`** list takes the place of **`Fbits`**. We apply **`sortGroups`** to produce **`groups1`**.

```
In[142]:= groups1 = sortGroups[Fbits1]

Out[142]= {{{0, 0, 0, -}, {0, -, 0, 0}, {-, 0, 0, 0}},
           {{0, 0, -, 1}, {0, 1, 0, -}, {0, -, 0, 1},
            {1, 0, -, 0}, {1, -, 0, 0}, {-, 1, 0, 0}},
           {{0, 1, -, 1}, {0, -, 1, 1}, {1, 0, 1, -}, {-, 0, 1, 1}}, {}, {}}
```

Then applying **`nextBitList`** to **`groups1`** produces **`Fbits2`** and **`primes2`**.

```
In[143]:= {Fbits2, primes2} = nextBitList[groups1, coverageDict]

Out[143]= {{{0, -, 0, -}, {0, -, -, 1}, {-, -, 0, 0}},
           {{1, 0, 1, -}, {1, 0, -, 0}, {-, 0, 1, 1}}}
```

We see that we have found three prime implicants. The coverage dictionary was further expanded to include the new bit strings.

Do the same thing again with **`Fbits2`**.

```
In[144]:= groups2 = sortGroups[Fbits2]

Out[144]= {{{0, -, 0, -}, {-, -, 0, 0}}, {{0, -, -, 1}}, {}, {}, {}}

In[145]:= {Fbits3, primes3} = nextBitList[groups2, coverageDict]

Out[145]= {{}, {{0, -, 0, -}, {-, -, 0, 0}, {0, -, -, 1}}}
```

This time, **`Fbits3`** was empty, which indicates that no more merging is possible and all prime implicants have been found.

This part of the process concludes by forming the list of all the prime implicants.

```
In[146]:= allprimeImps = Union[primes1, primes2, primes3]

Out[146]= {{0, -, 0, -}, {0, -, -, 1}, {1, 0, 1, -},
           {1, 0, -, 0}, {-, 0, 1, 1}, {-, -, 0, 0}}
```

Forming the Coverage Table

Now that we have identified all of the prime implicants, we will use the coverage dictionary to create the coverage table.

Take a look at the final state of the coverage dictionary.

```
In[147]:= Definition[coverageDict]

Out[147]= coverageDict[{0, 0, 0, 0}] = {10}

          coverageDict[{0, 0, 0, 1}] = {9}

          coverageDict[{0, 0, 0, -}] = {9, 10}

          coverageDict[{0, 0, 1, 1}] = {8}

          coverageDict[{0, 0, -, 1}] = {8, 9}

          coverageDict[{0, 1, 0, 0}] = {7}

          coverageDict[{0, 1, 0, 1}] = {6}

          coverageDict[{0, 1, 0, -}] = {6, 7}

          coverageDict[{0, 1, 1, 1}] = {5}

          coverageDict[{0, 1, -, 1}] = {5, 6}

          coverageDict[{0, -, 0, 0}] = {7, 10}

          coverageDict[{0, -, 0, 1}] = {6, 9}

          coverageDict[{0, -, 0, -}] = {6, 7, 9, 10}

          coverageDict[{0, -, 1, 1}] = {5, 8}

          coverageDict[{0, -, -, 1}] = {5, 6, 8, 9}

          coverageDict[{1, 0, 0, 0}] = {4}

          coverageDict[{1, 0, 1, 0}] = {3}

          coverageDict[{1, 0, 1, 1}] = {2}

          coverageDict[{1, 0, 1, -}] = {2, 3}

          coverageDict[{1, 0, -, 0}] = {3, 4}
```

```
coverageDict[{1, 1, 0, 0}] = {1}

coverageDict[{1, -, 0, 0}] = {1, 4}

coverageDict[{-, 0, 0, 0}] = {4, 10}

coverageDict[{-, 0, 1, 1}] = {2, 8}

coverageDict[{-, 1, 0, 0}] = {1, 7}

coverageDict[{-, -, 0, 0}] = {1, 4, 7, 10}
```

Each bit string, and in particular each prime implicant, is an index in this table. The corresponding entry is the set of integers which are the indices to the original minterms in **`Fbits`**. Thus, to determine which of the original minterms are covered by each prime implicant, we just look it up in the table.

We will model the coverage table as a matrix. Each row corresponds to a prime implicant, so there will be

```
In[148]:= Length[allprimeImps]

Out[148]= 6
```

rows. And each column corresponds to a minterm, so there are

```
In[149]:= Length[Fbits]

Out[149]= 10
```

columns. The entries in the matrix will be 0s and 1s with 1 in position (*i*, *j*) indicating that the prime implicant at position *i* in **`allprimeImps`** covers the minterm at position *j* in **`Fbits`**.

We use `ConstantArray` with first argument 0 and second argument a list consisting of the number of rows and columns to form a matrix of all 0s of the appropriate size.

```
In[150]:= ConstantArray[0, {6, 10}] // MatrixForm

Out[150]//MatrixForm=
  0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0
  0 0 0 0 0 0 0 0 0 0
```

To enter 1s in the appropriate positions, we loop over the rows, considering each prime implicant in turn. For each prime implicant, we look up its entry in the coverage dictionary to obtain the set of minterms it covers. For each of those minterms, we place a 1 in the matrix.

The following function initializes the coverage table.

```
In[151]:= initCoverageMatrix[minterms_List, primeImps_List,
            coverDict_Symbol] := Module[{M, i, C, j},
            M = ConstantArray[0, {Length[primeImps], Length[minterms]}];
            For[i = 1, i ≤ Length[primeImps], i++,
             C = coverDict[primeImps[[i]]];
             Do[M[[i, j]] = 1, {j, C}]
            ];
            M
           ]
```

Applied to our example, this produces the following coverage table.

```
In[152]:= coverageTable =
            initCoverageMatrix[Fbits, allprimeImps, coverageDict];
          MatrixForm[coverageTable]

Out[153]//MatrixForm=
```

$$\begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

Manipulating the Matrix

Once the coverage table is set up, we move to steps 6 and 7, determining which prime implicants to include in the minimal expression. In step 6, we identify the essential prime implicants and in step 7 we identify which of the non-essential prime implicants we will include. We will see how to identify the prime implicants to use in a moment.

To aid in performing both steps 6 and 7, we will be manipulating the coverage table. Once we have decided to include a particular prime implicant in the minimal expression, we take three actions.

First, record the decision by adding the prime implicant to a new list, say **`minBits`**, the list of bit strings to be included in the minimal expression.

Second, delete that prime implicant's row from the coverage table and delete the columns corresponding to the minterms it covered. We know the prime implicant will be in the expression and thus the minterms it covers are satisfied. Hence, there is no longer any need to keep track of that information.

Third, delete the prime implicant and the minterms it covers from the lists storing them (**`allprimeImps`** and **`Fbits`**). This is to ensure that the indices of **`allprimeImps`** and **`Fbits`** continue to match the row and column numbers of the matrix.

We will now write a function that implements these actions. Its input will be the index to the prime implicant that has been chosen. It will also accept the names of the coverage matrix, the list of minterms, and the list of prime implicants. All of these will be modified in the function (refer to the subsection on “Modifying arguments” above). The function will return the bit string of the prime implicant that was chosen.

Our function will be called **`updateCT`**, for “update coverage table.” The **`minBits`** list, the list of chosen prime implicants, will be updated via the return value. This accomplishes the first task for this function.

Second, we must delete the row corresponding to the chosen prime implicant and the columns corresponding to the minterms covered by that implicant. Suppose, in our example, that we have decided to include the fourth prime implicant in the final result. This is:

```
In[154]:= allprimeImps[[4]]

Out[154]= {1, 0, -, 0}
```

From **coverageTable**, we need to remove row 4 (since this corresponds to the prime implicant). We also need to remove the columns corresponding to the minterms covered by this prime implicant. To determine which columns are to be removed, we find the locations of the 1s in the row of the matrix.

To determine the locations of the 1s, we'll loop over the columns checking each position in row 4 to see if it is 1 or not. We use the `Dimensions` function to determine the number of columns. For a matrix, `Dimensions` returns a list whose elements are the number of rows and columns.

```
In[155]:= covered = {};
          For[i = 1, i ≤ Dimensions[coverageTable][[2]], i++,
            If[coverageTable[[4, i]] == 1, AppendTo[covered, i]]
           ];
          covered

Out[157]= {3, 4}
```

We now know that we need to remove row 4 and columns 3 and 4. To do this, we will use a complicated selection. Recall that `Part` (`[[…]]`) can be used with a list within the double brackets to select specific elements. For example,

```
In[158]:= {"a", "b", "c", "d", "e", "f", "g"}[[{1, 2, 4, 5, 6}]]

Out[158]= {a, b, d, e, f}
```

The same is true for matrices. Given a matrix, a list of rows, and a list of columns, you can obtain the submatrix containing only the specified rows and columns as illustrated below.

```
In[159]:= exampleMatrix = Table[a{i,j}, {i, 7}, {j, 6}];
          exampleMatrix // MatrixForm
```

Out[160]//MatrixForm=

$$\begin{pmatrix} a_{\{1,1\}} & a_{\{1,2\}} & a_{\{1,3\}} & a_{\{1,4\}} & a_{\{1,5\}} & a_{\{1,6\}} \\ a_{\{2,1\}} & a_{\{2,2\}} & a_{\{2,3\}} & a_{\{2,4\}} & a_{\{2,5\}} & a_{\{2,6\}} \\ a_{\{3,1\}} & a_{\{3,2\}} & a_{\{3,3\}} & a_{\{3,4\}} & a_{\{3,5\}} & a_{\{3,6\}} \\ a_{\{4,1\}} & a_{\{4,2\}} & a_{\{4,3\}} & a_{\{4,4\}} & a_{\{4,5\}} & a_{\{4,6\}} \\ a_{\{5,1\}} & a_{\{5,2\}} & a_{\{5,3\}} & a_{\{5,4\}} & a_{\{5,5\}} & a_{\{5,6\}} \\ a_{\{6,1\}} & a_{\{6,2\}} & a_{\{6,3\}} & a_{\{6,4\}} & a_{\{6,5\}} & a_{\{6,6\}} \\ a_{\{7,1\}} & a_{\{7,2\}} & a_{\{7,3\}} & a_{\{7,4\}} & a_{\{7,5\}} & a_{\{7,6\}} \end{pmatrix}$$

```
In[161]:= exampleMatrix[[{1, 2, 4, 6, 7}, {1, 2, 3, 6}]] // MatrixForm
```

Out[161]//MatrixForm=

$$\begin{pmatrix} a_{\{1,1\}} & a_{\{1,2\}} & a_{\{1,3\}} & a_{\{1,6\}} \\ a_{\{2,1\}} & a_{\{2,2\}} & a_{\{2,3\}} & a_{\{2,6\}} \\ a_{\{4,1\}} & a_{\{4,2\}} & a_{\{4,3\}} & a_{\{4,6\}} \\ a_{\{6,1\}} & a_{\{6,2\}} & a_{\{6,3\}} & a_{\{6,6\}} \\ a_{\{7,1\}} & a_{\{7,2\}} & a_{\{7,3\}} & a_{\{7,6\}} \end{pmatrix}$$

This is the matrix that results from deleting the 3rd and 5th rows, and the 4th and 5th columns.

So for the coverage table, we can obtain the modified matrix by forming the lists of row and column numbers that should remain and then apply Part (`[[…]]`).

```
In[162]:= remainingRows = {1, 2, 3, 5, 6}

Out[162]= {1, 2, 3, 5, 6}

In[163]:= remainingColumns = Complement[Range[Length[Fbits]], covered]

Out[163]= {1, 2, 5, 6, 7, 8, 9, 10}

In[164]:= coverageTable[[remainingRows, remainingColumns]] // MatrixForm
```

Out[164]//MatrixForm=

$$\begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

The reader is encouraged to compare this to the original matrix. Note that in this example, we have not actually modified **`coverageTable`**.

The last tasks are to remove the selected prime implicant from the list of prime implicants, and remove the covered minterms from its list. Note that **`remainingRows`** and **`remainingColumns`** are indices into the **`allprimeImps`** list and the list of minterms. Applying Part (`[[…]]`) produces the desired outcome.

```
In[165]:= allprimeImps[[remainingRows]]

Out[165]= {{0, -, 0, -}, {0, -, -, 1}, {1, 0, 1, -}, {-, 0, 1, 1}, {-, -, 0, 0}}

In[166]:= Fbits[[remainingColumns]]

Out[166]= {{1, 1, 0, 0}, {1, 0, 1, 1}, {0, 1, 1, 1}, {0, 1, 0, 1},
            {0, 1, 0, 0}, {0, 0, 1, 1}, {0, 0, 0, 1}, {0, 0, 0, 0}}
```

Here is the function.

```
In[167]:= SetAttributes[updateCT, {HoldRest}];
          updateCT[newPI_, coverTable_, minterms_, primeImps_] :=
           Module[{newPIbitstring, numRows, numCols,
             covered, i, remainingRows, remainingCols},
            newPIbitstring = primeImps[[newPI]];
            {numRows, numCols} = Dimensions[coverTable];
            covered = {};
            For[i = 1, i ≤ numCols, i++,
             If[coverTable[[newPI, i]] == 1, AppendTo[covered, i]]
            ];
            remainingRows = Delete[Range[numRows], newPI];
            remainingCols = Complement[Range[numCols], covered];
            coverTable = coverTable[[remainingRows, remainingCols]];
            primeImps = primeImps[[remainingRows]];
            minterms = minterms[[remainingCols]];
            newPIbitstring
           ]
```

Finding Essential Prime Implicants

Next we write a function to identify the essential prime implicants. Recall that a prime implicant is essential when it is the only prime implicant to cover some minterm. In terms of the coverage table, this is equivalent to the existence of a column with only one 1.

We will locate the essential prime implicants as follows. First, we initialize the set of essential prime implicants to the empty list.

We proceed in a manner similar to the **mergeBitstrings** function. We use a `For` loop to step through the columns of the coverage table. Within this loop, we initialize a symbol, **rowhas1**, to 0.

We then enter a second loop to step through the entries in the columns. When a 1 entry has been found, we check **rowhas1.** If that symbol is 0, then it is assigned to the current row number. If it is not 0, then we have found a second 1 in the column and we assign **rowhas1** to -1 and use `Break` to terminate the inner loop. After the inner loop, we test **rowhas1**. If it is positive, then we know that only one 1 was located in that column, and hence the row the solitary 1 was found in corresponds to an essential prime implicant. In this case, we add the row number (**rowhas1**) to essentials.

The following function implements this algorithm and returns the list of essential prime implicants.

```
In[169]:= findEssentials[coverTable_List] :=
           Module[{essentials = {}, r, c, i, j, rowhas1},
            {r, c} = Dimensions[coverTable];
            For[i = 1, i ≤ c, i++,
             rowhas1 = 0;
             For[j = 1, j ≤ r, j++,
              If[coverTable[[j, i]] == 1,
               If[rowhas1 == 0,
                rowhas1 = j,
                rowhas1 = -1; Break[]
               ]
              ]
             ];
             If[rowhas1 > 0,
              AppendTo[essentials, rowhas1]
             ]
            ];
            Sort[Union[essentials], Greater]
           ]
```

The Sort applied with the comparison function Greater ensures that the output will be in decreasing order. This will allow us to update the coverage table and other lists without affecting the index of smaller-indexed essential prime implicants.

We use this to determine the essential prime implicants of our example.

```
In[170]:= essentialPIs = findEssentials[coverageTable]

Out[170]= {6, 2}
```

Now that we have the essential prime implicants, we can initialize **minBits** and apply **updateCT** to the essential prime implicants.

```
In[171]:= minBits = {}

Out[171]= {}

In[172]:= Do[AppendTo[minBits,
            updateCT[i, coverageTable, Fbits, allprimeImps]]
           , {i, essentialPIs}]

In[173]:= minBits

Out[173]= {{-, -, 0, 0}, {0, -, -, 1}}
```

```
In[174]:= coverageTable // MatrixForm
```

Out[174]//MatrixForm=

$$\begin{pmatrix} 0 & 0 \\ 1 & 1 \\ 0 & 1 \\ 1 & 0 \end{pmatrix}$$

Completing the Coverage

Provided that the essential prime implicants did not completely cover the original minterms, we must complete the coverage with non-essential prime implicants. First, we ensure that the coverage is not complete by checking the column dimension.

```
In[175]:= Dimensions[coverageTable][[2]] > 0

Out[175]= True
```

We will use a heuristic approach to find a minimal set of prime implicants rather than using an exhaustive search to determine the minimum. The heuristic we use will be to choose the prime implicant with the most extensive coverage of the remaining minterms.

To find such a prime implicant, we will do the following. First, initialize **`maxCoverage`** and **`bestImp`** both to 0. Then loop over each row of the (modified) coverage table. For each row, we will compute the sum of the entries. If this sum is greater than **`maxCoverage`**, then set **`maxCoverage`** to the sum and set **`bestImp`** to the row number. Once the loop is complete, **`bestImp`** will be the index to a row with maximum coverage and will be the next prime implicant added to the **`minBits`** list.

Here is the function that implements this strategy.

```
In[176]:= findBestImp[coverTable_] :=
           Module[{maxCoverage = 0, bestImp = 0, i, j, sum},
            For[i = 1, i ≤ Dimensions[coverTable][[1]], i++,
             sum = Plus @@ coverTable[[i]];
             If[sum > maxCoverage,
              maxCoverage = sum;
              bestImp = i
             ]
            ];
            bestImp
           ]
```

As long as the coverage table is not empty, we apply this function to it to obtain the next implicant. We add the implicant to the list **`minBits`** representing the minimal expression and update the coverage table using **`updateCT`**.

```
In[177]:= While[Dimensions[coverageTable][[2]] > 0,
           nextPI = findBestImp[coverageTable];
           AppendTo[minBits,
            updateCT[nextPI, coverageTable, Fbits, allprimeImps]]
          ]
```

```
In[178]:= minBits

Out[178]= {{-, -, 0, 0}, {0, -, -, 1}, {1, 0, 1, -}}
```

All that's left is to translate **`minBits`** back into a logical expression. This can be done using **`bitListToDNF`** created earlier.

```
In[179]:= bitListToDNF[minBits, {w, x, y, z}]

Out[179]= (! y && ! z) || (! w && z) || (w && ! x && y)
```

Putting It All Together

Finally, we assemble the pieces into a single function, which accepts a logical expression in disjunctive normal form and a list of its variables. It returns a minimal equivalent expression.

```
In[180]:= quineMcCluskey[F_, variables_] :=
           Module[{fBits, fBitsL, coverageDict, groupsL,
             primesL, newFbits, newprimes, i, allprimeImps, j,
             coverageTable, essentialPIs, minBits, nextPI},
            fBits = dnfToBitList[F, variables];
            initCoverDict[coverageDict, fBits];
            i = 0;
            fBitsL = {fBits};
            groupsL = {};
            primesL = {};
            While[fBitsL[[-1]] =!= {},
             AppendTo[groupsL, sortGroups[fBitsL[[-1]]]];
             {newFbits, newprimes} =
              nextBitList[groupsL[[-1]], coverageDict];
             AppendTo[fBitsL, newFbits];
             AppendTo[primesL, newprimes];
            ];
            allprimeImps = Union @@ primesL;
            coverageTable =
             initCoverageMatrix[fBits, allprimeImps, coverageDict];
            essentialPIs = findEssentials[coverageTable];
            minBits = {};
            Do[AppendTo[minBits,
              updateCT[i, coverageTable, fBits, allprimeImps]]
             , {i, essentialPIs}];
            While[MatrixQ[coverageTable] &&
              Dimensions[coverageTable][[2]] > 0,
             nextPI = findBestImp[coverageTable];
             AppendTo[minBits,
              updateCT[nextPI, coverageTable, fBits, allprimeImps]]
            ];
            bitListToDNF[minBits, variables]
           ]
```

Define **ex10** to be the expression in Example 10 from Section 12.4 of the text.

```
In[181]:= ex10 = (w && x && y && ! z) || (w && ! x && y && z) ||
            (w && ! x && y && ! z) || (! w && x && y && z) || (! w && x && ! y && z) ||
            (! w && ! x && y && z) || (! w && ! x && ! y && z)

Out[181]= (w && x && y && ! z) || (w && ! x && y && z) ||
           (w && ! x && y && ! z) || (! w && x && y && z) || (! w && x && ! y && z) ||
           (! w && ! x && y && z) || (! w && ! x && ! y && z)
```

```
In[182]:= quineMcCluskey[ex10, {w, x, y, z}]

Out[182]= (w && y && ! z) || (! w && z) || (w && ! x && y)
```

Note that this is the first of the two answers given in the solution to Example 10.

Solutions to Computer Projects and Computations and Explorations

Computer Projects 2

Construct a table listing the set of values of all 256 Boolean functions of degree three.

Solution: The Boolean functions of degree three are in one-to-one correspondence with the subsets of {true, false}^{3}. This is because each subset S of {true, false}^{3} can be identified with the unique Boolean function of degree three that returns true on the members of S and false on all other inputs.

The set {true, false}^{3} consists of 8 elements: (true, true, true), (true, true, false), (true, false, true),..., (false, false, false). The power set can therefore be identified with bit strings of length 8, with a 1 indicating inclusion of the corresponding member of {true, false}^{3}. For example, "10100000" would correspond to the set {(true, true, true), (true, false, true)} which in turn corresponds to the Boolean function that returns true on (true, true, true) and (true, false, true) and false for all other input. The bit strings, in turn, can be identified with integers between 0 and 255, based on their binary representation.

Mathematica's `BooleanFunction` function takes advantage of this correspondence between integers and Boolean functions. Given two integers k and n as arguments, `BooleanFunction` produces the Boolean function, as a pure `Function`, on n variables corresponding to the binary representation of k.

```
In[183]:= BooleanFunction[132, 3]

Out[183]= BooleanFunction[ < 3 >]
```

The resulting `BooleanFunction` can be applied to truth values.

```
In[184]:= BooleanFunction[132, 3][True, False, True]

Out[184]= False
```

With a list of variables passed as a third argument, the output will be an expression for the Boolean function.

```
In[185]:= BooleanFunction[132, 3, {x, y, z}]

Out[185]= (x && y && z) || (! x && y && ! z)
```

Applying `BooleanTable`, described in the first section of this chapter, to the output of `BooleanFunction` produces the functions' truth table.

```
In[186]:= BooleanTable[BooleanFunction[132, 3]]

Out[186]= {True, False, False, False, False, True, False, False}
```

We prefer a table that shows the input values along with the output. For this, we apply `BooleanTable` to a list consisting of variables as the second argument. Then in the first argument, we can use the variables.

```
In[187]:= BooleanTable[
            {x, y, z, BooleanFunction[132, 3][x, y, z]}, {x, y, z}]

Out[187]= {{True, True, True, True}, {True, True, False, False},
            {True, False, True, False}, {True, False, False, False},
            {False, True, True, False}, {False, True, False, True},
            {False, False, True, False}, {False, False, False, False}}
```

The third element of the output indicates that the function 132 returns false on (true, false, true).

We can now replace the specific application of `BooleanFunction` with a `Table` to allow the integer to range. For demonstration purposes, we restrict the range to 100 to 105. We also apply `TableForm` with the `TableDepth` option set to 1 to make the output readable.

```
In[188]:= TableForm[BooleanTable[
              {x, y, z, Table[BooleanFunction[k, 3][x, y, z], {k, 100, 105}]},
              {x, y, z}], TableDepth → 1]

Out[188]//TableForm=
          {True, True, True, {False, False, False, False, False, False}}
          {True, True, False, {True, True, True, True, True, True}}
          {True, False, True, {True, True, True, True, True, True}}
          {True, False, False, {False, False, False, False, False, False}}
          {False, True, True, {False, False, False, False, True, True}}
          {False, True, False, {True, True, True, True, False, False}}
          {False, False, True, {False, False, True, True, False, False}}
          {False, False, False, {False, True, False, True, False, True}}
```

The output above indicates that, on the input (True, False, True), all 6 Boolean functions associated to the integers 100 through 105 output true.

Computations and Explorations 6

> Randomly generate 10 different Boolean expressions in four variables and determine the average number of steps required to minimize them using the Quine-McCluskey method.

Solution: To solve this problem, we need to find a way to generate random Boolean expressions, and then we must find a method of examining the minimization process so that we can count the number of steps.

Using what we learned about `BooleanFunction` in the solution to Computer Projects 2 above, we can produce random Boolean functions by applying `BooleanFunction` to random integers. For an expression in four variables, there are $2^{2^4} = 65\,536$ different Boolean functions. So we choose 10 different integers between 0 and 65535 using `RandomSample`. Recall that `RandomSample` accepts a list of elements, which in this case will be obtained from `Range`, and a positive integer. It returns a list of the specified number of elements randomly selected from the list of objects.

```
In[189]:= RandomSample[Range[0, 65 535], 10]

Out[189]= {6993, 6632, 36 964, 58 324,
           61 650, 17 988, 9289, 28 671, 21 165, 4861}
```

Applying `BooleanFunction` to each of the values output by `RandomSample` will produce a list of 10 randomly chosen Boolean expressions.

```
In[190]:= Table[BooleanFunction[k, 4, {x, y, z, w}],
            {k, RandomSample[Range[0, 65 535], 10]}] // TableForm

Out[190]//TableForm=
          (w && ! x && ! y && ! z) || (! w && y && ! z) || (! w && ! y && z) || (x && y &&
          (w && y) || (! w && ! y) || (x && ! y) || (y && z)
          (w && y) || (w && z) || (! w && ! x && ! z) || (x && ! y && z)
          (x && ! y) || (x && ! z) || (! y && ! z)
          (w && x && ! y && z) || (w && ! x && ! z) || (! w && ! y && ! z) || (! x && ! y &
          (w && y) || (! x && ! y) || ! z
          (w && ! z) || x || y
          (w && ! x && z) || (! w && x && y) || (! w && ! y && ! z) || (x && y && ! z)
          (w && ! x && y && z) || (! w && ! x && ! y) || (! w && ! x && ! z) || (x && ! y &
          (w && ! x) || (! w && x) || (x && ! y) || (! y && z)
```

Having determined how to generate random expressions, we need to find a way to count the number of steps taken during the minimization process. There are several approaches we could take to this part of the problem.

The first is to measure the time taken to execute the procedure. We have done this many times before.

```
In[191]:= QMtimes = {};
          randExprs = Table[BooleanFunction[k, 4, {x, y, z, w}],
             {k, RandomSample[Range[0, 65 535], 10]}];
          For[i = 1, i ≤ 10, i++,
            AppendTo[QMtimes,
             Timing[quineMcCluskey[randExprs[[i]], {x, y, z, w}]][[1]]]
           ];
          Mean[QMtimes]

Out[194]= 0.000904
```

The second approach is to modify the implementation of Quine-McCluskey to count the number of times certain operations are called. For example, we may be interested in the number of times that the **updateCT** procedure is executed. In this case, we can alter **quineMcCluskey** to include a variable that is incremented at the start of every execution of **updateCT**.

```
In[195]:= quineMcCluskeycountCT[F_, variables_] :=
           Module[{fBits, fBitsL, coverageDict, groupsL,
             primesL, newFbits, newprimes, i, allprimeImps, j,
             coverageTable, essentialPIs, minBits, nextPI, count = 0},
            fBits = dnfToBitList[F, variables];
            initCoverDict[coverageDict, fBits];
            i = 0;
            fBitsL = {fBits};
            groupsL = {};
            primesL = {};
            While[fBitsL[[-1]] =!= {},
             AppendTo[groupsL, sortGroups[fBitsL[[-1]]]];
             {newFbits, newprimes} =
              nextBitList[groupsL[[-1]], coverageDict];
             AppendTo[fBitsL, newFbits];
             AppendTo[primesL, newprimes];
            ];
            allprimeImps = Union @@ primesL;
            coverageTable =
             initCoverageMatrix[fBits, allprimeImps, coverageDict];
            essentialPIs = findEssentials[coverageTable];
            minBits = {};
            Do[AppendTo[minBits,
              updateCT[i, coverageTable, fBits, allprimeImps]];
             count ++
             , {i, essentialPIs}];
            While[MatrixQ[coverageTable] &&
              Dimensions[coverageTable][[2]] > 0,
             nextPI = findBestImp[coverageTable];
             AppendTo[minBits,
              updateCT[nextPI, coverageTable, fBits, allprimeImps]]
            ];
            bitListToDNF[minBits, variables];
            count
           ]
```

Now execute **quineMcCluskeycountCT** on 10 random expressions.

```
In[196]:= QMtotal = 0;
          randExprs = Table[BooleanFunction[k, 4, {x, y, z, w}],
             {k, RandomSample[Range[0, 65 535], 10]}];
          For[i = 1, i ≤ 10, i++,
            QMtotal = QMtotal +
               quineMcCluskeycountCT[randExprs[[i]], {x, y, z, w}];
           ];
          N[QMtotal / 10]

Out[199]= 4.9
```

Exercises

1. Use *Mathematica* to verify De Morgan's Laws and the commutative and associative laws. (See Table 5 of Section 12.1.)
2. Construct truth tables for each of the following pairs of Boolean expressions and decide whether they are logically equivalent.
   - **a.** $a \to b$ and $b \to a$
   - **b.** $a \to \overline{b}$ and $b \to \overline{a}$
   - **c.** $a + b\,c$ and $(a + b + d)\,(a + c + d)$
3. Write a *Mathematica* function that, given a Boolean function, represents this function using only the `Nand` operator.
4. Use the function in the previous exercise to represent the following Boolean functions using only the `Nand` operator.
   - **a.** $F(x, y, z) = x\,y + \overline{y}\,z$
   - **b.** $G(x, y, z) = x + \overline{x}\,y + \overline{y\,z}$
   - **c.** $H(x, y, z) = x\,y\,z + \overline{x\,y\,z}$
5. Write a *Mathematica* function that, given a Boolean function, represents this function using only the `Nor` operator.
6. Use the function in the previous exercise to represent the Boolean functions in Exercise 4 using only the `Nor` operator.
7. Write a *Mathematica* function for determining the output of a threshold gate, given the values of n Boolean variables as input, and given the threshold value and a set of weights for the threshold gate. (See the Supplementary Exercises of Chapter 12 for information on threshold gates.)
8. Develop a *Mathematica* function that, given a Boolean function in four variables, determines whether it is a threshold function, and if so, finds the appropriate threshold gate representing this function. (See the Supplementary Exercises of Chapter 12.)
9. A Boolean expression e is called self dual if it is logically equivalent to its dual e^d. Write a *Mathematica* function to test whether a given expression is self dual.

10. Determine, for each integer $n \in \{1, 2, 3, 4, 5, 6\}$, the total number of Boolean functions of n variables and the number of those functions that are self dual.

11. Write a *Mathematica* function that, given a positive integer n, constructs a list of all Boolean functions of degree n. Use your function to find all Boolean functions of degree 4. Do not use `BooleanFunction`.

12. At the end of Section 12.3 of this manual, it was suggested that the procedure for producing tree representations of logical circuits could be improved by combining successive and or or gates into gates accepting multiple inputs. Implement this.

13. Use `BooleanFunction` to compute a minimal sum of products expansion for the Boolean functions with don't care conditions specified by the Karnaugh maps shown in Exercises 30 through 32 of Section 12.4.

14. Use the function you wrote in Exercise 9 to write a *Mathematica* function to generate random Boolean expressions in 4 variables and stop when it is has found one that is self dual. Run the program several times and time it. Find the average number of random expressions needed before stopping. Repeat for Boolean expressions in 5 and 6 variables. Can you make any conjectures from this information?

15. Modify **`quineMcCluskey`** to allow for don't care conditions.

16. Modify **`quineMcCluskey`** to use backtracking instead of the heuristic approach in order to determine the expression with the minimum number of terms. Use a large number of randomly generated expressions to compare the old function with the new and determine how often the heuristic produces non-optimal output.