Chapter 4

Synthesis of Two-Level Circuits

As we saw in Chapter 2, there are several points to be considered in formulating a synthesis problem:

- **Specifications.** Formal specifications are required as the starting point for synthesis. *Behavior* as well as *constraints* should be expressed. In the behavior specification, we should try to capture all possible degrees of freedom, to provide the algorithms with the largest search space that is possible.

- **Constraints.** Constraints may concern speed, testability, types of available packages, power dissipation, reliability, etc. Whenever possible, we try to incorporate the constraints in the algorithmic formulation of the problem.

- **Cost Function.** Should take into account as many factors as possible: cost of fabricating the chip, cost of testing it, cost of the package, etc. Each component of the cost depends on many factors, that are often difficult to estimate from the specifications. We shall see that we often resort to fairly crude approximations of these factors.

4.1 Design Optimality

As in the design of most complex systems, circuit designers usually have to tradeoff one design objective for another. Synthesis tools and designers try to make this tradeoff optimally, as discussed briefly in Section 1.4.1. We treat this subject at greater length here.

For example, often a designer tries to find the fastest possible circuit equivalent to a given previously designed circuit. It is often the case that overall power dissipation is strongly correlated to delay, so in seeking a faster circuit he is willing to incur a power dissipation penalty. Low power may itself be a paramount design consideration, as it is in portable computers and telecommunication devices.

However, to meet this objective, he may or may not have settle for a larger circuit. This is a typical design tradeoff, but how does he know he cannot improve both area and speed? The answer to this question depends on the optimality of the existing design. A typical design process is illustrated in Figure 4.1. In the figure, the original infeasible design (Design 0)is marked by an ellipse, the feasible design (Design 1) is

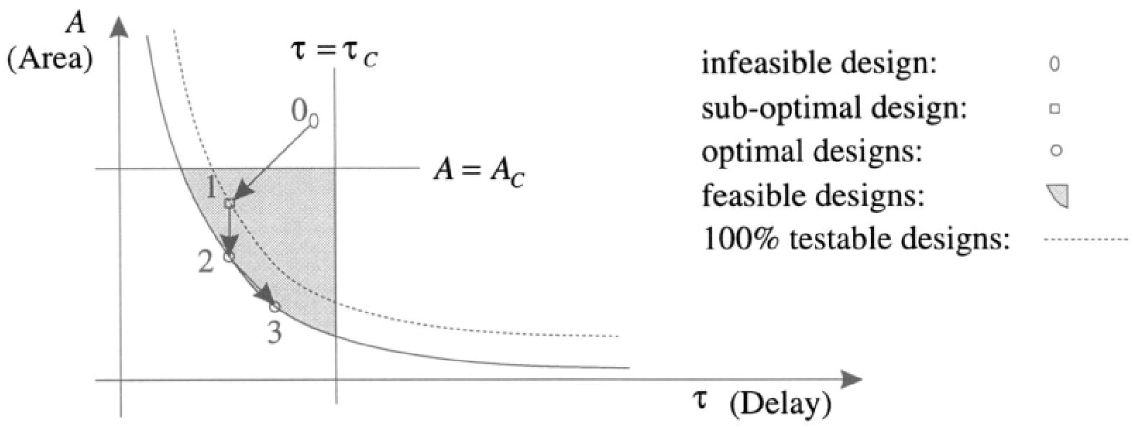

Figure 4.1: Tradeoff of area for speed for optimal designs.

represented by an small open square, and the optimal designs (Designs 2 and 3) are marked by circles. Typically the design starts out as a sub-optimal "first cut" design, which is furthest from the optimal tradeoff curve (solid line) shown in the figure. If the initial design is sub-optimal, it will be possible to redesign the circuit so as to decrease both delay and area, as in the design move from Design 0 to Design 1. As the design is improved, successive redesign iterations leave the design closer to the tradeoff curve, until a design is reached for which any attempt to increase speed (that is, decrease delay) will result in increased area. The locus of such points is called the **Pareto Critical Set** [208], and is represented by the solid curve.

At this point, the designer must establish his or her own priority, or design policy. If as is often the case, speed is the paramount consideration, the designer will continually iterate the design until a satisfactory speed objective is reached. In this process, the design moves along the critical set to the right.

Often however, the circuit eventually meets an area upper bound constraint A_U. For example, if the design becomes to large, it won't fit on a single chip. In this case, the design finalizes on the design point at the lower right. Sometimes, the designer finds that subsequent decreases in delay are not worth the area penalty they incur, so an intermediate point like the design point on the central "knee" of the curve is chosen.

It is sometimes the case that low area is the paramount design objective, so the uppermost (optimal) design point is to be chosen. Also it may be the case that delay and power are not really well correlated, so the optimal tradeoff curve becomes a 3-dimensional surface.

In any event, the purpose of logic synthesis tools is to aid the designer in reaching the optimal tradeoff curve. It is to be emphasized, however, that only the designer can set the priority for trading off area for delay and/or power dissipation.

Another complication is the issue of testability. The process of determining the testability of a design and finding all the appropriate tests (Cf., Chapter 12) for an adequate level of product assurance may be so expensive, so that design with 100% testability is sought. The family of such designs has its own tradeoff curve, generally higher that the optimal tradeoff curve, as illustrated by the dashed line in Figure 4.1.

Designs on this curve are likely to be redundant. For example, a carry-look-ahead adder has extra circuitry to get a fast output carry even though the carry propagation circuitry of a simpler ripple-carry-adder is still necessary for logic functionality. "Synthesis for Testability" tools exist whose main objective is to get onto (and stay on) this second tradeoff curve — for example, such tools might stop Design 1, the interior feasible point of the design curve, often generating all the required tests as a byproduct of this optimization.

Synthesis algorithms deal with a model of the problem, rather than the problem itself. If we want to use synthesis, we have to perform a modeling step, that converts the most important features of the design problem into features of the (mathematical) model.

We begin by looking at a restricted style of design for combinational circuits—two-level implementations—and its associated model. In spite of the restriction of two-level logic, we shall be able to derive techniques that are of very general applicability in synthesis.

4.2 Two-Level Logic

Two levels of logic are the minimum required to implement an arbitrary Boolean function. Here we assume that the primitives are AND and OR gates. AND gates are used at the first level and OR gates are used in the second level. Inverters may be present at some inputs of the gates of the first level, but we shall not count them as an additional level. Other choices are possible. In particular we could reverse the role of AND and OR gates, we could use all NAND gates or all NOR gates, or we could employ XOR gates at the second level. Other choices are possible as well.

There are two main reasons why we may want to implement a circuit in two levels, rather than multiple levels:

- Speed;
- Simplicity.

The delay of a network depends on several factors. The numbers of logic stages a signal must go through is among the important ones. So, two-level implementations tend to be fast. Notice, however, that reducing the number of levels may increase the fanin and fanout counts of gates. This may adversely impact speed.

Simplicity comes in two flavors with two-level networks. Two-level networks are easier to design and analyze, because the solution space is greatly restricted, and are easier to implement, because there are simple implementations schemes.

Historically, two-level circuits have been popular in the fifties, because they were the only circuits for which effective systematic design procedures were known. The algorithms for the optimum implementation of two-level functions were developed in the early fifties [221, 222, 188].

About twenty years later, the interest in the field was re-kindled by the advent of programmable logic: PLAs and PALs [101]. PLAs and PALs offered flexibility as well as the ability to automate a large part of the design, or even to customize the function on the field (especially PALs). Figure 4.2 shows the organization of an NMOS NAND-NAND PLA, that was popular in the seventies and early eighties.

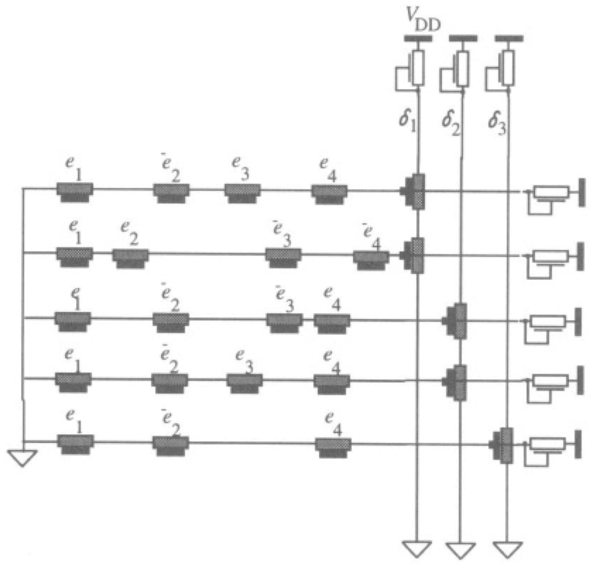
V_{DD}
δ_1
δ_2
δ_3
e_1 $\bar{e}_2$ $\bar{e}_3$ e_4
e_1 e_2 $\bar{e}_3$ $\bar{e}_4$
e_1 $\bar{e}_2$ $\bar{e}_3$ e_4
e_1 $\bar{e}_2$ e_3 e_4
e_1 $\bar{e}_2$ e_4

Figure 4.2: NMOS NAND-NAND PLA.

This particular architecture was fast and compact, though limited in the maximum number of inputs.

4.2.1 Cost Functions for Two-Level Implementations

Back in the fifties and sixties, it was customary to evaluate an implementation according to the number of diodes required to fabricate it. Later, people began using the number of gates and the number of gate inputs as criteria. This was a good reflection of the cost of the circuit in technologies like TTL.

In PLAs the area is primarily related to the number of product terms, which is in one-to-one correspondence with the number of rows of the array. So, in the seventies, the number of product term became a popular way to measure the cost of an implementation. Notice, however, that speed, testability, and folding (a layout optimization technique sometimes used with PLAs), all benefit from a sparser array. Hence, it is advantageous to use the number of gate inputs as a secondary criterion.

The advent of CMOS[1] and semi-custom design methodologies has marked a decline in the popularity of PLAs and PALs. (The former replaced by standard cells, the latter by more sophisticated forms of programmable logic like FPGAs.) When implementing a circuit with standard cells, it is customary to use multi-level implementations. The cost of a multi-level implementation is not directly related to the cost of an equivalent two-level circuit, but the role of the two-level techniques is still important, as we shall see. The most widely used model for the optimization of multi-level logic is actually a network whose nodes represent functions. These functions are often represented as two-level circuits. More on this in Chapter 10.

When minimizing a piece of logic for a subsequent multi-level implementation, the cost function tries to guide the optimization process towards a function that can

[1]CMOS PLAs must be dynamic in order not to draw static current; however, semi-custom design styles favor static circuitry.

be easily factored. It turns out that minimizing for number of gates and gate inputs normally provides a good starting point, especially for the testability properties of the resulting circuit.

4.2.2 Minimality and Testability

The previous overview shows that, with the partial exception of the diode count, the number of gates and the number of gate inputs has enjoyed a fairly constant success as a measure of the cost of a two-level implementation. One additional reason that we should mention is related to the concepts of testability and irredundancy.

Since a proof of correctness cannot be obtained from a black-box experiment, the testing of digital circuits is normally accomplished by checking each part for a predefined list of possible defects. Test generation is the process of finding the input sequences that cause the defect to manifest itself at the output of the circuit, in the form of errors.

As with synthesis, a good deal of modeling is required, so that the process may be carried out successfully for non-trivial devices. In particular, one idealizes to some extent the defects that may actually occur in a circuit. The most popular *fault model* is the so-called *stuck-at* (stuck-at-0, stuck-at-1) fault model. A stuck-at fault occurs when a connection (either a gate output or a gate input) is permanently stuck at one of the two logic levels. A multiple stuck-at fault is the simultaneous presence of several single stuck-at faults. From this point on we shall consider single faults unless otherwise specified. The tasks of identifying and generating tests for stuck-at faults will be discussed in detail in Chapter 12 — here we limit our discussion to a brief treatment of the connection between logic optimization and testability.

Notice that a stuck-at fault may be seen as transforming a circuit into another circuit of lower cost, according to *our* cost function. If there is no test for a given fault, then the fault is *untestable.* The connection affected by the fault is *redundant,* since the circuit can be simplified by removing the the connection itself. This link between redundant connections and untestable faults is the reason why we sometimes refer to redundant (or irredundant) faults.[2]

It is desirable to have 100% testable circuits (albeit for a restricted fault model): Hence it is good for the cost function to reflect the testability of the circuit. Specifically, if a circuit has untestable faults, then there is a cheaper (according to the cost function) implementation, which is more testable. Hence, a minimizer that can find at least a local minimum, will produce a 100% testable circuit.

For example, suppose we are given a 3-Level circuit consisting of gates g_1, g_2 and g_3, where gate g_i is characterized be the equation $y_i = F_i(x, y), \quad i = 1, 2, 3.$ Suppose the primary outputs of the circuit are y_1 and y_2, and the circuit connectivity is implied by the specification

$$\begin{aligned} F_1 &= x_1'x_2' + y_3, \\ F_2 &= x_1x_2' + x_1'x_2, \\ F_3 &= x_1x_2y_2' + x_1'x_2'. \end{aligned}$$

Here x_1 and x_2 are the primary inputs, and x_1' means the complement of the first

[2]We just note in passing that things are more complex in sequential circuits.

primary input. To test for the fault "input y_2 of gate g_3 stuck-at 1", we would (in principle — in practice there are much more clever techniques) simply compare this circuit to a faulty circuit. This faulty circuit is identical to the original except that the connection from gate g_2 to gate g_3 is replaced by a connection to 1 (as if this particular wire had been shorted to the power supply voltage V_{DD}). In the comparison we try to find a particular primary input combination, called a **test** or **test pattern** for which the two circuits have at least one output with different logic values.

Now consider the environment of gate g_3 in the example circuit. In the input space of variables x_1, x_2, and y_2, one can see that since g_2 is an exclusive OR gate, y_2 can't be positive while x_1 and x_2 are both positive or both negative. Consequently, the input minterms $x_1x_2y_2$ and $x_1'x_2'y_2$ are in the satisfiability don't care set $D^{Sat} = x_1x_2y_2 + x_1'x_2'y_2$ for gate g_3 — that is, these input combinations never occur. This has been discussed in Section 3.4. Thus it is easily seen that this fault is untestable, and therefore the literal y_2' in the logic function of gate g_3 is redundant.

The point here is that the process of identifying a redundant literal for optimization purposes is formally identical to that of testing for an input stuck-at fault. In fact we are assured that under certain assumptions about circuit cost (as discussed above), if we synthesize an area-optimal circuit, we shall be guaranteed that it is 100% testable for stuck-at faults.

In the above example it follows that $y_3 \equiv y_2'$, and therefore the following specification, with only two gates and 5 literals, is equivalent to the original, which had 3 gates and 12 literals.

$$\begin{aligned} F_1 &= y_2', \\ F_2 &= x_1x_2' + x_1'x_2, \end{aligned}$$

4.3 Sums of Products and Products of Sums

We now begin examining the minimization of two-level Boolean *formulae.* This is what normally people call "minimizing Boolean functions" and we shall also occasionally say so, since no ambiguity will arise. At this point it should be clear that our objective is to find the simplest **two-level formula** that represents a given function. The formula is related to a circuit that *implements* the given function. Simplicity is measured, as we discussed in Section 4.2.1, with respect to the number of gates and gate inputs of the circuit.

Our first step is to define formally what we mean by two-level formulae. Formulae consist of *constants, variables,* parentheses, and operators, combined according to the recursive definition we have seen. A **letter** is a constant or a variable. A **literal** is a letter or its complement. For instance, for $B = \{0,1\}$ and variables x_1, x_2, $0, 1, x_1, x_2$ are letters and $0, 1, x_1, x_1', x_2, x_2'$ are literals. For simplicity, we give the following definitions for the switching algebra only; this is the case we are most interested in.

A *product term* (or *product,* or simply *term*) is a formula of one of the following forms:

- 1;
- a non-constant literal;
- a conjunction of non-constant literals where no letter appears more than once.

A sum *term* (or **sum**, or **alterm,** or **clause**) is a formula of one of the following forms:

- 0;
- a non-constant literal;
- a disjunction of non-constant literals where no letter appears more than once.

For example, x_1x_2' is a product term, $x_1 + x_2$ is a sum term and x_1' is both. On the other hand, x_1x_1' and x_1x_1 are neither product terms nor sum terms. A **sum of products formula** is one of the following:

- 0;
- a product term;
- a disjunction of product terms.

Likewise, a **product of sums formula** is one of the following:

- 1;
- a sum term;
- a conjunction of sum terms.

For instance,

$$f = x_1x_2' + x_2'x_3 + x_1x_3' \tag{4.1}$$

is a sum of product formula for f. The product of sums dual to (4.1) is $(x_1 + x_2')(x_2' + x_3)(x_1 + x_3')$. Sum of products is abbreviated SOP or $\Sigma\Pi$ and is also called **disjunctive normal form** (DNF). Product of sums is abbreviated POS or $\Pi\Sigma$ and is also called **conjunctive normal form** (CNF).

The *cost* of a SOP formula is determined by the number of product terms and the number of literals. The cost of a POS formula is determined by the number of sum terms and the number of literals. If two SOP formulae have the same number of terms, then the one with fewer literals is cheaper. Likewise for POS formulae. It is also meaningful to compare the cost of a POS formula to the cost of a SOP formula. Indeed, every time we use a technology where the cost of a POS implementation is comparable to the cost of a SOP implementation of the same (abstract) cost, we should derive the best possible POS and the best possible SOP for the function and compare them. The cost of the SOP formula (4.1) is 3 terms and 6 literals. One can verify that the same function can be represented by the POS formula $(x_1 + x_3)(x_2' + x_3')$, whose cost is 2 terms and 4 literals.

A *two-level* formula is either a SOP or a POS. The two forms are one the dual of the other. This is very important, since it allows us to describe all our theorems and algorithms for SOP formulae, without loss of generality. A computer program does not need to know whether a formula is a SOP or a POS in order to find the cheapest equivalent formula of the same kind.

4.4 Implicants and Prime Implicants

An **implicant** of a function f is a product term p that is included in the function f $(p \leq f)$. For instance, both xy' and xyz are implicants of $xy' + yz$.

A **prime implicant** of f is an implicant of f that is not included in any other implicant of f. One can easily see that if p is not prime, then it is possible to obtain another implicant of f by removing one of the literals from p. With reference to the previous example, xy' is prime, whereas xyz is not. Indeed, it is possible to remove y from the latter to get xz, which is a (prime) implicant of the given function. It is also possible to remove x to get yz.

If a prime implicant is an implicant which includes a minterm that is not included in any other prime implicant, then that prime implicant is **essential.** In the previous example, both xy' and yz are essential primes, whereas xz is not.

4.4.1 Quine's Prime Implicant Theorem

The key result for the minimization of two-level formulae is due to Quine [221].

Theorem 4.4.1 *A minimal SOP must always consist of a sum of prime implicants if any definition of cost is used in which the addition of a single literal to any formula increases the cost of the formula.*

The proof of this theorem is fairly simple. One assumes that a minimum-cost formula exists, that contains a non-prime implicant. One then shows that another formula can be obtained by replacing the non-prime implicant by a prime implicant that contains it. The cost does not increase and the formula is equivalent to the original one.

As an example, consider $f = xy' + y$. We know that x is a prime implicant of f and it includes xy'. We can then rewrite f as $x + y$, thereby saving one literal.

The consequence of the Prime Implicant Theorem is that we can focus on only those formulae that are composed of prime implicants. If we want to guarantee the optimality of the solution, we need to choose from all primes. Therefore, as the next step we analyze how to derive all the prime implicants of a given function. Later, we shall see how to select a subset of minimum cost from all the prime implicants.

Efficiency in deriving all primes is important, if we want to handle functions with more than a few inputs. The number of the prime implicants is indeed smaller in general than the number of implicants for a given function, but still grows exponentially with the number of inputs in the worst case.

4.5 Iterated Consensus

Two common methods to generate prime implicants are based on applying the **consensus theorem.** Brown [44] notes that Blake [25] called the consensus of two terms their *syllogistic result.* To understand why, we take a short digression that will be useful in the future.

4.5.1 Consensus and Implications: A Digression

In logic, $x \Rightarrow y$ (read x implies y) is a proposition that is true if y is true whenever x is true. If x is false, then the proposition is true, regardless of the value of y. Therefore

$$x \Rightarrow y = x' + y \tag{4.2}$$

as one may find out by examining all possible cases or just from the previous discussion.

Let us consider the famous **syllogism** "Socrates is a man; all men are mortal; hence Socrates is mortal." Skipping a few formal steps, we can write it as

$$(s \Rightarrow h) \wedge (h \Rightarrow m) \Rightarrow (s \Rightarrow m),$$

where s is the truth value of the proposition "to be Socrates;" similarly for h and m. If we now rewrite it using (4.2), we get

$$(s' + h)(h' + m) \Rightarrow (s' + m).$$

But now we see that the implied term—the conclusion of our syllogism—is actually the consensus term of the two premises.

The important thing to keep in mind from this example is Equation (4.2) that we shall use liberally in the mathematical formulation of problems.

4.5.2 The Tabular Method of Computing the Prime Implicants

We are given an initial SOP formula and we want to find another SOP formula that is the sum of all prime implicants of the function represented by the initial formula.

One way to achieve our goal is to first express the function f in **minterm canonical form.** We then consider all pairs of *adjacent* terms, i.e., the pairs of terms to which consensus can be applied. The consensus terms are clearly implicants of f, though not necessarily prime. All terms that were used to form these new terms are included in the new terms, and hence they are not prime. We mark them as such.

We now take the new terms and repeat the process. We only consider pairs of terms that differ in exactly one letter, which must appear complemented in one term and uncomplemented in the other.

The process is repeated until no more consensus terms can be found. All terms that are absorbed (or contained) by the new terms are marked. Finally, the terms that are not marked constitute all the prime implicants of f.

Calculations by hand are better carried out with the help of a table like the one in Figure 4.3. To compute the complete sum for $f = x'y' + wxy + x'yz' + wy'z$, we initially compute its minterm canonical form:

$$f = w'x'y'z' + w'x'y'z + w'x'yz' + wx'y'z' + wx'y'z + wx'yz' + wxyz' + wxy'z + wxyz.$$

The minterms appearing in the canonical form are entered in the leftmost column. Notice the grouping of the terms that minimizes the number of comparisons. Each group of minterms separated by a horizontal line is composed of minterms with the same number of uncomplemented literals. Hence, the first group consists of the only minterm with no uncomplemented literals. In general, some groups may be empty.

$w'x'y'z'$	✓	$w'x'y'$	✓	$x'y'$
		$w'x'z'$	✓	$x'z'$
		$x'y'z'$	✓	
$w'x'y'z$	✓	$x'y'z$	✓	
$w'x'yz'$	✓	$x'yz'$	✓	
$wx'y'z'$	✓	$wx'y'$	✓	
		$wx'z'$	✓	
$wx'y'z$	✓	$wy'z$		
$wx'yz'$	✓	wyz'		
$wxyz'$	✓	wxy		
$wxy'z$	✓	wxz		
$wxyz$	✓			

Figure 4.3: Tabular Method Applied to $f = x'y' + wxy + x'yz' + wy'z$.

The separation into groups allows one to compare a minterm of a group only to minterms of the immediately successive group. Indeed, these are the only minterms that may be adjacent to it. (We do not need to consider the minterms in the immediately preceding group, because this would only cause us to repeat each comparison.) In our example, for instance, we compare $wxyz'$, from the fourth group, only to $wxyz$. Their consensus term is wxy. Both $wxyz'$ and $wxyz$ are marked: They are not prime, because there exists another implicant (wxy) that contains them.

The results of merging pairs of adjacent minterms are implicants of f with one fewer literal than the minterms; they are entered in the second column. These terms are also divided according to the number of uncomplemented literals and compared to the terms of the next group. The process is then repeated, until no new terms are formed. In our example, there are six terms that are not marked at the end of the process. (They were not used to form any new term.) They are the prime implicants of f:

$$wy'z, wyz', wxy, wxz, x'y', x'z'.$$

Starting from the third column, it is possible to form an implicant in more than one way. For instance, $x'y'$ can be obtained by merging $w'x'y'$ and $wx'y'$; or by merging $x'y'z'$ and $x'y'z$.

If the function is incompletely specified, then we shall mark appropriately the terms that are **don't care,** and drop those terms that are generated only with don't care minterms. Specifically, a product term p is a prime implicant of an incompletely specified function $ff = (f, d, r)$ if $p \leq f + d$, $p \cdot f \neq 0$, and p is not contained in any other implicant of ff. In words, a prime implicant of an incompletely specified function is a prime of $f + d$ that covers at least one element of f.

All prime computation procedures can be extended to handle incompletely specified functions. We examine in detail the extension of Quine's tabular method. Consider the following example:

$$f = yz' + xy'z$$

$x'yz'$	✓	$x'y$	
$x'y'z$	✓ d	yz'	
		$x'z$	d
		$y'z$	
$x'yz$	✓ d		
xyz'	✓		
$xy'z$	✓		

Figure 4.4: Tabular Method Applied to an Incompletely Specified Function.

$$d = x'z$$

In applying Quine's method, we have to keep track of what implicants have been formed by merging **don't care** terms only.

Initially, we mark with a 'd' all the minterms of d. When we merge two implicants, we mark the result with a 'd' only if both the terms that are merged are marked with a 'd.' The result of the procedure is shown in Figure 4.4.

In this example we see that there are three prime implicants. The term $x'z$ is entirely contained in d; hence, it is not a prime implicant.

The tabular method for the generation of prime implicants is due to Quine. Good accounts can be found in [186, 187].

4.5.3 Iterated Consensus in General

The tabular method is based on the application of the theorem

$$Xy + Xy' = X \tag{4.3}$$

This theorem, called **distance-1 merging,** can be seen as a specialized form of consensus, since X is the consensus term of Xy and Xy' and contains both. Because it only uses (4.3), the tabular method is simple and can avoid many comparisons. However, it requires the minterm canonical form to start with. We want to avoid expanding the function into minterms for efficiency. Therefore we look for a different approach, based on the general form of the consensus theorem. We define a **complete sum** as a SOP formula composed of all the prime implicants of the function it represents. We can restate the problem of finding all the prime implicants for f as the problem of finding a complete sum for f. Fortunately, the following result can be proven.

Theorem 4.5.1 *A SOP formula is a complete sum if and only if:*

1. *No term includes any other term.*

2. *The consensus of any two terms of the formula either does not exist or is contained in some term of the formula.*

We shall not prove this result (see [186, p. 168] or [44, Appendix A] for that), but rather suggest why the theorem works.

Suppose a SOP F representing f is given that is not a complete sum, because there is a prime implicant of f that does not appear in F. This implicant must be covered by two or more of the implicants in F. Suppose for simplicity they are two, p_1 and p_2. If we add the consensus term of these two implicants, we add one term that spans the border of p_1 and p_2 and therefore may cover the missing prime implicant. (It will actually cover it, if p_1 and p_2 are prime.)

The theorem and the discussion following it suggest a simple procedure to generate all the primes of a function, called *iterated consensus.* One starts from an arbitrary SOP formula and adds the consensus terms of all pairs of terms that are not contained in some other term. The new terms are compared to the existing terms and among themselves to see if new consensus terms can be generated. All terms that are contained in some other term are removed. When no further changes are possible, a complete sum is generated.

As in the tabular method, a clever organization of comparisons can substantially reduce the amount of work. In particular it is convenient to compare every term only to the terms that precede it in the formula. This prevents duplicate operations and also takes care naturally of the addition of new terms.

As an example, consider the computation of the complete sum starting from the following SOP formula:

$$x_1x_2 + x_2'x_3 + x_2x_3x_4.$$

We begin by comparing the second term to the first. We append the consensus term (x_1x_3) to the formula, obtaining:

$$x_1x_2 + x_2'x_3 + x_2x_3x_4 + x_1x_3.$$

We then compare the third term $(x_2x_3x_4)$ to the first and second terms. The latter gives a consensus term (x_3x_4) that we append to the formula:

$$x_1x_2 + x_2'x_3 + x_2x_3x_4 + x_1x_3 + x_3x_4.$$

Nothing happens when we compare the fourth term (x_1x_3) to those that precede it. Finally, when we compare the last term to $x_2x_3x_4$, we remove the latter, because it is included in the former. This terminates our computation. The resulting complete sum is:

$$x_1x_2 + x_2'x_3 + x_1x_3 + x_3x_4.$$

4.6 Recursive Computation of Prime Implicants

Another property of complete sums is given by the following theorem.

Theorem 4.6.1 *The SOP obtained from two complete sums F_1 and F_2 by the following procedure is a complete sum for $F_1 \cdot F_2$.*

1. *Multiply out F_1 and F_2 using the idempotent and distributive properties and $x \cdot x' = 0$.*

2. *Eliminate all terms that are contained in some other term.*

For the proof we refer to [44, Appendix A]. The result generalizes to the product of n complete sums. Since a sum term is a simple case of complete sum, if we start from a POS formula and apply the procedure of Theorem 4.6.1, we get a complete sum.

As an example, let us consider the following POS formula:

$$(x_1 + x_2)(x_2' + x_3)(x_3 + x_4).$$

After multiplying out the first two sum terms, we get:

$$(x_1x_2' + x_1x_3 + x_2x_3)(x_3 + x_4).$$

No term of the first sum is contained in any other term, so we continue:

$$x_1x_2'x_3 + x_1x_2'x_4 + x_1x_3 + x_1x_3x_4 + x_2x_3 + x_2x_3x_4.$$

There are now some cases of containment. Once all the contained terms are eliminated we get the complete sum:

$$x_1x_2'x_4 + x_1x_3 + x_2x_3.$$

If we are not given a POS formula, we can use the second form of Boole's expansion theorem to write f as the product of two other functions:

$$f(x_1, x_2, \ldots, x_n) = [x_1' + f(1, x_2, \ldots, x_n)] \cdot [x_1 + f(0, x_2, \ldots, x_n)].$$

This leads naturally to a recursive method where we apply the same procedure to each of $f(1, x_2, \ldots, x_n)$ and $f(0, x_2, \ldots, x_n)$ until they simplify to the point that we can derive all the prime implicants by inspection.

The simplification is guaranteed to take place, since at every step of the recursion the number of variables decreases. In the end, we shall find formulae of only one term, for which it is trivial to compute the complete sum (it is the term itself). When we return from the recursion, we combine the results of the two sub-problems using Theorem 4.6.1. We can indicate the complete sum of f by $CS(f)$ and write:

$$CS(f) = ABS([x_1 + CS(f(0, x_2, \ldots, x_n))] \cdot [x_1' + CS(f(1, x_2, \ldots, x_n))]),$$

where $ABS(f)$ returns the formula obtained by removing absorbed terms from f. For instance, $ABS(x + y + xz) = x + y$.

We shall see many recursive algorithms like this in the sequel. This one in particular is the algorithm used in ESPRESSO [37] to compute all the primes of a function, when the exact minimum-cost solution is requested. It is a good thing to spend some time to familiarize with the operation of this algorithm, especially by visualizing the **recursion tree.**

As an example, we now compute the complete sum for

$$f = v'xyz + v'w'x + v'x'z' + v'wxz + w'yz' + vw'z + vwx'z$$

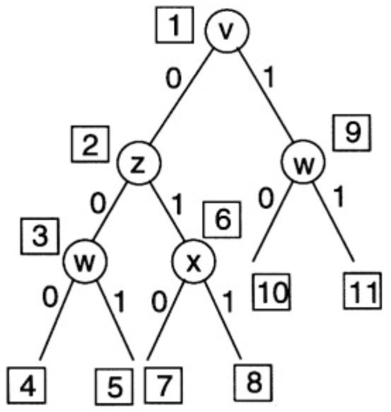

Figure 4.5: Example of Recursion Tree for the Computation of Prime Implicants.

$\boxed{1}$ $f(v,w,x,y,z) = v'xyz + v'w'x + v'x'z' + v'wxz + w'yz' + vw'z + vwx'z$

$\boxed{2}$ $f(0,w,x,y,z) = xyz + w'x + x'z' + wxz + w'yz'$

$\boxed{3}$ $f(0,w,x,y,0) = w'x + x' + w'y$

$\boxed{4}$ $f(0,0,x,y,0) = x + x' + y = 1$

$\boxed{5}$ $f(0,1,x,y,0) = x'$

$\boxed{3}$ $CS(f(0,w,x,y,0)) = ABS((w+1)(w'+x')) = w' + x'$

$\boxed{6}$ $f(0,w,x,y,1) = xy + w'x + wx$

$\boxed{7}$ $f(0,w,0,y,1) = 0$

$\boxed{8}$ $f(0,w,1,y,1) = y + w' + w = 1$

$\boxed{6}$ $CS(f(0,w,x,y,1)) = ABS((x+0)(x'+1)) = x$

$\boxed{2}$ $CS(f(0,w,x,y,z)) = ABS((z + w' + x')(z' + x))$
$= w'x + w'z' + x'z' + xz$

$\boxed{9}$ $f(1,w,x,y,z) = w'yz' + w'z + wx'z$

$\boxed{10}$ $f(1,0,x,y,z) = yz' + z = y + z$

$\boxed{11}$ $f(1,1,x,y,z) = x'z$

$\boxed{9}$ $CS(f(1,w,x,y,z)) = ABS((w + y + z)(w' + x'z))$
$= ABS(wx'z + w'y + x'yz + w'z + x'z)$
$= w'y + w'z + x'z$

$\boxed{1}$ $CS(f(v,w,x,y,z)) = ABS((v + w'x + w'z' + x'z' + xz)(v' + w'y + w'z + x'z))$
$= vw'y + vw'z + vx'z + v'w'x + w'xy + w'xz +$
$v'w'z' + w'yz' + v'x'z' + v'xz$

The recursion tree is given in Figure 4.5. The line numbers in the example correspond to the node numbers in the tree. When we apply Boole's expansion, it is important to choose a good *splitting variable*. The objective is to minimize the amount of computation, for instance, by minimizing the number of nodes in the tree. One way to heuristically achieve that is to minimize the number of terms that appear in both $f(1, x_2, \ldots, x_n)$ and $f(0, x_2, \ldots, x_n)$. This suggests the choice of the variable that appears in the largest number of terms. Indeed those terms where the selected variable does not appear, will become part of both sub-problems. By minimizing the size of the sub-problems, we increase the chance of early termination of the recursion.

Returning to our example, we chose v as initial splitting variable, because it was one of the best (the other being z) according to that criterion. We shall return to this subject in more depth when we deal with heuristic minimization in Chapter 5.

For an incompletely specified function, we need to expand the procedure we have seen by actually computing the prime implicants of $f + d$. By so doing, we may erroneously include in the list of prime implicants some terms that only cover don't care minterms. However, all the true prime implicants will be included. We shall see in Problem 13 that possibly including some terms that do not cover any 'care' minterm is not a mistake. In other words, it does not affect the correctness of the procedure.

4.7 Selecting a Subset of Primes

Recall that Quine proved that a minimum cost SOP formula can be obtained by considering prime implicants only. Since we know how to generate all the primes of a function, we now turn our attention to the selection of a subset of implicants of minimum cost. The approach to minimizing a SOP or POS formula based on computing all primes and then selecting some of them to form a cover goes under the name of the **Quine-McCluskey** procedure. As an example we consider the following formula:

$$f(x, y, z) = yz + x'y + y'z' + xyz + x'z'$$

The complete sum for f is

$$x'y + x'z' + y'z' + yz.$$

The condition that any subset of primes must satisfy to represent a valid formula of the function is that each minterm for which the function is 1 (each minterm of the function for short) be included in at least one implicant of the subset.

A subset of implicants that satisfies this requirement is called a *SOP cover* of the function, or simply a cover, when the context prevents ambiguity. We can build a **constraint matrix** that describes the conditions or constraints that a cover must satisfy. Each column of the constraint matrix corresponds to a prime implicant and each row corresponds to a minterm. Let A be the constraint matrix and let a_{ij} be the element in row i and column j. Then, $a_{ij} = 1$ if the j-th prime covers the i-th minterm. Otherwise, $a_{ij} = 0$. In our example, let p_1 stand for $x'y$, p_2 stand for $x'z'$, p_3 for $y'z'$, and p_4 for yz. A is given by:

$$\begin{array}{c|cccc} & p_1 & p_2 & p_3 & p_4 \\ \hline x'y'z' & 0 & 1 & 1 & 0 \\ x'yz' & 1 & 1 & 0 & 0 \\ x'yz & 1 & 0 & 0 & 1 \\ xyz & 0 & 0 & 0 & 1 \\ xy'z' & 0 & 0 & 1 & 0 \end{array} \qquad (4.4)$$

Given the constraint matrix, our problem is to find a subset of columns of minimum cost that *covers* all the rows. In other words, for every row there must be at least one selected column with a 1 in that row.

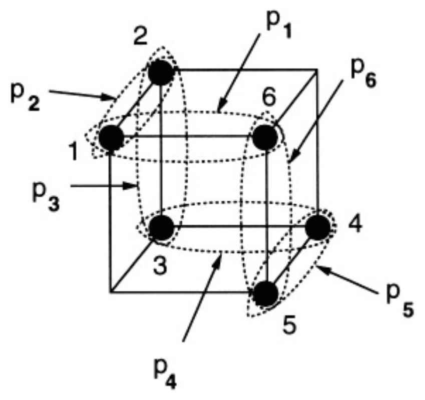

Figure 4.6: A Function with a Cyclic Core.

Before we proceed with the formal statement of our problem, we note that in our example, columns p_3 and p_4 must be part of every solution, because the last two rows are *singletons*. If a row is a singleton, there is only one column that may cover it and that column must be selected.[3] When we select some columns, we simplify the constraint matrix accordingly, by eliminating the selected columns and the rows covered by them:

$$\begin{array}{c c}
 & \begin{array}{cc} p_1 & p_2 \end{array} \\
x'yz' & \left[\begin{array}{cc} 1 & 1 \end{array}\right]
\end{array}$$

Once this is done, we can easily see that a complete solution may be obtained by adding either p_1 or p_2 to p_3 and p_4. In the first case we obtain $x'y + y'z' + yz$; in the second, we obtain $x'z' + y'z' + yz$. These are two sums of products of minimum cost for f.

Unfortunately, we cannot always find the solution directly, as in the example of Figure 4.6, whose constraint matrix is:

$$\begin{array}{c c}
 & \begin{array}{cccccc} p_1 & p_2 & p_3 & p_4 & p_5 & p_6 \end{array} \\
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \end{array} &
\left[\begin{array}{cccccc}
1 & 1 & & & & \\
 & 1 & 1 & & & \\
 & & 1 & 1 & & \\
 & & & 1 & 1 & \\
 & & & & 1 & 1 \\
1 & & & & & 1
\end{array}\right]
\end{array}$$

This function is said to have a *cyclic core* (which in this case is the function itself). We shall return to the definition of cyclic core later; for the time being, it is sufficient to say that a function has a cyclic core if we cannot identify columns that must be part of the solution or that can be eliminated.

In our example, each row is covered by exactly two columns and each column covers exactly two rows. There is no apparent reason to prefer one column over another. For this matrix we must proceed by choosing one column arbitrarily and

[3]Columns p_3 and p_4 correspond—as we have seen—to essential primes, that is, primes that cover a minterm not covered by any other prime.

finding the best solution subject to the assumption that the column is selected. We must then assume that the column is not in the solution and find another solution. We then compare the two solutions obtained and keep the best. For instance, we may select p_1 and find a solution including it. Then we may find a solution not including p_1. Finally, we choose the better of the two solutions.

We have seen, in the two previous examples, two important mechanisms in action: Reduction of the constraint matrix and branching in the case of cyclic cores. We shall develop these mechanisms in detail in the sequel. Before that, however, we want to mention another possible formulation of the problem that is important to us.

One may readily see that the first row of the constraint matrix in our first example can be written as the switching function

$$(p_2 + p_3).$$

This function evaluates to one if either $p_2 = 1$ or $p_3 = 1$ or both. If we interpret $p_i = 1$ as "column p_i is selected," we see that the switching function is 1 when the first row of the matrix is covered and vice versa. We can proceed similarly for the other rows. The expressions thus obtained are switching functions that must all be 1 for a solution to be valid. Hence, their product must be 1. We can therefore write the following equation as an equivalent to the constraint matrix:

$$(p_2 + p_3)(p_1 + p_2)(p_1 + p_4)p_3p_4 = 1. \tag{4.5}$$

This equation is called the *constraint equation* of the covering problem. The covering problem can be formulated in this setting as the problem of finding an assignment of zeroes and ones to the variables that is a solution to the constraint equation and that is of minimum cost.

The equation can be simplified, for instance, to

$$(p_1 + p_2)p_3p_4 = 1,$$

using the absorption property. This simplification has a similar effect to detecting the essential columns and illustrates the general fact that we shall find a Boolean operation on the constraint equation corresponding to each operation on the constraint matrix.

We conclude this introduction by noting that all variables in the constraint equation appear uncomplemented. This is not a coincidence, but rather a direct consequence of the way the equation is built. A formula where no letter appears with both phases is said **unate.** A non-unate formula is called **binate.** Because of the form of the constraint equation that we get, the covering problem we are dealing with is called sometimes **unate covering.** The problem that is obtained by relaxing the assumption that the constraint equation is unate is called **binate covering covering.**

We shall return to unate functions and formulae and to binate covering in the future. Now, however, we concentrate on the efficient solution of the unate covering problem.

4.8 The Unate Covering Problem

Here we define UCP, the **Unate Covering Problem,** giving both a constraint matrix and a constraint equation form of the problem. Our statement of the problem will be general, in the sense that the problem will not require the context of minterms covered by prime implicants, as in the previous section.

We then describe some fundamental methods that can be used to organize and greatly simplify the solution of UCP. We first detail the reduction techniques that can be applied to simplify the constraint matrix and/or the constraint equation. We then describe how the solution of UCP can be viewed as the process of enumerating all the solutions to the constraint equation and picking out one of the solutions of minimum cost. We conclude this section with a method for determining an **a priori** lower bound on the cost of the solution. In the succeeding section, we will organize all of these ideas into an efficient algorithm for solving UCP in the general case.

Definition 4.8.1 (UCP, constraint equation form) *Let $J = \{1, \ldots, n\}$, and let $p = (p_1, \ldots, p_n)$ be a vector of n Boolean variables. Let $\sigma_i = \sum_{j \in J_i} p_j$, $i = 1, \ldots, m$, where $J_i \subseteq J$ is a set of m simple sums of single, positive literals of the variables p_j.*

Then UCP is the problem of finding a minimum cardinality subset $S \subseteq J$, setting $p_j = 1$, $\forall p_j \in S$, guarantees that

$$\prod_i \sigma_i(p) = 1.$$

That is, for any other subset S' having the above properties, we have $|S| \leq |S'|$. Note each of the sums $\sigma_i(p) : \{0,1\}^n \longmapsto \{0,1\}$ is a Boolean function.

In the constraint equation of Equation 4.5 we have $\sigma_1 = (p_2+p_3)$, $\sigma_2 = (p_1+p_2)$, $\sigma_3 = (p_1 + p_4)$, $\sigma_4 = p_3$, and $\sigma_5 = p_4$. For this problem, we showed above that $\prod_i \sigma_i(p) = (p_1 + p_2)p_3p_4$.

Definition 4.8.2 (UCP, constraint matrix form) *Let M be a matrix of m rows and n columns, for which M_{ij} is either* 0 or 1. *Then UCP is the problem of finding a minimum cardinality column subset S, such that for all S' such that*

$$\exists_{j \in S'} M_{ij} = 1, \ \forall i \in \{1, \ldots, n\} \Rightarrow |S| \leq |S'|.$$

That is, the columns in the set S "cover" M in the sense, that every row of M contains a 1*-entry in at least one of the columns of S, and there is no smaller set S' which also covers M.*

For the constraint matrix of Equation 4.4 we have two possible solutions to the matrix form of UCP: $S^1 = \{1,3,4\}$, and $S^2 = \{2,3,4\}$.

The two forms of stating UCP are totally equivalent, and applicable in a context which is much broader than the minimization of logic functions. We demonstrate this with the following example. After the example, we shall take a closer look at how the efficiency of the solution is affected by the key mechanisms of reduction, enumeration, and lower bounding.

Example: Recent studies have indicated that a good diet should contain adequate amounts of proteins (P), vitamins (V), fats (F), and cookies (C). An astronaut has to choose from a menu of five different preparations with the following nutritional information labels.

- Preparation 1 contains: V and P;
- Preparation 2 contains: V and F;
- Preparation 3 contains: P and F;
- Preparation 4 contains: V;
- Preparation 5 contains: C.

(Preparation 5 is actually a ration of peanut butter cookies; the rest is yucky stuff.) Can the astronaut have a balanced diet with only two preparations?

In the constraint equation form of this instance of UCP, $n = 5$, and the 5 preparations are represented by the variables p_i, $i = 1, \ldots, 5$. Similarly, there are 4 constraint sums σ_i, corresponding to the requirement that each nutrient P, V, F, and C be included in the astronauts' diet. Thus, $\sigma_1 = (p_1 + p_3)$ represents the protein requirement, $\sigma_2 = (p_1 + p_2 + p_4)$, represents the vitamin requirement, $\sigma_3 = (p_2 + p_3)$, represents the fats requirement, and $\sigma_4 = (p_5)$, represents the cookie requirement. Thus the constraint equation is

$$\begin{array}{rcl} \prod_i \sigma_i(p) & = & (p_1 + p_3)(p_1 + p_2 + p_4)(p_2 + p_3)(p_5) \\ & = & (p_1 + p_3(p_2 + p_4))(p_2 + p3)p_5 \\ & = & (p_1)(p_2 + p3)p_5 + p_3(p_2 + p_4)(p_2 + p3)p_5 \\ & = & (p_1)(p_2 + p3)p_5 + p_3(p_2 + p_4)p_5 \\ & = & 1. \end{array} \tag{4.6}$$

Thus we see that there are exactly 4 solutions of size 3, but none of size 2. Further, every solution to the constraint equation requires the assignment $p_5 = 1$. We thus say that p_5 is "essential" in the sense that it is the only preparation providing the essential nutrient cookies (we define this formally below).

The corresponding covering matrix is

$$\begin{array}{c} \\ P \\ V \\ F \\ C \end{array} \begin{array}{c} \begin{array}{ccccc} 1 & 2 & 3 & 4 & 5 \end{array} \\ \left[\begin{array}{ccccc} 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{array}\right] \end{array}$$

In this representation, the essential nature of p_5 is manifested as a singleton row, meaning that only one column, p_5 can cover the cookie row C. Consequently, all minimum covers contain column 5 (p_5). Also, Preparation 4 is dominated by preparation 2. The remaining matrix is cyclic (see Figure 4.6), and, as we shall prove in Theorem 4.8.1 on Page 150, will therefore require at least two columns. Thus there are 3 solutions, which are $p_5(p_1p_2 + p_2p_3 + p_3p_1)$. Note that the solution $p_3p_4p_5$ is missing (because column 4 was eliminated by dominance). However, we are guaranteed to retain an optimum solution (of size 3). Hence, it is impossible to have a balanced diet with only two preparations. ■

4.8.1 Reduction Techniques

We consider three reduction techniques (to be defined below):

1. elimination of rows covered by "essential columns";
2. elimination of rows through "row dominance";
3. elimination of columns through "column dominance".

For each of them, we present two forms: One applicable to a constraint matrix and one applicable to a constraint equation. We also show that the two forms are equivalent.

In general, we want to iterate the three forms of reduction in a fixed order, reiterating as long as the matrix keeps simplifying. One common order is the one that we follow in our presentation: first, check for essential columns; second, check for row dominance; third, check for column dominance. One type of reduction often leads to another, but eventually the matrix reduces to a case in which no further reduction is possible. The iteration stops when such a case is reached.

4.8.2 Essential Columns or Variables

If a row of the constraint matrix is a singleton, the corresponding column must be part of the solution. The essential columns and all rows covered by them are removed from the constraint matrix. We saw an example of this process in Section 4.7.

The analogous process for a constraint equation consists of identifying the terms of the POS formula that consist of one literal only. The corresponding variables must be set to 1 and the equation simplified accordingly. For the following constraint equation,

$$(p_2 + p_3)(p_1 + p_2)(p_1 + p_4)p_3 p_4 = 1,$$

we have to set $p_3 = p_4 = 1$. The resulting simplified equation is:

$$(p_1 + p_2) = 1.$$

In the following two sections, we show how the subproblems based on the positive and negative cofactors of the constraint POS may have essential variables, even in cases for which the original POS for F had none.

4.8.3 Row or Constraint Dominance

If a row r_i of the constraint matrix has all the ones of another row r_j, then r_i is covered whenever r_j is covered. Therefore, we do not need r_i in our matrix, because the constraint it represents is superfluous. We say that r_i dominates r_j. All dominating rows can be eliminated from the constraint matrix. For the following matrix:

$$\begin{array}{c|cccccc|}
 & 1 & 2 & 3 & 4 & 5 & 6 \\
1 & 1 & 1 & & & & 1 \\
2 & 1 & 1 & & & & \\
3 & & 1 & 1 & & & \\
4 & & 1 & 1 & 1 & & \\
5 & & & & 1 & 1 & \\
6 & & & & 1 & 1 & 1
\end{array}$$

the first, fourth, and sixth rows dominate the second, third, and fifth rows, respectively and can be eliminated. The reduced matrix is:

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 2 & 1 & 1 & & & \\ 3 & & 1 & 1 & & \\ 5 & & & & 1 & 1 \end{array}$$

The corresponding reduction for the constraint equation is based on the application of the absorption property, i.e., $x(x+y) = x$. Let us consider the equation corresponding to the previous example:

$$(p_1 + p_2 + p_6)(p_1 + p_2)(p_2 + p_3)(p_2 + p_3 + p_4)(p_4 + p_5)(p_4 + p_5 + p_6) = 1.$$

By absorption, we can replace, for instance, $(p_1 + p_2 + p_6)(p_1 + p_2)$ by $(p_1 + p_2)$. The reduced equation is:

$$(p_1 + p_2)(p_2 + p_3)(p_4 + p_5) = 1.$$

We see that this equation corresponds to the reduced matrix of the previous example.

4.8.4 Column or Variable Dominance

Before we discuss this technique, we must briefly digress to consider the cost of a column or a variable. In general, each prime corresponds to one AND gate in a SOP circuit. If the number of gates is the only concern, it is correct to assign the same cost to all columns or variables. However, if the number of literals is more important, then a prime with, say, five literals, should be considered more expensive than a prime with three literals. This can be accommodated by assigning different costs to the columns or variables. The total cost of a solution is then the sum of the costs of the selected columns or, in other words, the cost of the variables set to 1.

Suppose now that a column p_i has all the ones of another column p_j. Suppose further that the cost of p_i is not greater than the cost of p_j. Then, we can say that p_i is not inferior to p_j, in that it covers all the rows that p_j covers, at a cost that is not larger. This means that we can discard p_j from the matrix, without giving up the possibility of finding an optimum solution.

As an example, let us consider the matrix obtained in the previous example by row dominance. Suppose all columns have the same cost. We see that the second column dominates the first and the third. According to our definition, the fourth column dominates the fifth and vice versa. In this case we can choose arbitrarily which column to retain. Say we choose the fourth. The result of the reduction is the following matrix:

$$\begin{array}{c|cc} & 2 & 4 \\ \hline 2 & 1 & \\ 3 & 1 & \\ 5 & & 1 \end{array}$$

Here we see how row and column dominance reductions can engender new essential columns/variables. Note when we employ column dominance, we choose to ignore some valid solutions to the constraint equation, some of which may be optimum.

However, we may do this with the assurance that we always retain at least one optimum solution.

If we are given the constraint equation, variable dominance can be checked as follows. We say that p_i dominates p_j if the cost of p_i does not exceed the cost of p_j and p_i appears in every term where p_j appears. This criterion is a simple translation of the one employed for the constraint matrix. A more general formulation is possible, but we shall not pursue it for the time being. Returning to our example, consider the constraint equation

$$(p_1 + p_2)(p_2 + p_3)(p_4 + p_5) = 1.$$

We see that p_2 appears in all terms where p_1 appears. Hence, p_2 dominates p_1. Similarly, p_3 is dominated by p_2 and p_4 and p_5 dominate each other. (We choose p_4 arbitrarily.) The reduction consists of taking the negative cofactor of the left-hand side of the equation with respect to all dominated variables (the order is unimportant). The reduced equation is thus:

$$p_2 p_2 p_4 = 1,$$

which becomes $p_2 p_4 = 1$ by idempotency.

4.8.5 Systematically Exploring the Search Space

When the constraint matrix cannot be further reduced, we have two cases: If the matrix has no rows left, then we say that we have reached a terminal case and we have solved the problem; otherwise the problem is cyclic. If we are working with the constraint equation, the terminal case occurs when the constraint equation simplifies to $1 = 1$.

If the reduced constraint equation, written $F = 1$, has no essential variables, then, since each p_i is a binary variable, we can seek the minimum cover by a divide and conquer strategy: first, consider all solutions for which $p_i = 1$, and then consider all solutions for which $p_i = 0$. The constraint equation which corresponds to the former case is obtained by setting $p_i = 1$, and is written

$$F_{p_i} = 1.$$

Similarly, in the latter case we set $p_i = 0$ and obtain $F_{p_i'} = 1$. The two formulae F_{p_i} and $F_{p_i'}$ are the *positive* and *negative cofactors* of F with respect to p_i. Cofactors were defined in Section 3.3.3 (Page 98).

It needs to be emphasized that a zero in the constraint matrix does not correspond to a negated literal in the constraint equation. This is because we are restricting attention in this chapter to UCP, the Unate Covering Problem, in which every variable appears in just one phase. Consequently, when we take a negative cofactor of the constraint equation with respect to p_i, the corresponding action on the constraint matrix is to simply delete the column. The rows of the constraint matrix which have 0s in column p_i are *not deleted,* because the 0s mean that variable p_i did not appear in the corresponding sum of the POS form of the constraint equation.

If we want to solve large enumeration problems, we have to avoid enumerating all solutions explicitly, since their number is exponential in the variables of the problem. What we want is an **implicit enumeration** of the solutions, where many (as many as possible) solutions are not explicitly considered. The reduction techniques of the previous section are an important part of the implicit enumeration scheme for the covering problem; they allow us to determine that some variables (prime implicants in the logic minimization context) are either essential or can be left out of the solution without impairing our ability to find an optimum solution.

We have seen in Section 4.7 that when the problem is cyclic we need to select a column (or a variable) and solve two reduced subproblems: One subproblem is obtained by accepting the selecting column; the other subproblem is obtained by rejecting it. The reduced subproblems may in turn be cyclic and therefore require the selection of new splitting variables. This gives rise to a recursive process of selecting variables and tentatively solving subproblems in order to find which one yields the best solution. This process can be seen as the exploration of the *search space* of the problem. In a cyclic covering problem, we must, in principle, enumerate all possible solutions, in order to find one of minimum cost. Therefore we say that the covering problem is an enumeration problem.

In the nutrient problem discussed above on Page 144 we enumerated all the solutions to the constraint equation by transforming from POS form to SOP form. This can be called "explicit enumeration" since all solutions were examined. All possible solutions can also be obtained by recursively cofactoring. However when we use the reduction techniques of the previous section, we eliminate some solutions, including optimum solutions, from consideration. This is called "implicit enumeration", because we are enumerating some, but not all, of the possible solutions.

We now consider how to systematically explore the search space in the case of cyclic problems, so as to minimize the number of solutions that we explicitly enumerate. The implicit enumeration scheme we adopt is called **Branch and Bound,** and is a general scheme for solving enumeration problems. Branch and Bound applies to search problems where we are interested in finding the minimum or maximum cost of a feasible solution. There are many problems for which clever specialized search strategies can be applied. For instance, we do not need to enumerate all possible paths between two cities to find the shortest one. Hence we do not apply a branch-and-bound technique to solving the shortest path problem. However, no such clever search strategy is known for the covering problem (and it is unlikely to exist). Hence, we have to enumerate and we are interested in minimizing the work. In this context, Branch and Bound can be (extremely) useful.

4.8.6 Computation of the Lower Bound

We now address the problem of computing a lower bound approximation to the cost of covering a constraint matrix. The method easily translates into corresponding problem of approximating the cost of satisfying a constraint equation.

In a given covering matrix *M*, suppose that two rows r_i and r_j have nonzero entries in sets $R_i = \{k \mid M_{ik} \neq 0\}$ and $R_j = \{k \mid M_{jk} \neq 0\}$. If $R_i \cap R_j = \emptyset$, that is, if r_i and r_j have no nonzero columns in common, we say these two rows are independent (that is, column-disjoint). It is apparent that we need two different columns to cover these two rows. Generalizing this argument, if a matrix has n rows that are similarly disjoint (pairwise), we need at least n columns to cover the whole matrix. In this case, the n rows are said to form an **independent** set of rows. For each row of the independent set, the cost of covering it is at least the cost of the cheapest column that covers the row. Hence, by summing the costs of the cheapest columns covering each of the rows in the independent set, we get a lower bound to the cost of covering the entire matrix.

As an example, let us consider the following matrix:

$$\begin{array}{c|cccccc} & 1 & 2 & 3 & 4 & 5 & 6 \\ \hline 1 & 1 & & & & & 1 \\ 2 & 1 & 1 & & & & \\ 3 & & 1 & 1 & & & \\ 4 & & & 1 & 1 & & \\ 5 & & & & 1 & 1 & \\ 6 & & & & & 1 & 1 \end{array} \tag{4.7}$$

The first, third, and fifth rows are independent. Hence, we need at least three columns to cover the matrix. There is another independent set of three rows, namely the second, fourth, and sixth rows. Notice that in this case the lower bound is exact: we can cover the matrix with exactly three columns. In general, however, there will be independent sets of different sizes and the lower bound will not necessarily be exact. It is also easy to see that there is always at least an independent set of size 1. Further, for cyclic (irreducible) matrices with unit cost columns, we always get a lower bound of size 2 or more, as shown in the following theorem.

Theorem 4.8.1 *In the unate covering problem with unit costs for the columns, the lower bound for a cyclic matrix is at least 2 (even if there are no two independent rows).*

Proof. Suppose a constraint matrix of a unate covering problem contains a full column of ones. In this case, the matrix cannot be cyclic because the column with all ones dominates all the others. The matrix can therefore be reduced to one column, which is obviously essential. Thus such a matrix is not cyclic. It then follows that if the matrix is cyclic, there can be no column that covers all the rows. Hence, at least two columns are required to cover all the rows. □

An independent set of rows is called **maximal** if it intersects (that is, has a column in common with) every other row of the covering matrix. We shall use the abbreviation MIS for a Maximal Independent Set. This means that unless some of the decisions already made in building the set are reversed, the set cannot grow larger while retaining its independence (pairwise disjointness).

A simple algorithm for quickly finding an MIS is Procedure MIS_QUICK of Figure 4.7. Here $\| M \|$ denotes the number of rows left in M after deleting the rows intersecting the chosen row.

The key feature of this algorithm is Subprocedure CHOOSE_SHORTEST_ROW. In its simplest form it just chooses the "shortest" row, that is, the row with the fewest nonzero columns, and breaking ties in ascending lexicographical order. Better heuristic performance is usually obtained by a more sophisticated heuristic, in which the weight of row i is defined in terms of the column counts of its columns. That is, let

```
MIS_QUICK($M$) {
    $MIS = \emptyset$
    do {
        $i$ = CHOOSE_SHORTEST_ROW($M$)
        $MIS = MIS \cup \{i\}$
        $M$ = DELETE_INTERSECTING ROWS($M, i$)
    } while ($\| M \| > 0$) continue
    return ($MIS$)
}
```

Figure 4.7: Algorithm for computing an MIS.

$C_j = \{k \mid M_{kj} \neq 0\}$ be the set of nonzero row in column j. Then the weight of row i is $\sum_{j \in R_i} |C_j|$, and CHOOSE_SHORTEST_ROW chooses the row of minimum weight, breaking ties by choosing shortest rows in ascending lexicographical order.

We first apply CHOOSE_SHORTEST_ROW to the covering matrix of Equation 4.7. Here all rows have equal weight by either heuristic, and the MIS chosen would be $\{1, 3, 5\}$, which is optimum, since $|MIS| = 3$ is the cost of the optimum solution of the covering problem. In this case the bound is sharp.

However, now let us apply this algorithm to the following example, which discriminates between the two heuristics.

$$M = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 \end{matrix} \\ \left[\begin{matrix} 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 \end{matrix}\right] \end{array} \begin{array}{l} \\ 1 \quad w_1^1 = 2 \quad w_1^2 = 5 \\ 2 \quad w_2^1 = 3 \quad w_2^2 = 9 \\ 3 \quad w_3^1 = 2 \quad w_3^2 = 6 \\ 4 \quad w_4^1 = 3 \quad w_4^2 = 7 \\ 5 \quad w_5^1 = 2 \quad w_5^2 = 6 \\ 6 \quad w_6^1 = 2 \quad w_6^2 = 5 \\ 7 \quad w_7^1 = 2 \quad w_7^2 = 6 \end{array}$$

The superscripts 1 and 2 indicates row weights computed according to the first and second heuristics. Thus for the first heuristic, MIS_QUICK would obtain $MIS = \{1\}$ on the first pass, and $MIS = \{1, 3\}$ on the second (and last) pass.

However, for the second heuristic, MIS_QUICK would obtain $MIS = \{1\}$ on the first pass, after which the reduced matrix is

$$M = \begin{array}{c} \begin{matrix} 1 & 2 & 3 & 4 & 5 & 6 \end{matrix} \\ \left[\begin{matrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 \end{matrix}\right] \end{array} \begin{array}{l} \\ 3 \quad w_3^1 = 2 \quad w_3^2 = 4 \\ 5 \quad w_5^1 = 2 \quad w_5^2 = 3 \\ 6 \quad w_6^1 = 2 \quad w_6^2 = 3 \end{array}$$

The second heuristic would give $MIS = \{1, 5\}$ on the second pass, and $MIS = \{1, 5, 6\}$ on the third (and last) pass. The improved heuristic was able to identify a

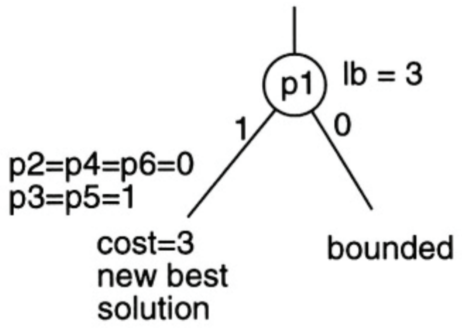

Figure 4.8: Recursion Tree for a Covering Problem.

larger MIS by paying attention to how many rows are eliminated by each choice of a row to be added to the the current *MIS*.

Returning to our example, we found that it was a cyclic problem. If we select Column 1 for splitting, we get the solution {1, 3, 5}. Since the cost of this solution (3) equals the lower bound, we know it is optimal and we do not need to find the best solution without Column 1. This process is illustrated in Figure 4.8, where the recursion tree is shown.

4.9 The Branch-and-Bound Algorithm

In this section we formulate an efficient procedure to solve the unate covering problem that we have informally introduced in the previous section. The procedure is essentially the one presented in [239], which in turn is based on the earlier procedure due to McCluskey [188]. We then consider cyclic problems and we present a branch-and-bound algorithm for them. Finally, we mention the connection between unate covering and integer linear programming.

The idea behind Branch and Bound is that we are only interested in finding one optimum solution (there may be many). Therefore, if we can determine that a given part of the search space does not contain any solution better than the best we have found so far, then we can avoid exploring that part of the search space altogether.

How do we come to the conclusion that there are no 'interesting' solutions in a part of the search space? In Branch and Bound, we resort to two basic ideas. The first is that the search space is organized in the form of a **search tree**(sometimes called a recursion tree). To fix ideas, we consider the case of a **binary** search tree, which is what we are going to use. Each node of the tree corresponds to a variable of the problem and the two branches out of the node correspond to the acceptance or rejection of the variable. The 'Branch' in Branch and Bound refers to the process of exploring the branches of the search tree. An example is shown in Figure 4.9. Let us examine the leftmost path in the tree. We assume that at the top node (the root) we have selected p_1 as the splitting variable. We also suppose that in the simplified subproblem we can identify that p_2 must be 1 (e.g., it is an essential variable for the subproblem) and p_6 can be set to 0 (e.g., it is a dominated variable). The resulting hypothetical simplified subproblem is still cyclic. Hence, we now select another splitting variable, p_4, whose choice allows us to set also $p_7 = 0$ (e.g., another

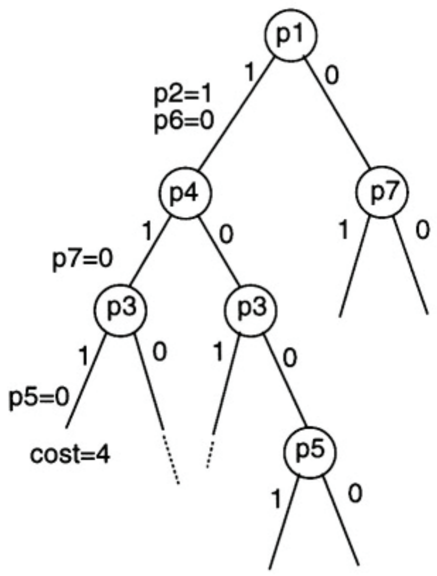

Figure 4.9: Example of Search Tree.

case of variable dominance). After choosing another splitting variable, we suppose to reach a terminal case. The cost of the solution—assuming unit cost for all variables, is given by the number of variables set to one along the path, namely 4.

At any given node of the search tree, we have selected and rejected some variables. These variables are identified by the path from the root of the tree to that node. Hence, at that node we have a partial solution. Also, we maintain an **upper bound** on the cost of the optimum solution. Initially, the upper bound is set to a suitably large number. When we find a new best solution, we set the upper bound to its cost. From that point on, we shall not be interested in solutions that are not cheaper than the new best solution.

If the cost of a partial solution exceeds or equals the value of the upper bound at a node, clearly we can abandon that node and back up. Branch and Bound goes one step further and tries to establish whether a new best solution can still be found by proceeding from the current node. This assessment is based on computing a **lower bound** on the cost of completing the current partial solution. If the cost of the current partial solution plus the lower bound on the cost of completing it exceeds the current upper bound, then the current node is abandoned. The 'Bound' in Branch and Bound takes its name from this strategy.

The way of computing the lower bound depends on the particular problem. We shall examine a lower bound that applies to the covering problem. It is obvious that a careful choice of the lower bound criterion is important. Ideally, the criterion should provide an accurate estimate of the real minimum cost incurred in completing the current solution. At the same time, the computation of the bound should be fast. Before we turn our attention to the computation of the lower bound for the unate covering problem, we now take a look at the pseudo-code for the branch-and-bound algorithm that we have delineated so far that is given in Figure 4.10.

The pseudo code assumes that we work with the constraint equation. The program is initially called with an empty current solution (*currentSol*), the initial left-hand

```
BCP($F, U, currentSol$) {
1   ($F, currentSol$) = REDUCE($F, currentSol$)
    if ($terminalCase(F)$) {
        if (COST($currentSol$) $< U$) {
            $U$ = COST($currentSol$)
2           return ($currentSol$)
        }
3       else return ("no solution")
    }
4   $L$ = LOWER_BOUND($F, currentSol$)
    if ($L \geq U$) return ("no solution")
5   $x_i$ = CHOOSE_VAR($F$)
6   $S^1$ = BCP($F_{x_i}, U, currentSol \cup \{x_i\}$)
7   if (COST($S^1$) $= L$) return ($S^1$)
    $S^0$ = BCP($F_{x_i'}, U, currentSol$)
8   return BEST_SOLUTION($S^1, S^0$)
}
```

Figure 4.10: Branch-and-Bound Algorithm for the Unate Covering Problem.

side of the constraint equation (*F*) and the upper bound (*U*) set to the total cost of all variables plus one. This initial value of the upper bound exceeds the cost of any possible solution and hence guarantees that the first solution found will be accepted.

The procedure REDUCE (Line 1) iteratively finds essential variables and applies row and column dominance. It also updates the current solution accordingly. If the problem is now reduced to a terminal case, the procedure checks whether the solution thus found is better than the current best. The current solution is returned only if it is the new best (Line 2).

If the problem is cyclic, the lower bound is computed by the call to LOWER_BOUND (Line 4). In this subprocedure, we find an MIS by an internal call to MIS_QUICK and add the size of the current solution to the size of the MIS to get a lower bound. If there is still a chance of getting an optimum solution, a splitting variable is chosen (Line 5), and the first of the two subproblems are solved recursively. If the cost of this subproblem is equal to the lower bound, we immediately return (Line 7). Else, the second subproblem is solved recursively, and the best solution is returned (Line 8).

4.9.1 Choice of the Splitting Variable

The choice of the splitting variable has no effect on the correctness of the procedure, but it is important for its efficiency. A column that covers many rows is typically a better candidate than a column that covers few rows. The former is more likely to be part of an optimum solution. It is convenient to find a good solution soon, so that the upper bound is close to the optimum value, and more pruning of the search tree due to bounding is possible.

A possible refinement of the above strategy consists of favoring columns that cover many short rows. (A short row is one with few ones.) This criterion is based on the assumption that shorter rows have a lower chance of being covered. It can be implemented by assigning a weight to each row inversely proportional to the row length and by summing the weights of all rows covered by a column in order to determine the value of that column. The column with the highest value is chosen. This idea is explored further in Solved Problem 18.

4.9.2 Examples of Splitting and Lower Bounding

Let us consider some examples of the application of the BCP algorithm. We begin with a simple example that illustrates the flow of execution.

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \left[\begin{array}{cccccc} 1 & & 1 & & 1 & \\ 1 & & & 1 & & 1 \\ & 1 & 1 & & & 1 \\ & 1 & & 1 & 1 & \end{array}\right] \end{array}$$

The given matrix has 4 rows and 6 columns and is cyclic, so BCP sets $U = 7$ and skips to Line 4 and calls LOWER_BOUND. We cannot find any set of two independent rows. Therefore, we set the initial lower bound to 2 (using Theorem 4.8.1). If we split on Column 1, we get the solution {1, 2} in the positive half of the search space. This cost equals the lower bound. Hence, we don't need to explore the half of the search space where Column 1 is rejected. We thus return the solution {1, 2} at Line 7.

We next consider the example of Equation 4.7. This matrix has 6 rows and 6 columns and is also cyclic, so BCP again sets $U = 7$ and skips to Line 4. There it calls LOWER_BOUND which this time returns a lower bound of 3. BCP again chooses Column 1 as the splitting variable, and for $p_1 = 1$, the positive cofactor the matrix reduces to

$$\begin{array}{c} \\ 3 \\ 4 \\ 5 \\ 6 \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & 5 & 6 \end{array} \\ \left[\begin{array}{cccccc} & 1 & 1 & & & \\ & & 1 & 1 & & \\ & & & 1 & 1 & \\ & & & & 1 & 1 \end{array}\right] \end{array},$$

in which columns p_2 and p_6 are dominated. Then columns 3 and 5 become essential, leading to a terminal case with solution $S^1 = \{1, 3, 5\}$. Thus U is lowered to 3, and this solution is returned back up to the original recursive call. Then BCP compares the cost of this solution to L, S^1 is returned as the final solution. Again, we don't need to explore the half of the search space where Column 1 is rejected.

In both of these examples, lower bounding techniques have limited the implicit enumeration to the first half-space of the recursion. We now explore the application of BCP to a covering matrix with 13 rows and 11 columns, denoted by variables p_i, $i = 1, \ldots, 11$. In this richer example, all of the features of BCP are explicitly

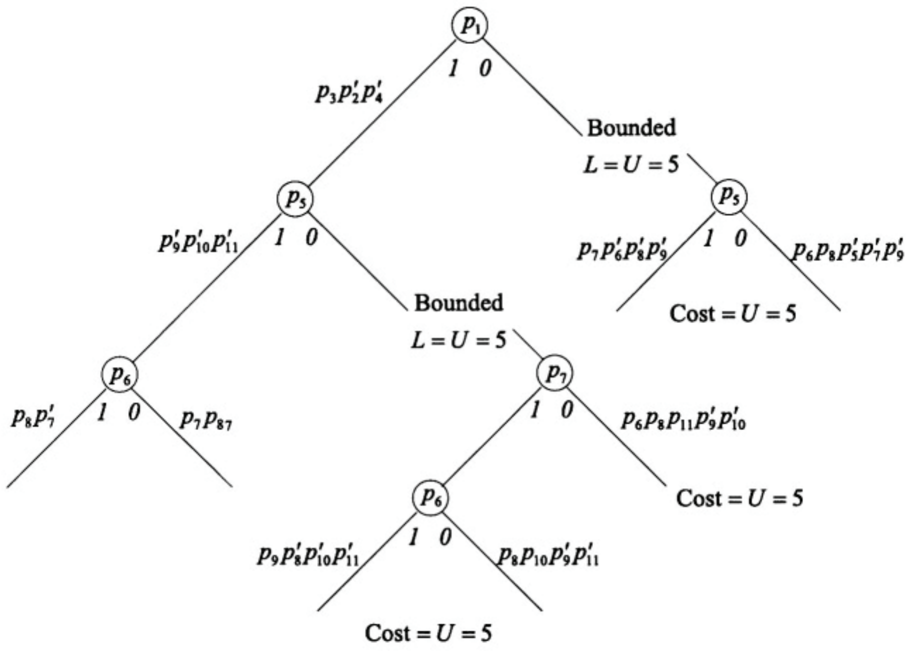

Figure 4.11: A search tree produced by Procedure BCP.

active.

$$
\begin{array}{c}
\begin{array}{ccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \end{array} \\
M = \left[\begin{array}{cccc|ccccccc}
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0
\end{array}\right]
\begin{array}{c} 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \\ 12 \\ 13 \end{array}
\end{array}
$$

We shall assume that the search tree (or recursion tree) of Figure 4.11 results from the application of algorithm BCP, given appropriate heuristic choices in the lower bound computation and in the choice of the splitting variable.

BCP begins by setting $U = 12$. Because of the block structure, it is not hard to see this matrix is cyclic (no reduction), so no implied variable assignments exist. Applying MIS_QUICK and using the simpler heuristic, BCP gets MIS={1,3,5,7}, and an initial lower bound $L = 4$.

Keeping in mind the block structure, we can ignore column counts, and split on the first variable. With $p_1 = 1$, rows 1, 4, 12 are covered, and we see that columns 2 and 4 are dominated, so we get a secondary essential column 3. The call to REDUCE leads to the following cyclic matrix.

$$M_{p_1} = \begin{array}{ccccccc|c} 5 & 6 & 7 & 8 & 9 & 10 & 11 & \\ \hline 1 & 1 & 0 & 0 & 0 & 1 & 0 & 5 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 6 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 7 \\ 0 & 1 & 0 & 1 & 0 & 1 & 1 & 8 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 9 \\ 1 & 0 & 0 & 1 & 1 & 0 & 0 & 10 \\ 1 & 0 & 1 & 0 & 0 & 0 & 1 & 11 \\ & & & & & & & \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 & 13 \end{array}$$

We may then split on a longest column p_5. For the positive cofactor the partial solution is $\{p_1, p_3, p_5\}$, and the reduced matrix is

$$M_{p_1 p_5} = \begin{array}{ccc|c} 6 & 7 & 8 & \\ \hline 1 & 1 & 0 & 6 \\ 0 & 1 & 1 & 7 \\ 1 & 0 & 1 & 8 \end{array}$$

Here we have observed that after deleting rows covered by p_5, columns 9, 10, and 11 are dominated.

This submatrix is cyclic, so we split on p_6. The assignment $p_6 = 1$ leads to a terminal case with solution $\{p_1, p_3, p_5, p_6, p_8\}$, and Cost $U = 5$. The assignment $p_6 = 0$ gives the solution $\{p_1, p_3, p_5, p_7, p_8\}$, with the same cost.

Since in this case $U > L$, we fall through at Line 7. We then obtain the solution S^0 obtained for the negative cofactor with respect to p_5. For the partial solution $\{p_1, p_3, p_5'\}$ the covering matrix is, after removing dominating row 8, as follows.

$$M_{p_1 p_3, p_5'} = \begin{array}{cccccc|c} 6 & 7 & 8 & 9 & 10 & 11 & \\ \hline 1 & 0 & 0 & 0 & 1 & 0 & 5 \\ 1 & 1 & 0 & 1 & 0 & 0 & 6 \\ 0 & 1 & 1 & 0 & 0 & 0 & 7 \\ 0 & 0 & 0 & 1 & 1 & 1 & 9 \\ 0 & 0 & 1 & 1 & 0 & 0 & 10 \\ 0 & 1 & 0 & 0 & 0 & 1 & 11 \\ 1 & 0 & 1 & 0 & 0 & 0 & 13 \end{array} \quad \Rightarrow L = 2 + 3 = 5$$

Here we note that rows 5,10, and 11 are independent, so with MIS={5,10,11}, we get a lower bound of $L = 2 + |\mathrm{MIS}| = 2 + 3 = 5$. Thus we know there is no better solution in this subspace, and immediately return. This is indicated by the legend "Bounded $U = L = 5$". At this point BCP is through with positive cofactor with respect to p_1, and next considers the negative cofactor.

However, it is interesting to see what is saved by this lower bound operation in this case. Without the lower bounding, since there is no reduction, we would split on p_7. Thus for $p_7 = 1$, we have the following.

$$M_{p_1,p_3,p_5'p_7} = \begin{array}{c|c} \begin{array}{ccccc} 6 & 8 & 9 & 10 & 11 \end{array} & \\ \left[\begin{array}{ccccc} 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 \end{array}\right] & \begin{array}{c} 5 \\ 9 \\ 10 \\ 13 \end{array} \end{array}$$

This matrix is again cyclic, so we split on p_6. For $p_6 = 1$, columns p_8, p_{10}, p_{11} are all dominated by column p_9, and we get the solution $\{p_1, p_3, p_7, p_6, p_9\}$ of cost 5. Similarly, for $p_6 = 0$, column p_{10} becomes essential, and after reduction p_8 does also, so we get the solution $\{p_1, p_3, p_7, p_{10}, p_8\}$ of cost 5.

Similarly for the partial solution $p_1 = p_3 = 1$, and $p_5 = p_7 = 0$, we get the following covering matrix.

$$M_{p_1p_5'p_7'} = \begin{array}{cc} \begin{array}{ccccc} 6 & 8 & 9 & 10 & 11 \end{array} & \\ \left[\begin{array}{ccccc} 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \end{array}\right] & \begin{array}{c} 5 \\ 6 \\ 7 \\ 8 \\ 9 \\ 10 \\ 11 \\ 13 \end{array} \end{array} \Rightarrow \begin{array}{c} p_9 = p_{10} = 0 \\ (p_6) = p_8 = p_{11} = 1 \end{array}$$

Here columns p_8 and p_{11} are essential, and, after deleting the rows they cover, and eliminating dominated columns, p_6 becomes (secondary) essential. Again, as expected cost = 5 for the solution $\{p_1, p_3, p_6, p_8, p_{11}\}$.

We see that a substantial amount of work is avoided by the lower bound comparison.

Now we return to the top of the recursion tree, and consider the result of excluding column p_1.

$$M_{p_1'} = \begin{array}{cccccccccc|c}
2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & \\
\hline
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 4 \\
0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 0 & 5 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 6 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 7 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 1 & 8 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 9 \\
0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 10 \\
0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 1 & 11 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 12 \\
0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 & 13
\end{array}$$

Here we see that columns p_2, p_4, p_{11} have become essential, and after the rows they cover are eliminated, column p_{10} is dominated by p_5. Also row 13 is now dominated by row 5. Thus $M_{p_1'}$ is equivalent to the following reduced (cyclic) matrix.

$$\begin{array}{ccccc|c}
5 & 6 & 7 & 8 & 9 & \\
\hline
1 & 1 & 0 & 0 & 0 & 5 \\
0 & 1 & 1 & 0 & 1 & 6 \\
0 & 0 & 1 & 1 & 0 & 7 \\
1 & 0 & 0 & 1 & 1 & 10
\end{array}$$

Here we see that for the partial solution $p_2 = p_4 = p_{11} = 1$ the reduced matrix has MIS= {5, 7}, so the lower bound is 3 + |MIS| = 3 + 2 = 5, so again it is known that this whole half of the solution space contains no solution of cost less than 5.

However, we can again check to see what we saved by the bounding operation. If we split on p_5, we see that for $p_5 = 1$, p_7 becomes essential, so we get a cost of 5, with a solution $p_1 = p_4 = p_{11} = p_5 = p_7 = 1$.

$$M_{p_1' p_5} = \begin{array}{cccc|c}
6 & 7 & 8 & 9 & \\
\hline
1 & 1 & 0 & 1 & 6 \\
0 & 1 & 1 & 0 & 7
\end{array} \Rightarrow \begin{array}{c} p_6 = p_8 = p_9 = 0 \\ p_7 = 1 \end{array}$$

Similarly, for $p_5 = 0$, we see that p_6, and then, after row deletion and column domin-