The minimum cover problem for the matrix M is to find a $\{0,1\}$ row vector x such that $M \cdot x^T \geq (1, \ldots, 1)^T$ and the sum $\sum_j x_j w_j$ is minimum. The minimum solution to this covering problem provides the minimum solution to the rectangle covering problem. The branch and bound technique for the minimum cover problem proposed in Chapter 2 can be used to find a minimum-cost rectangle cover.

Unfortunately, the weight of a rectangle in algebraic decomposition does not obey this assumption on the cost of a rectangle; as is shown in the next section, larger rectangles cost more than the rectangles they contain. Therefore, the minimum solution typically involves nonprime rectangles. To solve the problem exactly in this case requires enumeration of all rectangles, and not just the prime rectangles, for inclusion in the covering problem. Because of the tremendous number of rectangles present in even small problems, this is not a viable exact algorithm.

One approximate approach for rectangle covering is to find the minimum cover using prime rectangles and then apply a reduction step to reduce the size of the rectangles to improve the total cover cost. This reduction step, similar to the REDUCE operation in heuristic two-level minimization programs, is presented in Section 3.7.5.

3.5 Application of Rectangle Covering

In this section, the problems of distillation (*kernel-intersection extraction*) and condensation (*common-cube extraction*) are shown to be variations of the rectangle covering problem. The main result from this section is the derivation of a *weight* function and a *value* function for a rectangle for each of these two problems. These functions are defined for the optimization problem of minimizing the total number of literals in the network.

3.5.1 Common-Cube Extraction

Common-cube extraction is the process of finding cubes common to two or more expressions and extracting the common cube to simplify each of the expressions. The optimization problem is to find the particular cubes to introduce into the network to provide an optimal decomposition. For example, the optimal decomposition can be defined as minimizing the total number of literals summed over all expressions, or minimizing the total number of literals given a bound on the number of levels of logic in the final circuit.

Technology-dependent costs, such as the relative cost and delay of n-input NAND-gates can also be used to define the optimal decomposition.

Common-cube extraction has a relationship with rectangles and the rectangle covering problem as follows. First, the cube-literal matrix for the Boolean network is created. As before, each row corresponds to a cube of some expression, and each column corresponds to a literal present in some cube. The position B_{ij} is set to a 1 if cube i contains literal j. A rectangle in the cube-literal matrix identifies a cube which can be extracted from the network. The columns of the rectangle identify the literals in the common-cube, and the rows identify the cubes (and expressions) fed by the cube.

For example, given the equations:

$$\begin{aligned} F &= abc + abd + eg \\ G &= abfg \\ H &= bd + ef, \end{aligned}$$

the cube-literal matrix is:

		a	b	c	d	e	f	g
		1	2	3	4	5	6	7
F_1	1	1	1	1	.	.	.	.
F_2	2	1	1	.	1	.	.	.
F_3	3	.	.	.	.	1	.	1
G_1	4	1	1	.	.	.	1	1
H_1	5	.	1	.	1	.	.	.
H_1	6	.	.	.	.	1	1	.

The columns are annotated with the literal represented by the column, and the rows are annotated with the cube of the function which the row represents. For example F_1 is the cube abc.

The rectangle $(\{1,2,4\},\{1,2\})$ corresponds to the common cube ab. If this common cube is extracted as the new function X, the equations would be rewritten as:

$$\begin{aligned} F &= Xc + Xd + eg \\ G &= Xfg \\ H &= bd + ef \\ X &= ab. \end{aligned}$$

The process of extracting a cube modifies the Boolean network. A new node is added to the Boolean network with a logic function which is the common cube divisor. All

functions which the cube divides are replaced with the algebraic division of the function by the single cube. In order to extract cubes efficiently in an iterative algorithm, it is desirable to modify the cube-literal matrix incrementally to reflect the extraction of the cube. The advantage is that the cube-literal matrix does not have to be re-created as each cube is extracted.

The modification of the cube-literal matrix is straightforward. A new row is added to the cube-literal matrix to reflect the new single-cube expression added to the network, and a new column is added to represent the new literal in the network. The entries covered by the rectangle are marked with * to reflect that the position has been covered, but other rectangles are also allowed to cover the same position. The effect of rectangles which overlap in a position is considered in Section 3.6.

Continuing with the previous example, the cube-literal matrix, after extraction of the rectangle $(\{1,2,4\},\{1,2\})$ is:

		a	b	c	d	e	f	g	X
		1	2	3	4	5	6	7	8
F_1	1	*	*	1	.	.	.	.	1
F_2	2	*	*	.	1	.	.	.	1
F_3	3	.	.	.	.	1	.	1	.
G_1	4	*	*	.	.	.	1	1	1
H_1	5	.	1	.	1	.	.	.	.
H_1	6	.	.	.	.	1	1	.	.
X_1	7	1	1	.	.	.	.	.	.

Rectangle Weight and Value

The choice of the weight function for a rectangle measures the optimization goal for cube extraction. To minimize the total number of literals in the network, the weight of a rectangle is chosen so that the weight of a rectangle-cover of the cube-literal matrix equals the total number of literals in the network after the new single-cube functions are added to the network. Hence, the minimum-weighted cover corresponds to the optimal simultaneous extraction of a collection of cubes.

For cube extraction, the weight of a rectangle is defined as:

$$w(R,C) = \begin{cases} |C| & \text{if } |R| = 1 \\ |R| + |C| & \text{if } |R| > 1 \end{cases}$$

If a rectangle (R,C) has only a single row, this corresponds to leaving the cube unchanged in the network; hence, the weight of this rectangle counts the number of literals in the cube.

If the rectangle has more than one row, this corresponds to creating a new single-cube function (with $|C|$ literals), and substituting this new function into $|R|$ other cubes at a cost of $|R|$ literals; hence the weight of a multiple-row rectangle is $|R| + |C|$.

When searching for a rectangle to extract, it is useful to define a second function called the *value* of a rectangle. For cube extraction, the value of a rectangle is defined as:

$$v(R,C) = |\{(i,j)|B_{ij} = 1\}| - w(R,C).$$

The rectangle value reflects the desirability of choosing the rectangle, and is defined as the number of literals which would be saved in the network if this rectangle is chosen. This is simply the number of 1 points covered by the rectangle minus the weight of the rectangle. No additional literals are saved for covering a point marked as an asterisk in the matrix; hence, these are not counted in the value function. If a rectangle contains only points which are 1, then the value of a rectangle for cube extraction is the area minus the perimeter.

It is useful to define the weight function and value function in terms of three auxiliary functions – the row and column weight vectors w^r and w^c and the element value matrix V. Each row i has weight $w_i^r = 1$, and each column j has weight $w_j^c = 1$. Each position of the matrix V has $V_{ij} = 1$. As each element B_{ij} which is 1 is covered, the value of V_{ij} is set to 0. The weight of a rectangle (R,C) is then defined as

$$w(R,C) = \begin{cases} \sum_{i \in C} w_j^c & \text{if } |R| = 1 \\ \sum_{i \in R} w_i^r + \sum_{i \in C} w_j^c & \text{if } |R| > 1 \end{cases}$$

and the value of a rectangle (R,C) is defined as

$$v(R,C) = \sum_{i \in R, j \in C} V_{ij} - w(R,C).$$

The rectangle algorithms presented in Section 3.7 make use of w_i^r, w_j^c, and V_{ij} to compute $v()$.

3.5.2 Kernel-Intersection Extraction

As described in Theorem 3.3.1, intersections among the kernels of a collection of expressions are useful for finding common multiple-cube divisors between two or more expressions. If two functions share a common multiple-cube divisor, then the common divisor can be found as the intersection of a kernel from each of the functions.

To turn this into an optimization algorithm, the first step is to enumerate all kernels of each logic expression. If desired, the set of kernels is restricted to a subset of all kernels to reduce the processing time at the possible expense of a decrease in the solution quality. For example, a convenient subset is the set of all level-0 kernels. The problem is then to examine the intersections over the subsets of the set of kernels to find intersections to extract and substitute into a network. This problem is naturally mapped into a rectangle covering problem.

The Co-kernel Cube Matrix

Finding useful intersections of kernels is facilitated with the *co-kernel cube* matrix. A row in this matrix corresponds to a kernel (and its associated co-kernel), and a column corresponds to the cubes which are present in some kernel. The entry B_{ij} is set to 1 if the kernel i contains the cube j.

For example, given the equations:

$$\begin{aligned} F &= af + bf + ag + cg + ade + bde + cde \\ G &= af + bf + ace + bce \\ H &= ade + cde, \end{aligned}$$

the kernels (and co-kernels) of F are $de+f+g(a)$, $de+f(b)$, $a+b+c(de)$, $a+b(f)$, $de+g(c)$ and $a+c(g)$. The kernels (and co-kernels) of G are $ce+f(a,b)$, $a+b(f,ce)$, and the only kernel of H is $a+c(de)$. For ease of presentation, the functions F and G, which themselves are kernels, are not listed in the set of kernels. The co-kernel cube matrix is easily constructed from this data. The unique cubes from all of the kernels are a, b, c, ce, de, f, and g; these cubes are used to label the columns of the matrix. There are thirteen kernels, and the corresponding co-kernels are used to label the rows of the matrix.

The product of a co-kernel for a row and the kernel-cube for a column yields a cube of some expression. For reference, the cubes of the original expressions are numbered from 1 to 13. The number of the cube resulting from the product of the co-kernel for row i and the kernel-cube for column j is placed at position B_{ij} in the co-kernel cube matrix. For example, the co-kernel a when multiplied by the kernel $de+f+g$ yields the cubes numbered 5, 1, and 3, which are ade, af, and ag. Note that there is often more than

one way to form each cube in an expression. For example, cube 1 (af) is created by the co-kernel a multiplying the kernel $de + f + g$, and by the co-kernel f multiplying the kernel $a + b$.

The co-kernel cube matrix for the previous example is:

			a	b	c	ce	de	f	g
			1	2	3	4	5	6	7
F	a	1	.	.	.	.	5	1	3
F	b	2	.	.	.	.	6	2	.
F	de	3	5	6	7	.	.	.	.
F	f	4	1	2	.	.	.	.	.
F	c	5	.	.	.	.	7	.	4
F	g	6	3	.	4	.	.	.	.
G	a	7	.	.	.	10	.	8	.
G	b	8	.	.	.	11	.	9	.
G	ce	9	10	11	.	.	.	.	.
G	f	10	8	9	.	.	.	.	.
G	de	11	12	.	13	.	.	.	.

A rectangle of the co-kernel cube matrix identifies an intersection of kernels; this kernel-intersection is a common subexpression in the network. The columns of the rectangle identify the cubes in the subexpression, and the rows of the rectangle identify the particular functions that the subexpression divides. The entries covered by the matrix correspond to cubes from the original network.

From the previous example, the prime rectangle $(\{3,4,9,10\},\{1,2\})$ identifies the subexpression $a + b$ which divides the functions F and G. Cubes numbered 1, 2, 5, 6, 8, 9, 10, and 11 from the original set of functions are covered by this rectangle. This corresponds to the factorization of the equations into the form:

$$\begin{aligned} F &= deX + fX + ag + cg \\ G &= ceX + fX \\ H &= ade + cde \\ X &= a + b. \end{aligned}$$

When a new subexpression is identified, it is inserted into the Boolean network. This consists of adding a new node to the network and dividing the node into each of the expressions which this node divides. The expressions which the subexpression divide are

apparent from the rows in the rectangle for the subexpression. The new co-kernel cube matrix is then created for the modified Boolean network.

To reduce the complexity of extracting each factor from the network, it is desirable to modify the co-kernel cube matrix incrementally as each subexpression is identified. This is done as follows. New rows are added to the co-kernel cube matrix for each kernel of the new subexpression. The cubes which are formed by the insertion of this new factor into the network are marked as covered in the rectangle. This includes the points directly contained by the rectangle, and other points which are labeled with the same number. The cubes which have been covered by the new subexpression are labeled with *. The effect of rectangles which overlap in a position is considered in Section 3.6.

Continuing the previous example, after the rectangle $(\{3,4,9,10\},\{1,2\})$ is extracted, the modified co-kernel cube matrix is:

			a	b	c	ce	de	f	g
			1	2	3	4	5	6	7
F	a	1	.	.	.	.	*	*	3
F	b	2	.	.	.	.	*	*	.
F	de	3	*	*	7	.	.	.	.
F	f	4	*	*		.	.	.	.
F	c	5	.	.	.	.	7	.	4
F	g	6	3	.	4	.	.	.	.
G	a	7	.	.	.	*	.	*	.
G	b	8	.	.	.	*	.	*	.
G	ce	9	*	*		.	.	.	.
G	f	10	*	*		.	.	.	.
H	de	11	12	.	13	.	.	.	.
X	1	14	14	15	.	.	.	.	.

Note that no new columns were added in this case when the co-kernel cube matrix was modified.

Rectangle Weight and Value

The weight of a rectangle of the co-kernel cube matrix is chosen to reflect the number of literals in the network if the corresponding common subexpression is inserted into the network. A minimum-weighted rectangle-cover of the co-kernel cube matrix then corresponds to a simultaneous selection of a set of subexpressions to add to the network in order to minimize the total number of literals in the network.

In a manner similar to common-cube extraction, weights w_i^r and w_j^c are defined for the rows and columns of the matrix and the weight of a rectangle is defined in terms of these weights. Also, values V_{ij} are defined for the elements of the matrix, and the value of a rectangle is defined in terms of the values of the elements covered by the rectangle, and the weight of the rectangle.

Let w_j^c be the number of literals in the kernel-cube for column j. If a rectangle (R, C) is used to identify a subexpression, then a new function is formed from the columns of C. This new function has $\sum_{j \in C} w_j^c$ literals. Let w_i^r be the number of literals in the co-kernel for row i plus one. A subexpression divides the expressions indicated from the rows R of the rectangle. After algebraic division by the subexpression, each of these expressions consist of a sum of the corresponding co-kernel cubes multiplying the literal for the new expression. Therefore, the number of literals in the affected functions after extraction of the rectangle is $\sum_{i \in R} w_i^r$.

Therefore, the weight of a rectangle (R, C) of the co-kernel cube matrix is defined as:

$$w(R, C) = \sum_{i \in R} w_i^r + \sum_{j \in C} w_j^c.$$

The value of a rectangle measures the difference in the number of literals in the network if the particular rectangle is selected. The number of literals after the rectangle is selected is the weight of the rectangle as defined above. Let V_{ij} be the number of literals in the cube which is covered by position (i, j) of the co-kernel cube matrix. Then, the number of literals before extraction of the rectangle is simply $\sum_{i \in R, j \in C} V_{ij}$. As elements of the co-kernel cube matrix are covered, elements of V are set to zero. This includes the elements V_{ij} covered by the matrix and all other elements which represent the same cube in the network.

The value of a rectangle (R, C) of the co-kernel cube matrix is thus defined as:

$$v(R, C) = \sum_{i \in R, j \in C} V_{ij} - w(R, C)$$

Note that the weight function and value function are the same form as for common-cube extraction; only the definitions for w_i, w_j, and V have changed.

3.6 Effect of Overlapping Rectangles

The formulation of algebraic decomposition as a rectangle covering problem where the rectangles are allowed to overlap allows for valid decompositions which are nonalgebraic. Further, by introducing don't-care points in the matrix, other nonalgebraic decompositions are possible. This effect is described in this section for both cube-extraction and kernel-extraction.

3.6.1 Cube-Extraction

Consider two rectangles which overlap in the cube-literal matrix. Recall that each point of the cube-literal matrix corresponds to a literal of some cube in the network. For ease of presentation, assume the rectangles overlap in a single point in the matrix and call the cube which contains this point the *overlap cube*. The simultaneous extraction of both rectangles corresponds to duplicating the literal which is contained by both rectangles. The literals of the overlap cube which are covered by both rectangles are replaced with the product of the two extracted cubes. However, the two new cubes do not have disjoint support; hence, this corresponds to a nonalgebraic factoring of the original equation. This operation is valid because the Boolean identity $aa = a$ has been used. The duplication of literals, while seeming to introduce extra literals in the network, can lead to better decompositions.

Consider the cube-extraction example from the previous section. The common cube ab (prime rectangle $(\{1,2,4\},\{1,2\})$) has been extracted and the cube-literal matrix has the form:

		a	b	c	d	e	f	g	X
		1	2	3	4	5	6	7	8
F_1	1	*	*	1	.	.	.	.	1
F_2	2	*	*	.	1	.	.	.	1
F_3	3	.	.	.	.	1	.	1	.
G_1	4	*	*	.	.	.	1	1	1
H_1	5	.	1	.	1	.	.	.	.
H_1	6	.	.	.	.	1	1	.	.
X_1	7	1	1	.	.	.	.	.	.

Assume the rectangle $(\{2,5\},\{2,4\})$ is selected as the next cube to extract. This rectangle overlaps the first rectangle in the point $(2,2)$ which duplicates the literal b in cube

F_2. This corresponds to rewriting the equations as:

$$\begin{aligned} F &= Xc + XY + eg \\ G &= Xfg \\ H &= Y + ef \\ X &= ab \\ Y &= bd. \end{aligned}$$

This is a nonalgebraic decomposition because the product XY is used in F, but X and Y do not have disjoint support.

Recall that the weight of each rectangle reflects the number of literals after extraction of each rectangle; the weight of the cover is the final number of literals in the network. The minimum-weighted rectangle-cover selects precisely the literals to duplicate to optimize the decomposition.

3.6.2 Kernel Extraction

Likewise, rectangles are allowed to overlap in the co-kernel cube matrix. The points of the overlap correspond to duplication of the cubes which are contained by both rectangles. The Boolean identity $a + a = a$ is being used which makes this technique valid.

Consider the previous kernel-extraction example. After the subexpression $a + b$ (corresponding to the prime rectangle $(\{3,4,9,10\},\{1,2\})$) has been extracted, the co-kernel cube matrix is:

			a	b	c	ce	de	f	g
			1	2	3	4	5	6	7
F	a	1	.	.	.	.	*	*	3
F	b	2	.	.	.	.	*	*	.
F	de	3	*	*	7	.	.	.	.
F	f	4	*	*	.	.	.	.	.
F	c	5	.	.	.	.	7	.	4
F	g	6	3	.	4	.	.	.	.
G	a	7	.	.	.	*	.	*	.
G	b	8	.	.	.	*	.	*	.
G	ce	9	*	*	.	.	.	.	.
G	f	10	*	*	.	.	.	.	.
H	de	11	12	.	13	.	.	.	.
X	1	14	14	15	.	.	.	.	.

If the rectangle $(\{3,6,11\},\{1,3\})$ is extracted next, then the overlap in the point $(3,1)$ corresponds to the duplication of the term ade for the decomposition of function F. The rectangle $(\{3,6,11\},\{1,3\})$ corresponds to the subkernel $a+c$, which after extraction, yields the equations:

$$\begin{aligned} F &= deX + fX + deY + gY \\ G &= ceX + fX \\ H &= deY \\ X &= a+b \\ Y &= a+c. \end{aligned}$$

Sharad Malik has made the observation that it is possible to add don't-care points to the co-kernel cube matrix before finding the rectangle cover, as follows. Assume $B_{ij} = 0$. If the co-kernel c_i contains literal l ($\bar{l}$) and kernel cube k_j contains literal $\bar{l}$ (l), then set $B_{ij} = *$; that is, allow the position B_{ij} to be optionally covered if it leads to a better factorization. Given that $c_i k_j = \emptyset$, the decomposition remains valid even if a rectangle covers this position.

Len Berman has suggested inserting don't-cares at other positions where it can be proved that the addition of the implied cube to the corresponding function does not change the input-output behavior of the logic network. Berman has suggested a technique based on global-flow to detect efficiently when this condition is satisfied.

3.7 Rectangle Algorithms

Two algorithms for applying rectangle covering to algebraic decomposition are proposed. The first, **greedy_extract**, selects one rectangle at a time and modifies the matrix to reflect the extraction of the rectangle. The advantage of this technique is that it immediately takes into account common factors between the newly extracted function and the rest of the logic network. The disadvantage of this approach is that it selects only one rectangle at a time and does not easily account for the simultaneous extraction of multiple rectangles. Therefore, **covering_extract** is also presented. This algorithm finds the best collection of factors to extract at each step by solving the minimum-weighted rectangle-covering problem heuristically. These rectangles are then extracted, and the entire process

is repeated to find factors between the new expressions and the remainder of the logic network.

Several other algorithms are also described in this section. **gen_rectangles** is an algorithm for enumerating all prime rectangles in a matrix. This is used to generate the kernels of a logic expression, and is used by **best_rectangle** to find the best-valued prime rectangle in a matrix. **ping_pong** provides a heuristic alternative to **best_rectangle** to find a good-valued, but not necessarily the best-valued, rectangle in a matrix.

gen_rectangles is an adaptation of the algorithm originally described in [17] as a technique for kernel generation. The computer implementation described there was done in APL using full matrix techniques. Even though IBM APL uses a bit-matrix representation for $\{0,1\}$-matrices, using full matrices proved to be a limiting factor on the size of networks which could be optimized in YLE [13]. Other full-matrix implementations of this algorithm have been described, including a version by Karen Bartlett [8] and Albert Wang [77].

However, almost all examples of rectangle generation for logic synthesis involve matrices with very few nonzero entries. For example, in the case of common-cube extraction, a network with 1,000 single-cube expressions each with 10 literals creates a cube-literal matrix with 1,000,000 possible entries of which only 10,000 are nonzero. This matrix provides an example a dense matrix for logic optimization, but is only one percent full. Therefore, sparse matrices appear to be a natural representation for implementing and describing the rectangle covering algorithms.

To assist the implementation of algorithms for rectangle covering, a generic package for sparse matrices has been implemented. This package has proven useful for a variety of other applications, including algorithms for unate covering (as described in Chapter 2 of this thesis) and inverter-phase assignment [76]. A sparse matrix stores only the nonzero elements of a matrix. Stored with each sparse matrix element is the row and column number. Space is reserved at each sparse matrix element for application-specific information. The sparse matrix elements are doubly-linked to the next and previous elements in the row, and are doubly-linked to the next and previous elements in the column. Headers for each row and column provide random access to the first element in a row or column, given the row or column number. A doubly-linked list of the nonempty row and column headers are also maintained to provide fast access to the currently occupied rows and columns. As submatrices of the sparse matrix are extracted, the row and column numbers are not changed; this provides convenient correlation between a row in the sparse matrix, and a

row in any submatrix of the matrix.

Pseudo-code for the algorithms in this section are given in an informal pseudo-C notation. The sparse matrix package provides a number of primitive operations which are used in the description. A sparse matrix element has fields *rownum* and *colnum* which give the row and column number of the element. A sparse matrix row or column has a field *length* which gives the number of nonzero elements in that row or column. Note that in practice the sparse matrix element contains arbitrary user information at the given row and column; however, many algorithms are only interested in the topology of the sparse matrix (i.e., where the nonzero elements are) and not their contents. A rectangle has two fields, the set of rows in the rectangle (*rows*), which can be represented with a sparse matrix column vector, and the set of columns in the rectangle (*cols*), which can be represented with a sparse matrix row vector.

3.7.1 gen_rectangles: Finding All Prime Rectangles

gen_rectangles finds all prime rectangles in a matrix. As mentioned earlier, the worst-case complexity of this algorithm is exponential in the size of the matrix. However, when the matrix is sparse, it is often feasible to enumerate all prime rectangles.

The pseudo-code for the gen_rectangles is given in Figure 3.3. gen_rectangles calls gen_rectangles_recur with appropriate initial arguments. The final step is to process and record the trivial prime rectangles. A single row (column) is a prime rectangle if it is not contained in any other row (column). Also, if the matrix does not contain any columns (rows) with all 1's, then the rectangle consisting of all of the rows (columns) and no columns (rows) is also prime.

The arguments to gen_rectangles_recur are the current submatrix which is being searched for prime rectangles (*M*), the current column index (*index*), and the rectangle found up to this point (*rect*). Also passed to this routine are a function to be called when a prime rectangle is discovered (*func*), and a generic piece of state information to be passed to this function (*state*).

The recursive assumption on this routine is that all of the rows of *rect* contain a 1 (or *) for all columns of *rect*. The routine will search the submatrix *M* to find all of the prime rectangles with fewer rows but more columns.

Each column *c* with an index greater than the starting index is examined as a

column to include in the rectangle. If the column has only a single element, then it cannot create a nontrivial rectangle, so only columns with 2 or more elements are of interest. The submatrix *M1* of the original matrix is created by selecting only the rows where the column c has a nonzero value, and a new rectangle is formed from the columns of the old rectangle and the rows for which c has a nonzero value.

Any column of the submatrix *M1* (including c) which is now all 1's can also be added to the rectangle. A pruning operation is also performed at this step. If a column of 1's occurs for a column index less than the starting index, then all rectangles in the current submatrix *M1* have already been examined when that column index was processed. Hence, if this condition is detected, it is not necessary to recur.

At this point, *rect1* represents a new prime rectangle of the matrix, and *M1* is a new submatrix to be searched for more prime rectangles. The caller's function is called to process the prime rectangle. This function returns a status indicating whether the submatrix *M1* should be searched further. If the return value is 0, **gen_rectangles** is called recursively.

3.7.2 best_rectangle: Finding a Maximum-Value Prime Rectangle

Given a technique to generate all prime rectangles, it is now trivial to find the best-valued prime rectangle. **gen_rectangles** is used to find all prime rectangles, and the best-valued prime rectangle is recorded.

Note that for the value function defined in terms of V, w^r, and w^c, the best-valued rectangle is not necessarily prime. The value of the prime rectangle can be improved by deleting rows and columns which do not increase the value. If a row (column) of the prime rectangle does not contain any elements of positive value (i.e., $V_{ij} = 0$ for all $i \in R$), then that row (column) can be deleted from the rectangle.

Bounding Techniques

For common-cube extraction in a matrix with no don't-care entries, the value function takes the simple form of the area of the rectangle minus the semi-perimeter; i.e., $v(R, C) = |R||C| - (|R| + |C|)$. In this special case, a bound is easily placed on the size of the largest value rectangle in a matrix. This can be used to speed-up the search for the best-valued prime rectangle.

This is done by converting **gen_rectangles** into a branch and bound algorithm.

```
gen_rectangles_recur(M, index, rect, func, state) {

    foreach column c of M {
        if (c->length >= 2 && c->colnum >= index) {

            M1 = new matrix;
            foreach element p in column c {
                copy row p->rownum of M to M1;
            }

            rect1 = new rectangle;
            rect1->rows = duplicate column c of M;
            rect1->cols = duplicate rect->cols;

            prune = 0;
            foreach column c1 of M1 {
                if (c1->length == c->length) {
                    if (c1->colnum < c->colnum) {
                        prune = 1;
                        break;
                    } else {
                        add c1->colnum to rect1->cols;
                        delete column c1 from M1;
                    }
                }
            }

            if (not prune) {
                bound = (*func)(M1, rect1, state);
                if (not bound) {
                    gen_rectangles_recur(M1, c->colnum,
                                         rect1, func, state);
                }
            }
        }
    }
}
```

Figure 3.2: Algorithm gen_rectangles_recur.

```
gen_rectangles(M, func, state) {
    foreach trivial rectangle rect {
        if the rectangle is prime {
            (*func)(rect, state);
        }
    }
    rect = new rectangle;
    gen_rectangles_recur(M, 0, rect, func, state);
}
```

Figure 3.3: Algorithm gen_rectangles.

At each step of gen_rectangles_recur, assume a rectangle rect1= (R, C) is identified and M1 = $\tilde{M}$ is the remaining submatrix. Assume row i of $\tilde{M}$ contains the most nonzero entries; likewise, assume column j of $\tilde{M}$ contains the most nonzero entries. Let $n_r = |k|\tilde{M}_{jk} = 1|$ and $n_c = |k|\tilde{M}_{ik} = 1|$. Then the largest rectangle contained in $\tilde{M}$ adds at most n_c columns to the current rectangle and has at most n_r rows. Therefore, a simple bound on the value of the best rectangle for the remaining subproblem is $(|C| + n_c)n_r - (|C| + n_c + n_r)$. If this is less than or equal to the value of the best rectangle seen, then the search through the rectangles of $\tilde{M}$ can be avoided. For this reason, the function func which is passed to gen_rectangles is allowed to return a status indicating whether the remaining submatrix should be examined.

The branching of gen_rectangles_recur can also be modified to attempt to find a large rectangle as soon as possible to allow the bounding to be more effective. Rather than iterating for the columns of M in arbitrary order, the columns are ordered by decreasing count of the number of nonzero elements in each column.

A better bounding scheme uses the observation that for the matrix $M1$ to have a rectangle of size l rows by k columns, then at least l of the rows must have more than k elements. Let $r_i, i = 1, \ldots, n$ be the number of nonzero elements in each row in descending order (i.e., $r_1 \geq r_2 \cdots \geq r_n$), and let $c_j, j = 1, \ldots, m$ be the number of nonzero elements in each column also in descending order. Then, a tighter bound on the size of the value of the

best rectangle in $M1$ (and rect1 $= (R, C)$) is

$$\max\{(|C| + r_{c_i})c_i - (|C| + r_{c_i} + c_i) | i = 1, \ldots, m\}$$

That is, for each value of i, the row cardinality of rank c_i is the bound on the number of rows that can form a rectangle with c_i columns.

Experimental results comparing these bounding schemes is presented in Section 3.7.2. The simple bounding technique is effective because it is inexpensive and it reduces the number of rectangles visited. The tighter bound is expensive in practice to implement; sorting the row and column cardinalities at each step is expensive, and the reduction in the number of rectangles examined does not appear to justify the second bounding technique.

3.7.3 ping_pong: Finding a Maximal-Value Rectangle

ping_pong is a heuristic algorithm to find a good-valued rectangle without generating all prime rectangles of the matrix. The inputs to the algorithm are the matrix B and the value function $v()$ which computes the value of a rectangle of B. $v()$ is itself defined in terms of the row and column weights (w^r and w^c) and the value matrix (V).

ping_pong is given in Figure 3.4. **ping_pong_row** is used to find a rectangle starting from the *best* row in the matrix. To make **ping_pong** less dependent on the starting row, **ping_pong_col** is used to find a rectangle starting from the *best* column in the matrix. The best rectangle between these two passes is the final rectangle returned.

In the description of **ping_pong**, the row-oriented algorithms **ping_pong_row** and **greedy_row** are described. The analogous routines **ping_pong_col** and **greedy_col** are the same algorithms applied to the transpose of the matrix B with the weight function adjusted accordingly; hence, these algorithms are not described in detail.

ping_pong_row is given in Figure 3.5. The row i which maximizes the value of the single row rectangle is chosen as the starting seed. This becomes the seed row for **greedy_row** which is used to find a high-value rectangle intersecting row i. This rectangle becomes the current best rectangle (R_b, C_b). The iteration loop of **ping_pong_row** tries to improve the value of this rectangle. This is done first using **greedy_col**, but restricting the initial column seed to one of the columns in the current best rectangle. The rectangle returned will either be the rectangle (R_b, C_b), or will be a rectangle of higher value. If the rectangle value has improved, the new rectangle is recorded as the best rectangle, and the

```
ping_pong(B, v) {
    /* Find a good rectangle starting from the "best" row */
    (R1, C1) = ping_pong_row(B, v);

    /* Find a good rectangle starting from the "best" column */
    (R2, C2) = ping_pong_col(B, v);

    if (v(R1, C1) > v(R2, C2)) {
        (R, C) = (R1, C1);
    } else {
        (R, C) = (R2, C2);
    }
    return (R, C);
}
```

Figure 3.4: Algorithm ping_pong.

process is repeated starting from a row chosen from the set of columns in the best rectangle. This process is repeated until no better rectangle is found.

All that remains is the description for **greedy_row**, which is shown in Figure 3.6. **greedy_row** finds a good-valued rectangle which intersects row i. Row i becomes the seed rectangle (R_s, C_s) and the best rectangle seen (R_b, C_b) is initialized to the seed rectangle. During each pass of the loop, all rows not currently in the seed rectangle are examined, and the row k which, if added to the seed rectangle, maximizes the value of the seed rectangle is chosen. This row is then added to the seed rectangle, and columns not in both the seed column set and row k are deleted from the seed rectangle. This is repeated until the seed rectangle consists of only a single column. A sequence of rectangles with increasing row sets and decreasing column sets is generated in this manner; the best value rectangle seen in this process is returned.

In **ping_pong_row**, if the initial row i leads to a rectangle (R_b, C_b) which is trivial (i.e., a rectangle with only a single row or column), then the next best initial row should be chosen instead, continuing until all rows have been tried as an initial row. If this is done and **ping_pong** returns a trivial rectangle, then there is the assurance that no nontrivial

```
ping_pong_row(B, v) {

    /* Find the seed row of maximum value */
    i = arg max_i {v({i}, {j|B_ij = 1})};
    (R_b, C_b) = greedy_row(B, v, i);

    /* Try to improve the rectangle */
    do {
        /* start from best-valued column of (R_b, C_b) */
        j = arg max_j {v({i|B_ij = 1}, {j})|j ∈ C_b};
        (R_1, C_1) = greedy_col(B, v, j);
        if (v(R_1, C_1) > v(R_b, C_b)) {
            (R_b, C_b) = (R_1, C_1);
        } else {
            break;
        }

        /* start from the best-valued row of (R_b, C_b) */
        i = arg max_i {{i}, v({j|B_ij = 1})|i ∈ R_b};
        (R_2, C_2) = greedy_row(B, v, i);
        if (v(R_2, C_2) > v(R_b, C_b)) {
            (R_b, C_b) = (R_2, C_2);
        } else {
            break;
        }
    } while (1);
    return (R_b, C_b);
}
```

Figure 3.5: Algorithm **ping_pong_row**.

```
greedy_row(B, w^r, w^c, V, i) {
    (R_s, C_s) = ({i}, {j|B_ij = 1});
    (R_b, C_b) = (R_s, C_s);
    while (|C_s| > 1) {
        k = arg max_k {v(R_s ∪ {k}, C_s ∩ {j|B_kj = 1})|k ∉ R_s};
        (R_s, C_s) = (R_s ∪ {k}, C_s ∩ {j|B_kj = 1});
        if v(R_s, C_s) > v(R_b, C_b) {
            (R_b, C_b) = (R_s, C_s);
        }
    }
    return (R_b, C_b);
}
```

Figure 3.6: Algorithm **greedy_row**.

rectangle exists in the matrix.

ping_pong can be implemented efficiently for a sparse-matrix because all of the operations have a complexity related to the sparsity of the matrix. For example, finding the next row to add to the seed rectangle requires examining only the rows which are connected to a column in the seed row; because the matrix is stored with row and column pointers, this requires examining only a subset of the rows in the matrix. Also, incremental computation of the value and cost for each rectangle as rows and columns are added to the rectangle can be used to reduce the complexity of finding the cost for a rectangle.

3.7.4 greedy_extract: Greedy Selection of Rectangles

greedy_extract is given in Figure 3.7. The inputs are the matrix B and the value function $v()$. The value function is represented by the row and column weights w^r and w^c, and the value matrix V. A rectangle is chosen in **choose_rectangle** by either generating all prime rectangles and choosing the best-valued rectangle (**best_rectangle**), or by finding a good-valued rectangle heuristically (**ping_pong**). The matrices are then modified to reflect the extraction of the common factor as described in Section 3.5.

```
greedy_extract (B,v) {
    do {
        (R,C) = choose_rectangle(B,v);
        if ( v(R,C) > 0) {
            modify B to reflect extraction of (R,C);
        }
    } while ( v(R,C) > 0 );
}
```

Figure 3.7: Algorithm **greedy_extract**.

For common-cube extraction, an additional row, representing the new common cube factor, is added to the matrix B. A new column, representing the fanout of the cube factor, is also added to the matrix B. The value for any entry covered by the rectangle is set to zero. On subsequent iterations, no value is recorded for covering one of these already covered points.

For kernel-intersection extraction, the kernels of the new expression are generated and included in the co-kernel cube-matrix. The value for all entries corresponding to a cube which is covered by the rectangle is set to zero.

The process of selecting a good rectangle and extracting the rectangle is iterated while the value of the rectangle returned by **choose_rectangle** is positive. Recall that the rectangle value for both cube extraction and kernel extraction is defined to be the number of literals saved in the network by extracting the rectangle; hence, the algorithm terminates when no factors further reduce the number of literals in the network.

3.7.5 covering_extract: Simultaneous Selection of Rectangles

An alternate approach to the greedy nature of **greedy_extract** is to find a minimum-weight rectangle-cover and then simultaneously extract all of the rectangles from the matrix. This algorithm is called *covering_extract* and is shown in Figure 3.8.

First, an optimal prime rectangle cover is found using the **rect_prime_cover**. The rectangles are then incrementally modified to obtain a cover with a smaller total cost. The incremental modifications delete redundant rectangles (**rect_irredundant**) and reduce

```
covering_extract(B) {
    P = rect_prime_cover(B);
    P = rect_irredundant(P, B);
    P = rect_reduce(P, B);
    extract the rectangles of P;
}
```

Figure 3.8: Algorithm **covering_extract**.

```
rect_prime_cover(B) {
    P = 0;
    while (there are uncovered points in B) {
        (R, C) = choose_rectangle(B);
        add (R, C) to the rectangle cover P;
        set V(i,j) = 0 for all i in R and j in C;
    }
    return P;
}
```

Figure 3.9: Algorithm **rect_prime_cover**.

the total cost by trimming the rectangles without leaving 1's in the matrix uncovered (**rect_reduce**). With the assumption that rectangle weight is positive and increases with increasing size of the rectangle, both of these operations reduce the total cost of the cover.

rect_prime_cover chooses prime rectangles using either **best_rectangle** or **ping_pong**. After a rectangle is added to the cover, the points in the rectangle are marked as covered. Subsequent rectangles take into account that no benefit is realized from covering these same points again. This is iterated until every 1 in B is covered by some rectangle.

rect_irredundant is a modification of the **rect_reduce** algorithm and hence **rect_reduce** is presented first. Both of these algorithms attempt to improve the cost

of the rectangles selected for the cover.

rect_reduce first counts the number of times each point in the matrix B is covered. Then the essential points for each rectangle are determined. For a given rectangle, if a point is covered only by that rectangle (i.e., the count on the number of times the point is covered is 1), then the row and column for that point are essential for the rectangle. After checking all points in the rectangle, the rectangle is replaced with its essential parts, and the counts are modified to reflect replacement by the new rectangle.

If a rectangle is completely covered by other rectangles, the essential parts of the rectangle will be empty, and *rect_reduce* will delete the rectangle. However, the order in which the rectangles are processed in *rect_reduce* is significant. The reduction of one rectangle may block the reduction or removal of a rectangle which is processed later. Hence a simple heuristic is used to order the rectangles before processing by sorting them in decreasing order by size.

One problem with **rect_reduce** is that if the rectangles are processed in the wrong order, a redundant rectangle may not be detected. Therefore, a simple modification of *rect_reduce*, called *rect_irredundant*, is used first to detect and remove all redundant rectangles. *rect_irredundant* determines the essential row and column sets for each rectangle, but only modifies a rectangle if its essential sets are empty – i.e., the rectangle is deleted. If the rectangle is not redundant, it is skipped and processing continues with the next rectangle. Although **rect_irredundant** is itself order-dependent, it at least guarantees that some of the redundant rectangles will be removed. More sophisticated heuristics for irredundant, such as those used in ESPRESSO, can also be applied to the rectangle covering problem.

The procedure *rect_cover* is analogous to a single pass of the *expand*, *irredundant*, *reduce* sequence of Espresso [19]. This operation can be iterated, as done in Espresso, by defining an *expand* procedure to expand each rectangle from an initial covering into a prime rectangle. This is then made irredundant and reduced with the reduced rectangles becoming the input to the first part for reexpansion. Iteration would continue until no decrease in weight is obtained. As in Espresso, this style of heuristic algorithm depends on finding good heuristics for choosing the direction for expansion, and the sequence in which the rectangles are reduced.

One advantage of using covering extraction rather than greedy extraction is that the collection of rectangles in the cover are providing information on the best set of simul-

```
rect_reduce(P) {

    /* Count how many times each point in B is covered */
    M = 0;
    foreach rectangle (R, C) in P {
        foreach row in R {
            foreach column in C {
                M[row][column] = M[row][column] + 1;
            }
        }
    }

    /* Check each rectangle for nonessential parts */
    foreach rectangle (R, C) in P {
        essential_R = 0;
        essential_C = 0;
        foreach row in R {
            foreach column in C {
                if (M[row][column] == 1) {
                    add row to essential_R;
                    add column to essential_C;
                }
            }
        }

        /* modify counts */
        foreach row in (R - essential_R) {
            foreach column in (C - essential_C) {
                M[row][column] = M[row][column] - 1;
            }
        }

        Replace (R,C) with (essential_R, essential_C);
    }
}
```

Figure 3.10: Algorithm **rect_reduce**.

taneous factors to remove from the matrix. Note that each pass of the covering extraction algorithm adds only a controlled number of levels of logic to the network. By solving the minimum weight rectangle-covering problem, it is possible to find a low cost solution which minimizes the increase in circuit depth.

3.8 Selective Collapse

The initial Boolean network has an initial set of common factors already identified. However, there is no guarantee that all of these factors are high-quality. Therefore *selective-collapsing* is performed on the initial Boolean network to provide a better starting point for decomposition. The goal of selective collapse is to remove those factors which provide little value to the current network, while retaining those factors which appear to be of high value.

In the limit, selective collapsing can reduce a network to two-level form. However, for many circuits, this is not a reasonable synthesis technique. Many functions cannot be represented efficiently in two-level form. For example, even a simple function such as comparison of two 32-bit values for equality ($\prod_{i=0}^{32} a_i \oplus b_i$) requires 2^{32} product terms in sum-of-products form. Arithmetic structures, such as n-bit adders, also have an exponential number of product terms (as a function of n) in their minimum two-level form. As another example, consider what happens when a multiplexor is placed in front of a function f and the composite function is collapsed to two-levels. Assume f has p product terms in its minimum two-level form. If n values are multiplexed into a single value, $2^n p$ product terms are needed for the minimum representation of the multiplexed function. This is a simple example where the initial network contains some factors which are valuable for representing the function. Therefore, blindly collapsing a network to two-level form often does not make sense.

Selective-collapsing is implemented by defining a value function for each node in the network. A node of low value is collapsed into all of its fanout. Collapsing one node into another is merely the process of representing the second node without using the first node; i.e., given $f(x_1, \ldots, x_n)$ and $g(f, x_1, \ldots, x_n, y_1, \ldots, y_m)$, determine G such that

$$G(x_1, \ldots, x_n, y_1, \ldots, y_m) = g(f(x_1, \ldots, x_n), y_1, \ldots, y_m).$$

G is uniquely defined, but its representation as a Boolean function is not; hence, a two-level