EXPLICIT AND IMPLICIT ALGORITHMS FOR BINATE COVERING PROBLEMS

by

Tiziano Villa, Timothy Kam, Robert K. Brayton,
and Alberto L. Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M95/108

19 December 1995

cover page

EXPLICIT AND IMPLICIT ALGORITHMS FOR BINATE COVERING PROBLEMS

by

Tiziano Villa, Timothy Kam, Robert K. Brayton,
and Alberto L. Sangiovanni-Vincentelli

Memorandum No. UCB/ERL M95/108

19 December 1995

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Explicit and Implicit Algorithms for Binate Covering Problems

Tiziano Villa[1] Timothy Kam[2] Robert K. Brayton[1]
Alberto L. Sangiovanni-Vincentelli[1]

[1]Department of EECS
University of California at Berkeley
Berkeley, CA 94720

[2]Intel Development Labs
Intel Corporation
Hillsboro, Oregon 97124-6497

December 19, 1995

1 Introduction

At the core of the exact solution of various logic synthesis problems lies often a so-called covering step that requires the choice of a set of elements of minimum cost that *cover* a set of ground items, under certain conditions. Prominent among these problems are the covering steps in the Quine-McCluskey procedure for minimizing logic functions, selection of a minimum number of encoding columns that satisfy a set of encoding constraints, selection of a set of encodeable generalized prime implicants, state minimization of finite state machines, technology mapping and Boolean relations. Let us review first how covering problems are defined formally.

Suppose that a set $S = \{s_1, \ldots, s_n\}$ is given. The cost of selecting s_i is c_i where $c_i \geq 0$. In a general formulation also the cost of not selecting s_i may be non-negative, but here it will be assumed that the cost of not selecting an item is strictly zero, unless otherwise stated. By associating a binary variable x_i to s_i, which is 1 if s_i is selected and 0 otherwise, the binate covering problem (BCP) can be defined as finding $S' \subseteq S$ that minimizes

$$\sum_{i=1}^{n} c_i x_i,$$

subject to the constraint

$$A(x_1, x_2, \ldots, x_n) = 1,$$

where A is a Boolean function, sometimes called the constraint function. The constraint function specifies a set of subsets of S that can be a solution. No structural hypothesis is made on A. Binate refers to the fact that A is in general a binate function [1]. BCP is the problem of finding an onset minterm of A that minimizes

[1]A function is binate if it is not unate. A function $f(x_1, x_2, \ldots, x_n)$ is unate if for every x_i, $i = 1, \ldots, n$, f is either positive or negative unate in the variable x_i. f is said to be positive unate in a variable x_i, if for all 2^{n-1} possible combinations of the remaining $n-1$ variables,

$$f(x_1, x_2, \ldots, x_{i-1}, 1, x_{i+1}, \cdots, x_n) \geq f(x_1, x_2, \ldots, x_{i-1}, 0, x_{i+1}, \cdots, x_n).$$

In other words, changing variable x_i from 0 to 1, f does not decrease. Similarly, f is said to be negative unate in a variable x_i, if the cost function (i.e., a solution of minimum cost of the Boolean equation $A(x_1, x_2, \ldots, x_n) = 1$).

If A is given in product-of-sums form, finding a satisfying assignment is exactly the problem SAT, the prototypical NP-complete problem [20]. In this case it also possible to write A as an array of cubes (that form a matrix M with coefficients from the set $\{0, 1, 2\}$). Each variable of A is a column and each sum (or clause) is a row and the problem can be interpreted as one of finding a subset C of columns of minimum cost, such that for every row r_i, either

1. $\exists j$ such that $a_{ij} = 1$ and $c_j \in C$, or
2. $\exists j$ such that $a_{ij} = 0$ and $c_j \notin C$.

In other words, each clause must be satisfied by setting to 1 a variable appearing in it in the positive phase or by setting to 0 a variable appearing in it in the negative phase. In a unate covering problem, the coefficients of A are restricted to the values 1 and 2 and only the first condition must hold. Here we shall consider the minimum binate covering problem where A is given in product-of-sums form. In this case, the term *covering* is fully justified because one can say that the assignment of a variable to 0 or 1 covers some rows that are satisfied by that choice. The product-of-sums A is called covering matrix or covering table.

As an example of binate covering formulation of a well-known logic synthesis problem, consider the problem of finding the minimum number of prime compatibles that are a minimum closed cover of a given FSM. A binate covering problem can be set up, where each column of the table is a prime compatible and each row is one of the covering or closure clauses of the problem [22]. There are as many covering clauses as states of the original machine and each of them requires that a state is covered by selecting any of the prime compatibles in which it is contained. There are as many closure clauses as prime compatibles and each of them states that if a given prime compatible is selected, then for each implied class in its corresponding class set, one of the prime compatibles containing it must be chosen too. In the matrix representation, table entry (i, j) is 1 or 0 according to the phase of the literal corresponding to prime compatible j in clause i; if such a literal is absent, the entry is 2.

A special case of binate covering problem is a unate covering problem, where no literal in the negative phase is present. Exact two-level minimization [42, 50] can be cast as a unate covering problem. The columns are the prime implicants, the rows are the minterms and there is a 1 entry in the matrix when a prime contains a minterm.

Various techniques have been proposed to solve binate covering problems. A class of them [5, 35] are branch-and-bound techniques that build explicitly the table of the constraints expressed as product-of-sum expressions and explore in the worst-case all possible solutions, but avoid the generation of some of the suboptimal solutions by a clever use of reduction steps and bounding of search space for solutions. We will refer to these methods as explicit.

A second approach [39] formulates the problem with Binary Decision Diagrams (BDD's) and reduces to finding a minimum cost assignment to a shortest path computation. In that case the number of variables of the BDD is the number of columns of the binate table.

Recently, a mixed technique has been proposed in [27]. It is a branch-and-bound algorithm, where the clauses are represented as a conjunction of BDD's. The usage of BDD's leads to an effective method to compute a lower bound on the cost of the solution.

Notice that unate covering is a special case of binate covering. Therefore techniques for the latter solve also the former. In the other direction, exact state minimization, a problem naturally formulated as a binate

for all 2^{n-1} possible combinations of the remaining $n-1$ variables,

$$f(x_1, x_2, \ldots, x_{i-1}, 0, x_{i+1}, \cdots, x_n) \geq f(x_1, x_2, \ldots, x_{i-1}, 1, x_{i+1}, \cdots, x_n).$$

covering problem, can be reduced to a unate covering problem, after the generation of irredundant prime closed sets [52]. But there is a catch here: the cost function is not any more additive, so that the reduction techniques so convenient to solve covering problems, are not any more applicable as they are.

Existing explicit methods do quite well in solving exactly small and medium-sized examples, but fail to complete on larger ones. The reason is that either they cannot build the binate table because the number of rows and columns is too large, or that the branch-and-bound procedure would take too long to complete. The approach of building a BDD of the constraint function and computing the shortest path fails when the number of variables (i.e., columns) is too large because a BDD with many thousands of variables usually cannot be stored in available computing memory.

The crux of the matter, when explicit techniques fail, is that we are representing and manipulating sets that are too large to be exhaustively listed and operated upon. Fortunately we know of an alternative way to represent and manipulate sets: it is by defining the set over an appropriate Boolean space (i.e., encoding the elements of the set), associating to it a Boolean characteristic function and then representing this function by a binary decision diagram (BDD) [6, 1]. From now on, by BDD of a set we will denote the BDD of the characteristic function of the set over an appropriate Boolean space. A BDD [6, 1] is a canonical directed acyclic graph that represents logic functions. The items that a BDD can represent are determined by the number of paths of the BDD, while the size of the BDD is determined by the number of nodes of the DAG. There is no monotonic relation between the size of a BDD and the number of elements that it represents. It is an experimental fact that often very large sets, that cannot be represented explicitly, have a compact BDD representation. Set operations are easily turned into Boolean operations on the corresponding BDD's. So we can manipulate sets by a series of BDD operations (Boolean connectives and quantifications) with a complexity depending on the sizes of the manipulated BDD's and not on the cardinality of the sets that are represented. One hopes that complex set manipulations of a given application have as counterparts Boolean propositions that can be represented with compact BDD's. Of course, this is not always the case and it may happen that an intermediate BDD computation, in a sequence of operations leading to a wanted set, blows up. The name of the game is a careful analysis of how propositional sentences can be transformed into logically equivalent ones, that can be computed more easily with BDD manipulations. Special care must be exercised with quantifications, that bring more danger of BDD explosions. All of this goes often under the name of implicit representations and computations.

The previous insight has already been tested in a series of applications. Research at Bull [10] and UC Berkeley [56] produced powerful techniques for implicit enumeration of subsets of states of a Finite State Machine (FSM). Later work at Bull [12, 37, 24, 18] has shown how implicants, primes and essential primes of a two-valued or multi-valued function can also be computed implicitly. Reported experiments show a suite of examples where all primes could be computed, whereas explicit techniques implemented in ESPRESSO [2] failed to do so. Finally, the fixed-point dominance computation in the covering step of the Quine-McCluskey procedure has been made implicit in [17, 24]. The experiments reported show that the cyclic core of all logic functions of the ESPRESSO benchmark can be successfully computed. For some of them ESPRESSO failed the task.

Last but not the least, it was shown in [31, 32] how all prime compatibles of an FSM can be computed implicitly. In some cases, their number is exponential in the number of states (the largest recorded number is 2^{1500}). Once prime compatibles have been obtained, one must solve a binate covering problem to choose a minimum closed cover. Of course, we cannot build and solve explicitly a table of such dimensions (this would defeat the purpose of computing implicitly prime compatibles in the first place). So it is necessary to extend implicit techniques to the solution of the binate covering problem. Another application of interest to us is the selection of a set of encodeable generalized prime implicants (GPI's), as defined in [19, 39]. It is not feasible to generate GPI's and to set up a related covering table by explicit techniques on non-trivial examples. Using techniques as in [37, 24], GPI's can be generated implicitly. An implicit table solver is therefore needed there too. We will use mainly the two latter applications to illustrate our techniques, but one could list a host of other problems in logic synthesis where a binate table solver would play an important role. Notice that potential applications include unate covering problems, such as selecting the minimum number of encoding-dichotomies that satisfy a set of encoding constraints [51].

We describe an implicit formulation of the binate covering problem and present an implementation. The implicit binate solver has been tested for state minimization of ISFSM's and pseudo NDFSM's [31, 32], and for the selection of an encodable set of GPI's [57]. The reported experiments show that implicit techniques have pushed the frontier of instances where binate covering problems can be solved exactly, resulting in better optimizations in key steps of sequential logic synthesis.

In the following sections, we will review the known algorithms to solve covering problems and then we will describe a new branch-and-bound algorithm based on implicit computations. The remainder is organized as follows. We have defined the minimum cost binate covering problem in this section. In Section 2, we will compare this problem with 0-1 integer linear programming. Solution of binate covering using Thelen-Mathony's algorithm is described in Section 3. The classical solution based on a branch-and-bound scheme will be introduced in Section 4. In Section 5, we will survey the classical reduction rules used in explicit algorithms. Methods to solve binate covering finding a shortest path in a graph-based representation of the clauses are found in Section 6. Our implicit binate covering algorithm is then outlined in Section 7. Section 8 illustrates how reduction techniques can be implicitized. Other kinds of implicit table manipulations are introduced in Section 9. Quantifier-free implicit table reductions are discussed in Section 10. Finally, we shall give experimental results in Section 11, for two applications: state minimization of ISFSM's [22] and selection of generalized prime implicants [19].

2 Relation to 0-1 Integer Linear Programming

There is an intimate relation between 0-1 integer linear programming (ILP) and binate covering problem (BCP). For every instance of ILP, there is an instance of BCP with the same feasible set (i.e., satisfying solutions) and therefore with the same optimum solutions and vice versa. As an example, the integer inequality constraint

$$3x_1 - 2x_2 + 4x_3 \geq 2,$$

with $0 \leq x_1, x_2, x_3 \leq 1$ corresponds to the Boolean equality constraint

$$x_1\overline{x_2} + x_3 = 1,$$

that can be written in product-of-sums form as:

$$(x_1 + x_3)(\overline{x_2} + x_3) = 1.$$

Given a problem instance, it is not clear a-priori which formulation is better. It is an interesting question to characterize the class of problems that can be better formulated and solved with one technique or the other.

As an example of reduction from ILP to BCP, a procedure (taken from [27]) that derives the Boolean expression corresponding to $\sum_{j=1}^{n} w_j.x_j \geq T$ is shown in Figure 1.
The idea of the recursion relies on the observation that:

1. $f = 0$ if and only if $max(I) = \sum_{w_i>0} w_i < T$;
2. $f = 1$ if and only if $min(I) = \sum_{w_i<0} w_i \geq T$;

When neither case occurs, the two subproblems I^1 and I^0, obtained by setting the splitting variable x_i to 1 and 0 respectively, are solved recursively.

```
LI_to_BDD(I) {
    let I be $\sum_{j=1}^{n} w_j \cdot x_j \geq T$
    if $(max(I) < T)$ return 0
    if $(min(I) \geq T)$ return 1
    i = ChooseSplittingVar(I)
    $I^1 = (\sum_{j \neq i} w_j \cdot x_j \geq T - w_i)$
    $I^0 = (\sum_{j \neq i} w_j \cdot x_j \geq T)$
    $f_1$ = LI_to_BDD($I^1$)
    $f_0$ = LI_to_BDD($I^0$)
    return $f = x_i \cdot f^1 + \overline{x_i} \cdot f^0$
}
```

Figure 1: Transformation from linear inequality to Boolean expression.

3 Binate Covering Using Mathony's Algorithm

In [41] Mathony extended an algorithm by Thelen to generate all prime implicants of a Boolean function and he applied it to various problems of two-level logic optimization, like complementation, expansion and reduction of implicants and computation of a minimal cover. A common characteristic of these problems is that finding their solution corresponds to finding only one prime implicant of an appropriate Boolean function represented in conjunctive form (product-of-sums).

A variant of Thelen-Mathony's algorithm that solves the problem of minimum cost satisfiability was implemented into SIS [35]. Minimum cost satisfiability includes binate covering as a special case, because it allows for variables in the negative phase to have cost ≥ 0, while in usual definitions of binate covering one assigns 0 cost to variables in the negative phase, and a cost ≥ 0 to variables in the positive phase.

The algorithm relies on an efficient depth-first multiplication strategy for converting a conjunctive form into the sum of all prime implicants. It chooses recursively one literal from every clause of the function, applying the following pruning rules at every step:

R1: a literal a cannot be chosen if the literal $\overline{a}$ has been chosen;

R2: a literal a cannot be chosen if it appears at an upper level and it has not yet been expanded (i.e., chosen) at that level (because rule **R3** will be applied to this clause when a will be chosen at the upper level);

R3: a clause is not expanded if it contains an already chosen literal (because any choice would generate a non-prime);

R4: a literal a cannot be chosen if:

- it appears at an upper level,
- it has been expanded at that level, and
- rule **R2** was not applied in the subtree of a at the upper level to prune the literal b that is currently being chosen at the upper level (to make sure that its subsuming literal was expanded).

To restrict the computation to find only a minimum cost prime, a fifth rule is added:

R5: a literal a cannot be chosen if the current partial assignment would cost more than the best reached so far.

Experiments on a binate covering application in technology mapping [3] have been reported [35], but no comparison with a traditional binate covering solver is available.

4 A Branch-and-Bound Algorithm for Minimum Cost Binate Covering

We will survey in this section a branch-and-bound solution of minimum cost binate covering. This technique has been described in [23, 22, 4, 5], and implemented in successful computer programs [49, 48, 54]. The branch-and-bound solution of minimum binate covering is based on a recursive procedure. A run of the algorithm can be described by its computation tree. The root of the computation tree is the input of the problem, an edge represents a call to *sm_mincov*, an internal node is a reduced input. A leaf is reached when a complete solution is found or the search is bounded away. From the root to any internal node there is a unique path, that is the current path for that node. In the sequel, we will describe in detail the binary recursion procedure. The presentation will refer to the pseudo-code *sm_mincov*, shown at the end of this subsection.

4.1 Branch-and-Bound as a General Technique

Branch-and-bound constructs a solution of a combinatorial optimization problem by successive partitioning of the solution space. The *branch* refers to this partitioning process; the *bound* refers to lower bounds that are used to construct a proof of optimality without exhaustive search. A set of solutions can be represented by a node in a search tree of solutions, and it is partitioned in mutually exclusive sets. Each subset in the partition is represented by a child of the original node. In this way, a computation tree is built. An algorithm that computes a lower bound on the cost of any solution in a given subset allows to stop further searches from a given node, if the best cost found so far is smaller than the cost of the best solution that can be obtained from the node (lower bound computed at the node). In this case the node is killed and therefore none of its children needs to be searched; otherwise it is alive.

If we can show at any point that the best descendant of a node y is at least as good as the best descendant of node x, then we say that y *dominates* x, and y can kill x.

Figure 2 shows the classical algorithm [47]. An *activeset* holds the live nodes at any point. A variable U is an upper bound on the optimal cost (cost of the best complete solution obtained at any given time). The branching process needs not produce only two children of a given node, but any finite number.

We will see in the next section that BCP can be solved by the following recursive equation

$$BCP(M_f) = BestSolution(BCP(M_{f_{x_i}}) \cup \{x_i\}, BCP(M_{f_{\overline{x_i}}}))$$

where M_f is the binate table that corresponds to a function in product-of-sum form f, and $BCP(M_{f_{x_i}})$ (respectively, $BCP(M_{f_{\overline{x_i}}})$) is the subproblem expressed by the function f_{x_i} (respectively, $f_{\overline{x_i}}$). $BCP(M_f)$ returns an onset minterm of f that minimizes the cost function.

The previous equation can potentially generate an exponential number of subproblems, but powerful dominance and bounding techniques as well as good branching heuristics help in keeping the combinatorial explosion under control.

4.2 The Binary Recursion Procedure

The inputs to the algorithm are:

```
$branch\_and\_bound()$ {
    $activeset$ = original problem
    $U = \infty$
    $currentbest$ = anything
    while ($activeset$ is not empty) {
        choose a branching node $k \in activeset$
        remove node $k$ from $activeset$
        generate the children of node $k$: child $i = 1, \ldots, n_k$
            and the corresponding lower bounds $z_i$
        for $i = 1$ to $n_k$ {
            if ($z_i \geq U$) kill child $i$
            else if (child $i$ is a complete solution) {
                $U = z_i$
                $currentbest$ = child $i$
                else add child $i$ to $activeset$
            }
        }
    }
}
```

Figure 2: Structure of branch-and-bound.

- a covering matrix M;
- a current-path partial solution *select* (initially empty);
- a row of non-negative integers $weight$, whose i-th element is the cost or weight of the i-th column of M;
- a lower bound *lbound* (initially set to 0), which is a monotonic increasing quantity along each path of the computation tree equal to the cost of the partial solution on the current path;
- an upper bound *ubound* (initially set to the sum of weights of all columns in M), which is the cost of the best overall complete solution previously obtained (a globally monotonic decreasing quantity).

The output is the best column cover for input M extended from the partial solution *select* along the current path, called best current solution, if this solution costs less than *ubound*. An empty solution is returned if a solution cannot be found which beats $ubound$ or an infeasibility is detected. By infeasibility, it is meant the case when no satisfying assignment of the product of clauses exists. Even though the initial problem in a typical logic synthesis application has usually at least a solution, some subproblems in the branch and bound tree may be infeasible. When *sm_mincov* is called with an empty partial solution *select* and initial *lbound* and *ubound*, it returns a best global solution.

The algorithm calls first a procedure *sm_reduce* that applies to M essential column detection and dominance reductions. The type of domination operations and the way in which they are applied are the subject of Section 5. Another more complex reduction criterion (Gimpel's rule) can also be applied (see Subsection 5.12). These reduction operations delete from M some rows, columns and entries. What is left after reduction is called a cyclic core. The final goal is to get an empty cyclic core. The value of the lower bound is updated using a maximal independent set computation (see Subsection 4.4). If no bounding is possible and the reductions do not suffice to solve completely the problem, a partition of the reduced problem into disjoint subproblems is attempted (see Subsection 4.3) and each of them is solved recursively. When everything fails, binary recursion is performed by choosing a branch column (see Subsection 4.5). Solutions to the subproblems obtained by including the chosen column in the covering set or by excluding it from the covering set are computed recursively and the best solution is kept (the second recursion is skipped if the solution to the first one matches the updated lower bound).
The procedure *sm_mincov* returns when:

- The cost of a partial solution, found by adding essential columns to *select*, is more than *ubound* or infeasibility is detected when applying the domination rules (line 1). An empty solution is returned.
- The best current solution is found by applying Gimpel's reduction technique (line 2). Since *gimpel_reduce* calls recursively *sm_mincov*, an empty solution could be returned too.
- The updated lower bound, determined by adding to *lbound* the cost of the essential primes and of the maximal independent set, is not less than *ubound* (line 5). An empty solution is returned.
- There is no cyclic core and we are not in the previous case. The best current solution is found by updating *select* with the new essential and unacceptable columns (line 6).
- The best current solution is found by partitioning the problem (line 7). The procedure *sm_mincov* is called recursively on two smaller covering matrices determined by *sm_block_partition* (line 8 and 10). An empty solution can be returned by either recursive call. If the first call to *sm_mincov* returns an empty solution, the second one is not invoked (line 9). If neither call returns empty, each contributes its returned value to the current solution.

```
$sm\_mincov(M, select, weight, lbound, ubound)$ {
    /* Apply row dominance, column dominance, and select essentials */                      (1)
    if (!$sm\_reduce(M, select, weight, ubound)$) return $empty\_solution$
    /* See if Gimpel's reduction technique applies */                                       (2)
    if ($gimpel\_reduce(M, select, weight, lbound, ubound, \&best)$) return $best$
    /* Find lower bound from here to final solution by independent set */                   (3)
    $indep = sm\_maximal\_independent\_set(M, weight)$
    /* Make sure the lower bound is monotonically increasing */                             (4)
    $lbound\_new = max(cost(select) + cost(indep), lbound)$
    /* Bounding based on no better solution possible */                                     (5)
    if ($lbound\_new \geq ubound$) $best = empty\_solution$
    else if ($M$ is empty) { /* New best solution at current level */                       (6)
        $best = solution\_dup(select)$
    } else if ($sm\_block\_partition(M, \&M_1, \&M_2)$ gives non-trivial bi-partitions) {   (7)
        $best1 = sm\_mincov(M_1, select1, weight, 0, ubound - cost(select))$                (8)
        /* Add best solution to the selected set */                                         (9)
        if ($best1 = empty\_solution$) $best = empty\_solution$
        else {                                                                              (10)
            $select = select \cup best1$
            $best = sm\_mincov(M_2, select, weight, lbound\_new, ubound)$
        }
    } else { /* Branch on cyclic core and recur */                                          (11)
        $branch = select\_column(M, weight, indep)$
        $select1 = solution\_dup(select) \cup branch$
        let $M_{branch}$ be the reduced table assuming $branch$ column is not in solution     (12)
        $best1 = sm\_mincov(M_{branch}, select, weight, lbound\_new, ubound)$
        /* Update the upper bound if we found a better solution */                          (13)
        if ($best1 \neq empty\_solution$) and ($ubound > cost(best1)$) $ubound = cost(best1)$
        /* Do not branch if lower bound matched */                                          (14)
        if ($best1 \neq empty\_solution$) and ($cost(best1) = lbound\_new$) return $best1$
        let $M_{\overline{branch}}$ be the reduced table assuming $branch$ column not in solution   (15)
        $best2 = sm\_mincov(M_{\overline{branch}}, select, weight, lbound\_new, ubound)$
        $best = best\_solution(best1, best2)$
    }
    return $best$
}
```

Figure 3: Detailed branch-and-bound algorithm.

- A branching column is chosen and *sm_mincov* is called recursively with the branch column in the covering set (line 12). If the recursive call of *sm_mincov* returns a non-empty solution that matches the current lower bound (*lbound_new*), that solution is returned as the current solution (line 14). If the cost of the current solution is less than *ubound*, *ubound* is updated, i.e., the current solution is also the best global solution (line 13).
- As in the previous case, but *sm_mincov* is called recursively with the branch column not in the covering set (line 15). The best among the solution found in the previous case and the one computed here is the current solution.

Notice the following facts about the procedure *sm_mincov*:

- The parameter *lbound* is updated once (line 4). The reason is that after the computation of the essential columns (line 1) and of the independent set (line 3), the cost of the previous partial solution summed to the cost of the essential columns and of the independent set is potentially a sharper lower bound on any complete solution obtained from this node of the recursion tree. The updated value *lbound_new* is used in the rest of the routine. The lower bound is a monotonically increasing quantity along each path of the computation tree.
- The parameter *ubound* is updated once (line 13). At that point a new complete solution has just been returned by the recursive call to *sm_mincov* (line 12) and an updated value of *ubound* must be recomputed for the following recursive call of *sm_mincov* (line 15). The reason is that when a new complete solution is obtained, the current *ubound* is not any more valid and therefore it must be updated before it is used again. To be updated, *ubound* is compared against the cost of the newly found solution, and the minimum of the two is the new *ubound*. The upper bound is a monotonically decreasing quantity throughout the entire computation.

The previous analysis proves that the algorithm finds a minimum cost satisfying assignment to the problem.

4.3 N-way Partitioning

If the covering matrix M can be partitioned into two disjoint blocks M_1 and M_2, the covering problem can be reduced to two independent covering subproblems, and the minimum covering for M is the union of the minimum coverings for M_1 and M_2. Such bi-partition can be found by putting in M_1 a row and all columns that have an element in common with the row (i.e., the columns intersecting the row) and recursively all rows and columns intersecting any row or column in M_1. The remaining rows and columns (i.e., not intersecting any row or column in M_1) are put in M_2. This algorithm can be generalized to find partitions made by N blocks, as shown in Figure 4.

Theorem 4.1 *If a covering matrix M can be partitioned into n disjoint blocks $M_1, M_2, \ldots, M_n$, the union of the minimum covers of $M_1, M_2, \ldots, M_n$ is the minimum cover of M.*

Bi-partitioning is implemented in [48, 54] as follows. When checking for a partition of the problem (line 7), the routine *sm_mincov* is called recursively on two independents subproblems (lines 8 and 10), if they exist. When solving the smaller of the two subproblems (line 8), the initial solution is empty, the initial lower bound is set to 0, the initial upper bound is set to the difference between the current *ubound* and the cost of the current partial solution. When solving the larger of the two subproblems (line 10), the initial solution is the current solution (to which the solution of the smaller subproblem is added, if it is not empty), the initial lower bound is set to the current lower bound *lbound_new*, the initial upper bound is set to the current *ubound*.

```
$n\_way\_partition(M)$ {
    while (there is a row $r_i$ not in any partition) {
        put $r_i$ in a new partition $M_k$
        while (there is a row $r_j$ connected to any row in partition $M_k$ ) {
            put row $r_j$ in partition $M_k$
        }
    }
}
```

Figure 4: N-way partitioning.

Theorem 4.2 *The upper bound set in the smaller subproblem is correct.*

Proof. Let *select* be the partial solution along the current path. It holds that (cost of the final solution along the current path) $\geq$ (cost of solving $M_1 + cost(select) + 1$). If (cost of solving M_1) $\geq$ $(ubound - cost(select))$, then (cost of the final solution along the current path) $\geq$ $(ubound + 1)$, i.e., (cost of the final solution along the current path) $>$ $ubound$. This is ruled out by setting the upper bound when solving M_1 to $(ubound - cost(select))$, since *sm_mincov* returns a non-empty solution only if it can beat the given upper bound. □

4.4 Maximal Independent Set

The cardinality of a maximum set of pairwise disjoint rows of M (i.e., no 1's in the same column) is a lower bound on the cardinality of the solution to the covering problem, because a different element must be selected for each of the independent rows in order to cover them. If the size of current solution plus the size of the independent set is greater or equal to the best solution seen so far, the search along this branch can be terminated because no solution better than the current one can possibly be found. It is also true that the size of the independent set at the first level of the recursion is a lower bound for the final minimum cover, so that the search can be terminated if a solution is found of size equal to this lower bound. Since finding a maximum independent set is an NP-complete problem, in practice an heuristic is used that provides a weaker lower bound. Notice that even the lower bound provided by solving exactly maximum independent set is not sharp. In [9] it is shown an example of size $O(n^2)$, whose minimal solution has a $O(n)$ cost, but whose lower bound by independent set is a constant 1. In practice a lower bound by independent set is poor when the covering matrix is dense.

In [49, 48, 54], the adjacency matrix B of a graph whose nodes correspond to rows in the cover matrix M is created. In the binate case, only rows are taken into consideration which do not contain any 0 element. An edge is placed between two nodes if the two rows have an element in common. While B is non-empty, a row R_i of B is found that is disjoint from a maximum number of rows (i.e., the row of minimum length in B). The column of minimum weight intersecting R_i is also found. The weight is cumulated in the independent set cost. All rows having elements in common with R_i are then deleted from B. At the end of the *while*-iteration a set of pairwise disjoint rows (independent set) and their minimum covering cost is found. Notice that one could think to the problem in a dual way as finding a maximal clique in a graph with the same rows as before, and edges between two nodes representing two disjoint rows.

In [9] some detailed analysis of independent set computations is made. A quantitative ratio between a maximal cost independent set and the independent set computed by a greedy algorithm based on set-packing is derived. A logarithmic ratio lower bound on unate problems is proved too.

4.5 Selection of a Branching Column

The selection of a good branching column is essential for the efficiency of the branch and bound algorithm. Since the time taken by the selection is a significant part of the total, a trade-off must be made between quality and efficiency.

In [49, 48, 54], the selection of the branching variable is restricted to columns intersecting the rows of the independent set, because a unique column must eventually be selected from each row of the maximal independent set. Among those rows, the selection strategy favors columns with large number of 1's and intersecting many short rows. Short rows are considered difficult rows and choosing them first favors the creation of essential columns. More precisely, the column of highest merit is chosen. The merit of a given column is computed as the product of the inverse of the weight of the column multiplied by the sum of the contributions of all rows intersected in a 1 by the column. The inverse of the contribution of a row is equal to the number of all non-2 elements (each can contribute in covering the row) minus 1. The inverse is well-defined, because at this stage each row has at least two-elements (it is not essential).

4.6 New Bounding Criteria

In [16] two new rules to prune the search space have been introduced. We are going to survey them here. Given a covering problem C that corresponds to a node c of the computation tree, define the following notation:

- C_l is the subproblem of C generated assuming that a given branching column b is selected;
- C_r is the subproblem of C generated assuming that a given branching column b is not selected;
- $C.min$ is the cost of a minimum solution;
- $C.lower$ is the value of a lower bound on $C.min$;
- $C.path$ is the cost of the partial solution from the root to node c;
- $C.upper$ is the cost of the best solution found so far.

The algorithm described in Figure 3 guarantees that the invariant $C.path + C.lower < C.upper$ is always true.

Theorem 4.3 (**Left-hand side lower bound**). *Given a binate covering problem C, suppose to branch on a unate column b. If*

$$C.path + C_l.lower \geq C.upper,$$

then both C_l and C_r can be pruned and $C_l.lower$ is a strictly better lower bound for C.

Proof. In C_l it holds (using the hypothesis):

$$C_l.path + C_l.lower = C.path + Cost(b) + C_l.lower \geq C.path + C_l.lower \geq C.upper,$$

so C_l is correctly pruned to keep the invariant $C_l.path + C_l.lower < C.upper$.

Let us see why also C_r can be pruned. First notice that $C_r.min \geq C_l.min$, since C_r has exactly the same columns than C_l, but it has more rows to cover (those covered by choosing b in the solution). Then the best global solution that can be found by solving C_r exceeds the upper bound, as shown by the following chain of inequalities (using the hypothesis):

$$C_r.path + C_r.min = C.path + C_r.min \geq C.path + C_l.min \geq C.path + C_l.lower \geq C.upper.$$

We show now that $C_l.lower$ is a lower bound on $C.min$. From $C.path + C_r.min \geq C.path + C_l.lower$, it follows $C_r.min \geq C_l.lower$; it is also $C_l.min \geq C_l.lower$. So it follows ($Cost(b) \geq 0$):

$$C.min = min(Cost(b) + C_l.min, C_r.min) \geq min(Cost(b) + C_l.lower, C_l.lower) = C_l.lower.$$

Lastly we show that $C_l.lower > C.lower$. By contradiction if $C.lower \geq C_l.lower$, then (using the hypothesis):

$$C.path + C.lower \geq C.path + C_l.lower \geq C.upper,$$

against the invariant $C.path + C.lower < C.upper$. □

The way in which the "old" lower bound and the "new" left-hand side lower bound work together is: if the current node is a left child and $lbound\_new - Cost(b) \geq ubound$ then bound computation and return flag to skip also the right branch ("new" left-hand side lower bound); otherwise if $lbound\_new \geq ubound$ then bound computation ("old" lower bound).

Theorem 4.4 **(Limit lower bound)**. *Given a binate covering problem C, let I be an independent set of the rows, i.e., a set of unate rows intersecting no common column. Let $C.lower$ be a lower bound from the independent set I, i.e., the sum of a minimum cost column for each row in I. Consider the set B of the columns b that do not intersect rows in I and such that $b \in B$ only if*

$$C.path + C.lower + Cost(b) \geq C.upper.$$

Then the columns in B and the rows that intersect them in a 0 can be removed from the covering table and a minimum solution can still be found.

Proof. If we choose a column b in B as a branching column, we obtain a subproblem C_l by assuming that b is in the solution. I is still an independent set of C_l, because by construction b does not intersect rows in I. So a lower bound for C_l has at least value $C.lower$ (but there could be a lower bound by independent set for C_l larger than $C.lower$), that is

$$C_l.lower \geq C.lower.$$

The following chain of implications follows (using the hypothesis):

$$C_l.path + C_l.min = C.path + Cost(b) + C_l.min \geq C.path + Cost(b) + C_l.lower$$

$$\geq C.path + Cost(b) + C.lower \geq C.upper,$$

meaning that the best solution that can be found by solving C_l exceeds the global upper bound. Therefore we can set to 0 the columns $b \in B$ and still get a minimum solution. □

In practice in the common case that all columns have cost 1 if included in a solution, one needs only to check whether

$$C.path + C.lower + 1 \geq C.upper,$$

i.e.,

$$lbound\_new + 1 \geq C.upper,$$

in which case all columns that do not intersect rows in the independent set I can be removed, together with the rows that they intersect in a 0. Experimental results in [16] on exact two-level minimization show strong gains by this new pruning technique, resulting in reductions of the search space up to three orders of magnitude.

4.7 Symmetric Covering Problems

In [58] symmetric unate covering problems, especially those arising from two-level logic minimization, are investigated. Given a unate covering problem P whose variables (columns) are $(x_1, x_2, \cdots, x_n)$, a permutation of $(x_1, x_2, \cdots, x_n)$ into $(\eta(x_1), \eta(x_2), \cdots, \eta(x_n))$ is a symmetric permutation of P if $(\eta(x_1), \eta(x_2), \cdots, \eta(x_n))$ is a feasible solution of P when $(x_1, x_2, \cdots, x_n)$ is a feasible solution of P. Both feasible solutions yield the same value of the objective function. P is said to be symmetric if it has some symmetric permutations. When a minimal unate covering problem has symmetric permutations, the Boolean function from which it derives may not be symmetric; vice versa, if a given Boolean function is symmetric, the minimal covering problem obtained from it is symmetric. In the paper, after a complete characterization of symmetric permutations, it is shown how to exploit symmetry to speed up the number of branchings required to certify that a solution is optimal. In particular the preservation of symmetric permutations under row dominance, column dominance and detection of essential columns is investigated; procedures are presented such that their repeated application to a problem P with a symmetric permutation η yields a problem P', reduced with respect to row dominance, column dominance and detection of essential columns, and still symmetric with respect to a permutation η' obtained from η in a given way.

5 Reduction Techniques

Three fundamental processes constitute the essence of the reduction rules:

1. Selection of a column: a column must be selected if it is the only column that satisfies a required constraint (Section 5.7). A dual statement holds for unacceptable columns (Section 5.8). Also related is the case of unnecessary columns (Section 5.9).

2. Elimination of a column: a column C_i can be eliminated, if its elimination does not preclude obtaining a minimal cover, i.e., if there exists in M another column C_j that satisfies at least all the constraints satisfied by C_i (Section 5.5).

3. Elimination of a row: a row R_i can be eliminated if there exists in M another row R_j that expresses the same or a stronger constraint (Section 5.1).

Even though more complex criteria of dominance have been investigated (for instance, Section 5.12), the previous ones are basic in any table covering solver. Reduction rules have previously been stated for the binate covering case [22, 23, 5, 4], and also for the unate covering case [42, 50, 4]. Here we will present the known reduction rules directly for binate covering and indicate how they simplify for unate covering, when applicable. For each of them, we will first define the reduction rule, and then a theorem showing how that rule is applied. Proofs for the correctness of these reduction rules have been given in [22, 23, 5, 4], and they will not be repeated here, except for a few less common ones. We will provide a survey comparing different related reduction rules used in the literature.

```
sm_reduce($A$, $solution$, $weight$, $ubound$) {
    do {
        apply $\beta$-dominance or $\alpha$-dominance
        find essential columns
        find unacceptable columns
        if (a column is both essential and unacceptable)
            return $empty\_solution$
        for each essential column {
            delete each row intersecting the column in a 1
            if (a row of length 1 intersects the column in a 0)
                return $empty\_solution$
            delete column
            add column to $solution$
            if (cost of $solution \geq ubound$)
                return $empty\_solution$
        }
        for each unacceptable column {
            delete each row intersecting the column in a 0
            if (a row of length 1 intersects the column in a 1)
                return $empty\_solution$
            delete column
        }
        apply row_consensus
        apply row_dominance
    } while (reductions are applicable)
    return $solution$
}
```

Figure 5: Flow of reduction rules.

The effect of reductions depends on the order of their application. Reductions are usually attempted in a given order, until nothing changes any more (i.e., the covering matrix has been reduced to a cyclic core). Figure 5 shows how reductions are applied in [49, 48, 54][2].

5.1 Row Dominance

Definition 5.1 *A row R_i dominates [3] another row R_j if R_j has all the 1's and 0's of R_i.*

[2]The reductions *β-dominance* and *row_consensus* are only in [48] and the reduction by implication is only in [54].

[3]This definition of row dominance is

- similar to column dominance (Rule 3) in [22], except that the labels of dominator row, R_i, and dominated row, R_j, are reversed (i.e., reverse definition of dominance),
- similar to column dominance (Rule 3) in [23], except that the labels of dominator row, R_i, and dominated row, R_j, are reversed (i.e., reverse definition of dominance),
- equivalent to row dominance (Definition 10) in [5],

Theorem 5.1 *If a row R_j is dominated by another row R_i, R_j can be eliminated without affecting the solutions to the covering problem.*

5.1.1 Row Dominance for a Unate Table

Definition 5.2 *A row R_i dominates another row R_j if R_j has all the 1's of R_i.*

5.2 Row Consensus

Theorem 5.2 *If R_i dominates R_j, except for a (unique) column C_k where R_i and R_j have different values, element $M_{j,k}$ can be eliminated from the matrix M (i.e., the entry in position $M_{j,k}$ becomes a 2) without affecting the solutions of the covering problem.*

Proof. Suppose that entry $M_{j,k}$ is 1 and entry $M_{i,k}$ is 0. The argument is the same if entry $M_{j,k}$ is 0 and entry $M_{i,k}$ is 1. If entry $M_{j,k}$ is removed, we are not able to satisfy row R_j by setting x_k to 1. A problem arises if a minimum-cost solution requires x_k set to 1, because we could miss the fact that setting x_k to 1 satisfies also row R_j. Instead we could obtain an higher-cost solution, by selecting another column in order to satisfy row $R_j - M_{j,k}$. We now show that this is not the case. If a minimum-cost solution requires x_k set to 1, we must still satisfy row R_i that cannot be satisfied by x_k set to 1. Whatever choice will be made to satisfy R_i, it will satisfy also $R_j - M_{j,k}$ (since $R_j - M_{j,k}$ has all 1's and 0's of R_i) and therefore no more cost will be incurred to satisfy row $R_j - M_{j,k}$. The previous argument fails if $R_j - M_{j,k}$ is empty and there are cases in which an higher-cost solution would be found. One could claim that if $R_j - M_{j,k}$ is empty, then R_j has only entry $M_{j,k}$ and therefore x_k is an essential, that is taken care by the essential column detection. In reality it may happen that by applying row consensus many times to the same row R_j (using different rows R_i) at a certain point R_j is emptied. In that case the last application of row consensus is potentially faulty and should not be done. □

Row consensus is applied in [48]. This criterion generalizes the one given in [25].

5.3 Column α-Dominance

Definition 5.3 *A column C_j α-dominates [4] another column C_k if:*

- $c_j \leq c_k$,
- *C_j has all the 1's of C_k,*
- *C_k has all the 0's of C_j.*

Theorem 5.3 *Let M be satisfiable. If a column C_k is α-dominated by another column C_j, there is at least one minimum cost solution with column C_k eliminated ($x_k = 0$), together with all the rows in which it has 0's.*

- identical to row dominance (Definition 2.11) in [4].

[4]This definition of column α-dominance is

- an extension to row α-dominance (Rule 1) in [22], because the latter doesn't include the case $M_{i,j} = 0$ and $M_{i,k} = 0$,
- equivalent to first half of Rule 4 in [23]: (a) C_j has all the 1's of C_k and (b1) C_k has all the 0's of C_j,
- identical to column dominance (Definition 11, Theorem 3) in [5],
- identical to column dominance (Definition 2.12, Theorem 2.4.1) in [4].

In [9] column dominance is formulated in a more general way as follows.

Theorem 5.4 *Suppose that v and v' are elements of $\{0,1\}$. If the clauses satisfied by column C_y set to the value v are satisfied at a lower cost by setting column $C_{y'}$ to v', and the clauses satisfied by $C_{y'}$ set to $\overline{v'}$ are also satisfied at zero cost by C_y set to $\overline{v}$, one can set C_y to $\overline{v}$ and remove the rows that intersect C_y in $\overline{v}$, without missing any optimal solution.*

Proof. Setting C_y to $\overline{v}$ does not lead to a suboptimal solution, because there is another column $C_{y'}$ that, if set to v', covers for less the rows that are left uncovered by setting C_y to $\overline{v}$, while setting C_y to $\overline{v}$ covers already all rows that would be left uncovered if one would have to set C_y to v. □

If negative literals have non-zero cost and positive literals have positive cost, it is exactly the definition of α-dominance.

5.3.1 Column Dominance for a Unate Table

Definition 5.4 *A column C_j dominates another column C_k if C_j has all the 1's of C_k,*

5.4 Column β-Dominance

Definition 5.5 *A column C_i β-dominates [5] another column C_j if:*

- $c_i \leq c_j$,
- *C_i has all the 1's of C_j,*
- *for every row R_p in which C_i has a 0, either C_j has a 0 or there exists a row R_q in which C_j has a 0 and C_i does not have a 0, such that disregarding entries in columns C_i and C_j, R_q dominates R_p.*

Theorem 5.5 *Let M be satisfiable. If C_i β-dominates C_j, there is at least one minimum cost solution with column C_j eliminated ($x_j = 0$), together with all the rows in which it has 0's.*

Proof. We must show that given a solution, one can find another solution, of cost lesser or equal, with column C_j eliminated ($x_j = 0$). There are two cases for the original solution: either $x_i = 1$ and $x_j = 1$ or $x_i = 0$ and $x_j = 1$ (if $x_j = 0$, we are done). The new solution has $x_i = 1$ and $x_j = 0$ and coincides for the rest with the given solution. The case when $x_i = 1$ and $x_j = 1$ is easy, because column C_i has all 1's of column C_i and therefore C_j is useless.

Consider now the case when $x_i = 0$ and $x_j = 1$. The clauses with a 0 in column C_i are satisfied by not choosing C_i and the clauses with a 1 in column C_j are satisfied by choosing C_j. Each clause with a 0 in column C_j (and without a 0 in column C_i) is satisfied by a proper assignment of a column different from C_i and C_j, say C_k. Notice that the hypothesis that column C_i does not have a 0 in the

[5]This definition of column β-dominance is

- strictly stronger than column α-dominance given in 5.3,
- more general than row β-dominance (Rule 5) in [22], because the latter assumes that the covering table contains only rows with no or one 0,
- equivalent to second half of Rule 4 in [23]: (a) C_i has all the 1's of C_j and (b2) for every row R_p in which C_i has a 0, there exists a row R_q in which C_j has a 0, such that disregarding entries in row C_i and C_j, R_p dominates R_q (with reverse definition of row dominance), noticing that by mistake the condition that C_i does not have a 0 in row R_q was omitted,
- not mentioned in [5] and [4].

clause is essential here, otherwise this clause would be satisfied already by not choosing C_i, without resorting to a column C_k. Now consider the assignment with column C_i and without column C_j ($x_i = 1$ and $x_j = 0$) and the same remaining assignments as the previous one. It costs no more than the previous one. We show that it is a solution. In order to do that we must make sure that the 0's covered by C_i and the 1's covered by C_j by setting $x_i = 0$ and $x_j = 1$, are still covered in the new assignment where $x_i = 1$ and $x_j = 0$. The clauses with a 1 in C_j are satisfied by C_i, because C_i has all 1's of C_j. Each clause, say R_p, with a 0 in column C_i is satisfied too, because there is a corresponding clause, say R_q, with a 0 in column C_j, and we already noticed that there exists another column, C_k, that satisfies R_q. But by hypothesis R_q dominates R_p, i.e., R_p has all the 1's and 0's of R_q, hence column C_k satisfies also clause R_p (if entry $M_{q,k} = 1(0)$, then entry $M_{p,k} = 1(0)$ also and $x_k = 1$ ($x_k = 0$) satisfies both clauses). □

5.5 Column Dominance

Definition 5.6 *A column C_i dominates another column C_j if either C_i α-dominates C_j or C_i β-dominates C_j.*

Theorem 5.6 *Let M be satisfiable. If C_i dominates C_j, there is at least one minimum cost solution with column C_j eliminated ($x_j = 0$), together with all the rows in which it has 0's.*

5.6 Column Mutual Dominance

Definition 5.7 *Two columns C_i and C_j mutually dominate* [6] *each other if:*

- *C_i has a 0 in every row where C_j has a 1,*
- *C_j has a 0 in every row where C_i has a 1.*

Theorem 5.7 *Let M be satisfiable. If C_i and C_j mutually dominate each other, there is at least one minimum cost solution with columns C_i and C_j eliminated ($x_i = x_j = 0$), together with all the rows in which they have 0's.*

In [9] column mutual dominance is formulated in a more general way as follows.

Theorem 5.8 *Suppose that v and v' are elements of $\{0,1\}$. Suppose that column C_y has minimum cost when set to v and column $C_{y'}$ has minimum cost when set to v'. If the clauses satisfied by setting column $C_{y'}$ to $\overline{v'}$ are satisfied by setting column C_y to v, and the clauses satisfied by setting C_y to $\overline{v}$ are satisfied by setting $C_{y'}$ to v', then one can set C_y to v, $C_{y'}$ to v' and remove the rows that intersect C_y in v and $C_{y'}$ to v', without missing any optimal solution.*

Proof. Column C_y set to v covers the rows otherwise covered by $C_{y'}$ set to $\overline{v'}$ and $C_{y'}$ set to v' covers the rows otherwise covered by C_y set to $\overline{v}$. Therefore setting C_y to v, $C_{y'}$ to v' is always better than any of the other three combinations, given that C_y has minimum cost at v and $C_{y'}$ has minimum cost at v'. □

If negative literals have non-zero cost and positive literals have positive cost, it is exactly the definition of column mutual dominance.

[6]This definition of column mutual dominance is

- identical to rule for mutually reducible variables in [53],
- not mentioned in other papers.

5.7 Essential Column

Definition 5.8 *A column C_j is an essential column* [7] *if there exists a row R_i having a* 1 *in column C_j and* 2*'s everywhere else.*

Theorem 5.9 *If C_j is an essential column, it must be selected ($x_j = 1$) in every solutions. Column C_j must then be deleted together with all the rows in which it has* 1*'s.*

5.7.1 Essential Column for a Unate Table

Definition 5.9 *A column is an essential column if it contains the* 1 *of a singleton row.*

5.8 Unacceptable Column

Definition 5.10 *A column C_j is an unacceptable column* [8] *if there exists a row R_i having a* 0 *in column C_j and* 2*'s everywhere else.*

This reduction rule is a dual of the essential column rule.

Theorem 5.10 *If C_j is an unacceptable column, it must be eliminated ($x_j = 0$) in every solution, together with all the rows in which it has* 0*'s.*

5.9 Unnecessary Column

Definition 5.11 *A column of only* 0*'s and* 2*'s is an unnecessary column* [9].

Notice that there is no symmetric rule for columns of 1's and 2's. The reason is that selecting a column to be in the solution has a cost, while eliminating it has no cost.

Theorem 5.11 *If C_j is an unnecessary column, it may be eliminated ($x_j = 0$), together with all the rows in which it has* 0*'s.*

[7] This definition of essential column is

- identical to essential row (Rule 2) in [22],
- identical to Rule 1 in [23],
- included in Definition 9 in [5]: the row R_i in the above definition corresponds to a singleton-1 essential row in [5],
- included in Definition 2.10 in [4]: the row R_i in the above definition corresponds to a singleton-1 essential row in [4].

[8] This definition of unacceptable column is

- identical to that of nonselectionable row in [22],
- identical to Rule 2 in [23],
- included in Definition 9 in [5]: the row R_i in the above definition corresponds to a singleton-0 essential row in [5],
- included in Definition 2.10 in [4]: the row R_i in the above definition corresponds to a singleton-0 essential row in [4].

[9] This definition of unnecessary column is

- identical to Rule 4 in [22],
- identical to Rule 5 in [23],
- not mentioned in [5] and [4].

5.10 Trial Rule

Theorem 5.12 *If there exists in a covering table M a row R_i having a 0 in column C_j, a 1 in column C_k and 2's in the rest, then apply the following test:*

- *eliminate C_k together with the rows in which it has 0's,*
- *eliminate C_j, which is now an unacceptable column, together with the rows in which it has 0's,*
- *continue as long as possible to eliminate the columns which becomes unacceptable columns.*

If at least one row of M has only 2's at the end of this test, then column C_k must be selected ($x_k = 1$)[10]*. Therefore, C_k can be deleted together with all the columns in which it has 1's* [11].

5.11 Infeasible Subproblem

Unlike the unate covering problem, the binate covering problem may be infeasible. In particular, an intermediate covering matrix M may found to be unsatisfiable by the following theorem. When an infeasible subproblem is found, that branch of the binary recursion is pruned.

Definition 5.12 *A covering problem M is infeasible* [12] *if there exists a column C_j which is both essential and unacceptable (implying $x_j = 1$ and $x_j = 0$).*

5.12 Gimpel's Reduction Step

Another heuristic for solving the minimum cover problem has been suggested by Gimpel [21]. Gimpel proposed a reduction step which simplifies the covering matrix when it has a special form. This simplification is possible without further branching, and hence is useful at each step of the branch and bound algorithm. In practice, Gimpel's reduction step is applied after reducing the covering matrix to the cyclic core.

Gimpel's reduction can be described in terms of the product-of-sums represented by a covering table. The product-of-sums is examined to see if any clause has only two literals of the same cost. For example, assume the expression has the form:

$$p = R(c_1 + c_2)(c_1 + S_1)\dots(c_1 + S_n)(c_2 + T_1)\dots(c_2 + T_m)$$

where c_1 and c_2 are single variables with a cost C, $S_i, i = 1\dots n$ and $T_j, j = 1\dots m$ are sums of variables not containing c_1 or c_2, and R is a product of sums of variables not containing c_1 or c_2. Because the covering table is assumed minimal, if there is a clause $(c_1 + c_2)$, then $m \geq 1$, $n \geq 1$, and none of S_i or T_j is identically zero.

[10]It is possible that a row is left with only 2's by a sequence of reduction steps.

[11]This reduction rule is

- identical to Rule 6 in [22],
- not mentioned in other papers.

[12]This definition of infeasibility is

- not mentioned in [22] and [23],
- briefly mentioned in [5],
- identical to the unfeasible problem in [4].

Note that with the expression written in this form, each parenthesized expression corresponds directly to a single row in the covering table. By algebraic manipulations, the expression can be re-written as:

$$p = R(c_1c_2 + c_1T + c_2S)$$

where $S = \prod_{i=1}^{n} S_i$, and $T = \prod_{i=1}^{m} T_i$.

A second covering problem is derived from the original covering problem with the following form:

$$\begin{aligned} p_1 &= R(c_2 + S + T) \\ &= R\prod_{i=1}^{n}\prod_{j=1}^{m}(c_2 + S_i + T_j) \end{aligned}$$

The main theorem of Gimpel is:

Theorem 5.13 *Let M_1 be a minimum cover for p_1. A cover for p can be derived from M_1 according to the rule:* if S is covered by M_1 then add c_2 to M_1 to derive a cover of p; otherwise, add c_1 to M_1 to derive a cover of p. *The resulting cover is a minimum cover for p.*

A proof can be found in [50], where a more extended discussion is presented.

Gimpel's reduction step was originally stated for covering problems where each column had cost 1. Robinson and House [26] showed that the reduction remains valid even for weighted covering problems if the cost of the column c_1 equals the cost of the column c_2, as it has been presented here. Gimpel's rule has been first proposed in [21] and then implemented in [49]. In [48, 54] Gimpel's rule has been extended to handle the binate case. This extension has been described in [55].

6 Semi-Implicit Solution of Binate Covering

6.1 Binary Decision Diagrams

Basic introductions to binary decision diagrams are found in [6, 1].

Definition 6.1 *A* **binary decision diagram** *(BDD) is a rooted, directed acyclic graph. Each nonterminal vertex v is labeled by a Boolean variable $var(v)$. Vertex v has two outgoing arcs, $child_0(v)$ and $child_1(v)$. Each terminal vertex u is labeled 0 or 1.*

Each vertex in a BDD represents a binary input binary output function and all accessible vertices are roots. The terminal vertices represent the constants (functions) 0 and 1. For each nonterminal vertex v representing a function F, its child vertex $child_0(v)$ represents the function $F_{\overline{v}}$ and its other child vertex $child_1(v)$ represents the function F_v. i.e., $F = \overline{v} \cdot F_{\overline{v}} + v \cdot F_v$.

For a given assignment to the variables, the value yielded by the function is determined by tracing a decision path from the root to a terminal vertex, following the branches indicated by the values assigned to the variables. The function value is then given by the terminal vertex label.

Definition 6.2 *A BDD is* **ordered** *if there is a total order $\prec$ over the set of variables such that for every nonterminal vertex v, $var(v) \prec var(child_0(v))$ if $child_0(v)$ is nonterminal, and $var(v) \prec var(child_1(v))$ if $child_1(v)$ is nonterminal.*

Definition 6.3 *A BDD is* **reduced** *if*

1. it contains no vertex v such that $child_0(v) = child_1(v)$, and

2. *it does not contain two distinct vertices v and v' such that the subgraphs rooted at v and v' are isomorphic.*

Definition 6.4 *A* **reduced ordered binary decision diagram** *(ROBDD) is a BDD which is both reduced and ordered.*

Definition 6.5 *The ITE operator returns function G_1 if function F evaluates true, else it returns function G_2:*

$$ITE(F, G_1, G_0) = \begin{cases} G_1 & \text{if } F = 1 \\ G_0 & \text{otherwise} \end{cases}$$

where range(F)={0,1}.

6.2 The Shortest Path Method

In [40] the solution of a binate covering problem was reduced to a shortest path computation on the BDD representing the clauses. We will present the theorem supporting the reduction.

Suppose that the length (or cost) of a 0-edge of a BDD is 0 and the length of a 1-edge is a positive constant. A shortest path between two nodes is a path of total minimum length.

Theorem 6.1 *A minimum cost assignment satisfying a Boolean formula $T(x_1, \cdots, x_n)$ is given by a shortest path from the root to the terminal 1 of a ROBDD representing T.*

Proof. For every path p from the root to the terminal 1 there is a set of associated variable assignments $\{\mathbf{x}\}_p$ that satisfy T. An assignment $\mathbf{x}_p$ is in $\{\mathbf{x}\}_p$, only if variable x_i is set to 1(0) when x_i and its 1-edge (0-edge) appear in p. For any combination of assignments to the variables that do not appear in p there is a corresponding complete $\mathbf{x}_p$ and varying all such combinations one spans all assignments $\mathbf{x}_p \in \{\mathbf{x}\}_p$. The unique assignment $\mathbf{x}_p \in \{\mathbf{x}\}_p$ such that the variables not in p are set to 0 has the property that its cost is equal to the length of path p. Call it $\mathbf{x}_{p_{min}}$. Vice versa given a satisfying assignment $\mathbf{x}$ there is a unique path p from the the root to the terminal 1 whose associated set $\{\mathbf{x}\}_p$ includes $\mathbf{x}$. The length of p is less or equal to the cost of $\mathbf{x}$.

Given a shortest path p in the ROBDD, consider the corresponding minimum cost assignment $\mathbf{x}_{p_{min}}$. Suppose by contradiction that there is another satisfying assignment $\mathbf{x}'$ of smaller cost. Consider the unique path p' such that $\{\mathbf{x}'\}_{p'}$ includes $\mathbf{x}'$. The length of p' is less or equal to the cost of $\mathbf{x}'$, but the cost of $\mathbf{x}'$ is less than the cost of $\mathbf{x}_{p_{min}}$, that coincides with the length of p, and so we found a path p' of length strictly less than the length of p, against the hypothesis that p is a shortest path. □

6.3 The Method Based on a Product of BDD's

In [27, 28], a branch-and-bound algorithm for the binate covering problem expressed as a product of general boolean formulas and represented by a conjunction of multiple BDD's is presented. This is in alternative to the case when the constraints are expressed as a product-of-sums (POS) and represented by a matrix where each row is a clause and each column is a variable. The attractive feature of a BDD-based algorithm is that finding the solution only requires computing the shortest path to the 1 terminal in the BDD. Since in cases of practical interest, it happens often that a single BDD representing all clauses is too large to be built, it has been proposed to represent the constraints as a product of sub-constraints, each of which can be represented by a BDD. The question is how to find a minimum solution, having a product of BDD's, instead than a single BDD. It is clear that if each subconstraint is a sum-of-products (SOP) clause, the BDD-based formulation is analogous to the one based on a matrix. This motivates the extension to a conjunction of BDD's of the reduction and bounding techniques devised to solve a table.

The algorithm assumes that the constraint function is in the form $f = \prod_{i=1}^{n} f_i$ where each f_i is represented by a BDD F_i. Each f_i or F_i is called a sub-constraint. The conjunction of the F_i is called F. Under this assumption, BCP amounts to finding an assignment for $x_1, x_2, \ldots, x_n$ that minimizes the cost function and that satisfies all f_i's simultaneously. If $n = 1$, we have a single BDD and the minimum cost assignment that satisfies f can be found by computing the shortest path connecting the root of f to the '1' leaf. If $n > 1$ a branch-and-bound algorithm as in the matrix-based case can be devised. Reduction and bounding techniques are extended as shown next.

A variable x_j is essential for f if and only if $f_i \leq x_j$, for some i, $i = 1, 2, \ldots, n$. A variable x_j is unacceptable for f if and only if $f_i \leq x_j'$, for some i, $i = 1, 2, \ldots, n$.

Row dominance is extended to the more general definition of constraint dominance. Function f_i dominates function f_j if and only if $f_j \leq f_i$. Constraint dominance reduces to row dominance if subconstraints coincide with SOP clauses.

Column dominance is extended to the following definition of variable dominance. Variable x_i dominates variable x_j if and only if $c_i \leq c_j$ and $\exists x_i f_{x_j} \leq \exists x_i f_{x_j}'$. Since the constraint f is in the form of conjunction of subconstraints, the previous definition cannot be checked directly. However the following sufficient conditions can be checked efficiently. If either of the following conditions is satisfied

- $(f_k)_{x_j} \leq (f_k)_{x_i x_j'}$ for each f_k
- $(f_k)_{x_j} \leq (f_k)_{x_i' x_j'}$ for each f_k

where $c_i \leq c_j$ then x_i dominates x_j. As another special case, if $(f_k)_{x_j} \leq (f_k)_{x_j'}$ for each f_k, then any variable x_i, $(i \neq j)$ dominates variable x_j.

When x_j has cost 0, a more general definition of variable dominance is that variable x_i dominates variable x_j if and only if and only if $\exists x_i f_{x_j} \leq \exists x_i f_{x_j}'$ or $\exists x_i f_{x_j}' \leq \exists x_i f_{x_j}$.

In [27] variable x_i is said to dominate variable x_j iff $c_i \leq c_j$ and one of the following conditions is satisfied $\forall k \in \{1, \ldots, n\}$:

1. $(f_k)_{x_j} \leq (f_k)_{x_i x_j'}$
2. $(f_k)_{x_j} = (f_k)_{x_j'} = (f_k)$, i.e., f_k does not depend on x_j, and there exists a p such that $(f_p)_{x_i' x_j} \leq (f_k)_{x_i}$

If subconstraints coincide with SOP clauses, the first condition gives the definition of alpha_dominance. If subconstraints coincide with SOP clauses, the first and second condition together give the definition of beta_dominance.

A lower bound to the cost of satisfying F is given by the sum of the minimum costs of satisfying each BDD in a set of BDD's with disjoint supports (an independent set of BDD's). These minimum costs can be found by computing the shortest paths of those BDD's. If the shortest paths satisfy all the other sub-constraints, the solution determined by the independent set is optimal and the current recursion node can be pruned.

A most common variable in the BDD's is chosen as a splitting variable (i.e., a variable whose corresponding column in the dependence matrix intersects most rows). This favours the simplification of as many BDD's as possible, the partitioning of the BDD's in sets with disjoint support and the generation of larger independent sets. Experiments show that this splitting variable criterion is less effective that the one (in section 4.5) used for a matrix-based formulation and as a consequence the number of recursion nodes is greater.

We notice that in both approaches presented in this section, the usage of BDD's allows potentially to handle problems with many clauses (if they have a compact BDD representation), but does not address the problem of covering matrices with many columns. In such problems, it is unlikely that the BDD can be built at all, because each column is a variable in the support of the BDD.

It may be worthy of mention at this point that in [33, 34] a more general algorithm to solve integer linear programming based on edge-valued binary decision diagrams has been presented.

7 Implicit Solution of Binate Covering

```
$mincov(R, C, U)$ {
    $(R, C) = Reduce(R, C, U)$
    if $(Terminal\_Case(R, C))$
        if $(cost(R, C) \geq U)$ return $empty\_solution$
        else $U = cost(R, C)$; return $solution$
    $L = Lower\_Bound(R, C)$
    if $(L \geq U)$ return $\_solution$
    $c_i = Choose\_Column(R, C)$
    $S^1 = mincov(R_{c_i}, C_{c_i}, U)$
    $S^0 = mincov(R_{\overline{c_i}}, C_{\overline{c_i}}, U)$
    return $Best\_Solution(S^1 \cup \{c_i\}, S^0)$
}
```

Figure 6: Implicit branch-and-bound algorithm.

The classical branch-and-bound algorithm [22, 23] for minimum-cost binate covering has been described in previous sections, and implemented by means of efficient computer programs (ESPRESSO and STAMINA). These state-of-the-art binate table solvers represent binate tables efficiently using sparse matrix packages. But the fact that each non-empty table entry still has to be explicitly represented put a bound on the size of the tables that can be handled by these binate solvers. For example, one would not expect these binate solvers to handle examples requiring over 10^6 columns (up to 2^{1500} columns), reported in state minimization of FSM's [29]. To keep with our stated objective, the binate table has to be represented implicitly. We do not represent (even implicitly) the elements of the table, but we make use only of a set of row labels and a set of column labels, each represented implicitly as a BDD. They are chosen so that the existence and value of any table entry can be readily inferred by examining its corresponding row and column labels. In the sequel, we shall assume that every row has a unit cost.

7.1 Implicit Set Manipulation

In [29] it is presented a full-fledged theory on how to represent and manipulate sets using a BDD-based representation. It extends the notation used in [37]. An outline is available also in [31]. This theory is especially useful for applications where sets of sets need to be constructed and manipulated.

Given a ground set G of cardinality less or equal to n, any **subset** S can be represented in a Boolean space B^n by a unique Boolean function $\chi_S : B^n \rightarrow B$, which is called its **characteristic function** [7], such that:

$$\chi_S(x) = 1 \text{ if and only if } x \text{ in } S.$$

In other words, a subset is represented in *positional-set* or *positional-cube* notation form [13], using n Boolean variables, $x = x_1 x_2 \ldots x_n$. The presence of an element s_k in the set is denoted by the fact that variable x_k takes the value 1 in the positional-set, whereas x_k takes the value 0 if element s_k is not a member of the set. One Boolean variable is needed for each element because the element can either be present or absent in the set. As an example, for $n = 6$, the set with a single element s_4 is represented by 000100 and the set $s_2 s_3 s_5$ is represented by 011010. The elements s_1, s_4, s_6 which are not present correspond to 0's in the positional-set.

A set of subsets of G can be represented by a Boolean function, whose minterms correspond to the single subsets. In other words, a set of sets is represented as a set S of positional-sets, by a characteristic function $\chi_S : B^n \to B$ as:

$$\chi_S(x) = 1 \text{ if and only if the set represented by the positional-set } x \text{ is in the set } S \text{ of sets.}$$

Any **relation** $\mathcal{R}$ between a pair of Boolean variables can also be represented by a characteristic function $\mathcal{R} : B^2 \to B$ as:

$$\mathcal{R}(x, y) = 1 \text{ if and only if } x \text{ is in relation } \mathcal{R} \text{ to } y$$

$\mathcal{R}$ can be a one-to-many relation over the two sets in B. These definitions can be extended to any relation $\mathcal{R}$ between n Boolean variables, and can be represented by a characteristic function $\mathcal{R} : B^n \to B$ as:

$$\mathcal{R}(x_1, x_2, \ldots, x_n) = 1 \text{ if and only if the } n\text{-tuple } (x_1, x_2, \ldots, x_n) \text{ is in relation } \mathcal{R}$$

In this way, useful relational operators on sets can be derived. Operators Op acts on two sets of variables $x = x_1 x_2 \ldots x_n$ and $y = y_1 y_2 \ldots y_n$ and returns a relation $(x \; Op \; y)$ (as a characteristic function) of pairs of positional-sets. Alternatively, they can also be viewed as constraints imposed on the possible pairs out of two sets of objects, x and y. For example, given two sets of sets X and Y, the set pairs (x, y) where x contains y are given by the product of X and Y and the containment constraint, $X(x) \cdot Y(y) \cdot (x \supseteq y)$. We present a few examples of these operators. Variations and extensions used later, can be defined in a similar manner.

Lemma 7.1 *The* **equality relation** *evaluates true if the two sets of objects represented by positional-sets x and y are identical, and can be computed as:*

$$(x = y) = \prod_{k=1}^{n} x_k \Leftrightarrow y_k$$

where $x_k \Leftrightarrow y_k = x_k \cdot y_k + \neg x_k \cdot \neg y_k$ designates the Boolean XNOR *operation and $\neg$ designates the Boolean* NOT *operation.*

Proof. $\prod_{k=1}^{n} x_k \Leftrightarrow y_k$ requires that for every element k, either both positional-sets x and y contain it, or it is absent from both. Therefore, x and y contains exactly the same set of elements and thus are equal. □

Lemma 7.2 *The* **containment relation** *evaluates true if the set of objects represented by x contains the set of objects represented by y, and can be computed as:*

$$(x \supseteq y) = \prod_{k=1}^{n} y_k \Rightarrow x_k$$

where $x_k \Rightarrow y_k = \neg x_k + y_k$ designates the Boolean implication operation.

[13] Called also *1-hot encoding*.

Proof. $\prod_{k=1}^{n} y_k \Rightarrow x_k$ requires that for all object, if an object k is present in y (i.e., $y_k = 1$), it must also be present in x ($x_k = 1$). Therefore set x contains all the objects in y. □

Similarly one can define operations on sets of sets. A few important examples follow.

Lemma 7.3 *Given the characteristic functions χ_A and χ_B representing the sets A and B, set operations on them such as the* **union, intersection, sharp,** *and* **complementation** *can be performed as logical operations on their characteristic functions, as follows:*

$$\begin{aligned}\chi_{A\cup B} &= \chi_A + \chi_B\\ \chi_{A\cap B} &= \chi_A \cdot \chi_B\\ \chi_{A-B} &= \chi_A \cdot \neg\chi_B\\ \chi_{\overline{A}} &= \neg\chi_A\end{aligned}$$

Lemma 7.4 *The* **maximal** *of a set χ of subsets is the set containing subsets in χ not strictly contained by any other subset in χ, and can be computed as:*

$$Maximal_x(\chi) = \chi(x) \cdot \nexists y\ [(y \supset x) \cdot \chi(y)]$$

Proof. The term $\exists y\ [(y \supset x) \cdot \chi(y)]$ is true if and only if there is a positional-set y in χ such that $x \subset y$. In such a case, x cannot be in the maximal set by definition, and can be subtracted out. What remains is exactly the maximal set of subsets in $\chi(x)$. □

An efficient recursive implementation of the operation maximal is described in [29].

7.2 Setting of Implicit Solution

A binate covering problem instance can be characterized by a 6-tuple $(r, c, R, C, 0, 1)$, defined as follows:

- the group of variables for labeling the rows: r
- the group of variables for labeling the columns: c
- the set of row labels: $R(r)$
- the set of column labels: $C(r)$
- the 0-entries relation at the intersection of row r and column c: $0(r, c)$
- the 1-entries relation at the intersection of row r and column c: $1(r, c)$

In other words, the user of our implicit binate solver would first choose an encoding for the rows and columns. Given a binate table, the user will then supply a set of row labels as a BDD $R(r)$ and a set of column labels as a BDD $C(c)$, and also the two inference rules in the form of BDD relations, $0(r, c)$ and $1(r, c)$, capturing the 0-entries and 1-entries.

The classical branch-and-bound solution of minimum cost binate covering is based on the recursive procedure as shown in Figure 3. In our implicit formulation, we keep the branch-and-bound scheme summarized in Figure 6, but we replace the traditional description of the table as a (sparse) matrix with an implicit representation, using BDD's for the characteristic functions of the rows and columns of the table. Moreover, we have implicit versions of the manipulations of the binate table required to implement the branch-and-bound scheme. In the following sections we are going to describe the following:

- implicit representation of the covering table,
- implicit reduction,
- implicit branching column selection,
- implicit computation of the lower bound, and
- implicit table partitioning.

At each call of the binate cover routine *mincov*, the binate table undergoes a reduction step *Reduce* and, if termination conditions are not met, a branching column is selected and *mincov* is called recursively twice, once assuming the selected column c_i in the solution set (on the table R_{c_i}, C_{c_i}) and once out of the solution set (on the table $R_{\overline{c_i}}, C_{\overline{c_i}}$). Some suboptimal solutions are bounded away by computing a lower bound L on the current partial solution and comparing it against an upper bound U (best solution obtained so far). A good lower bound is based on the computation of a maximal independent set.

7.3 Implicit Table Generation

Here we define different ways of specifying the binate covering table in decreasing order of generality of the binate covering problem. A table is defined implicitly by generating BDD-based representations of the rows and columns and by giving relations specifying the 1 and 0 entries, given the rows and columns. By imposing restrictions on the way in which rows and columns are labeled and entries are defined, one gets representations with varying degrees of generality. We distinguish between the case of a general binate covering table (1.) and of a binate covering table with at most one 0 per row (2.), even though they use the same table specification, because in the second case some simplifications of the computations to reduce implicitly the table will be possible and pointed out in the text. Historically the third (less general) way was implemented first to solve exact state minimization of ISFSM's [30]. It is applicable to other problems whose covering table can be represented in the same way, e.g., the exact formulation of technology mapping for area minimization [50]. There is a trade-off between generality of the representation and efficiency of the computations: "hard-wiring" the rules that define a table may speed up table manipulations, to the price of more limited applicability.

1. General binate covering table
   - the group of variables for labeling the rows: r
   - the group of variables for labeling the columns: c
   - the set of row labels: $R(r)$
   - the set of column labels: $C(c)$
   - the 0-entries relation at the intersection of row r and column c: $\mathtt{0}(r, c)$
   - the 1-entries relation at the intersection of row r and column c: $\mathtt{1}(r, c)$

   item Binate covering table assuming each row has at most one 0:
   - same as 1. above.
2. Specialized binate covering table for exact state minimization and similar problems:
   - the group of variables for labeling the rows (each label is a pair): (c, d)
   - the group of variables for labeling the columns: p

- the set of row labels: $R(c, d)$
- the set of column labels: $C(p)$
- the 0-entries relation at the intersection of row (c, d) and column p: $0((c, d), p) = (p = c)$
- the 1-entries relation at the intersection of row (c, d) and column p: $1((c, d), p) = (p \supseteq d)$

As an example, for the problem of exact state minimization, $C(p)$ is the set of labels that denote the prime compatibles p of an FSM, i.e., p is in set C if it is the label of a prime compatible p. Prime compatibles are sets of states and they are represented using positional set notation. For instance, if an FSM has 5 states $s1, s2, s3, s4, s5$ and $p = \{s1, s4\}$ is a compatible, set C is represented with 5 Boolean variables and p is labeled as 10010. $R(c, d)$ is the relation expressing covering clauses and closure clauses. A covering clause for a state says that the state must be contained in at least one prime compatible. A binate clause for a compatible says that if the compatible is chosen in a solution then at least another compatible from a related set must be in that solution, e.g., clause $(\overline{p} + p_1 + p_2 + \cdots + p_k)$, meaning that if p is in a solution, either one of $p_1, p_2, \cdots, p_k$ must be in that solution. A covering clause yields a unate row, labeled by a c part that denotes an empty set and by a d part that denotes a singleton set, requiring a given state be covered. Whenever $p \supseteq d$, there is a 1 at the intersection of the row labeled by d and the column representing prime compatible p, meaning that the compatible p contains state d. A closure clause yields a binate row, labeled by a c part that is the label of the unique prime compatible whose corresponding column has a zero at the intersection with this row (condition $p = c$), and by a d part that is the label of a compatible such that there is a 1 at the intersection of this row and any column whose label p is a prime compatible that contains compatible d. We refer to [31] for a complete treatment of implicit state minimization of incompletely specified FSM's.

If the covering problem is unate, the $0(r, c)$ relation is empty. A typical example is exact two-level minimization where $R(r) = R(m)$, for m labeling minterms, $C(c) = C(p)$, for p labeling prime implicants and $1(r, c) = (p \supseteq m)$. The label of an implicant can be constructed by representing each Boolean variable in multi-valued notation, for instance encoding 0 as 10, 1 as 01 and $-$ as 11. A complete treatment of this special case can be found in [24, 8]. The more complex case of implicit exact minimization of generalized prime implicants is described in [57].

In the next section, we will describe how a binate covering table can be manipulated implicitly so as to solve the minimum cost binate covering problem. BDD formulas of implicit table operations will be labeled 1, 2, or 3, depending on which of the three previous formulations it refers to.

8 Implicit Table Reduction Techniques

Reduction rules aim to the following:

1. Selection of a column. A column must be selected if it is the only column that satisfies a given row. A dual statement holds for columns that must not be part of the solution in order to satisfy a given row.

2. Elimination of a column. A column c_i can be eliminated if its elimination does not preclude obtaining a minimum cover, i.e., if there is another column c_j that satisfies at least all the rows satisfied by c_i.

3. Elimination of a row. A row r_i can be eliminated if there exists another row r_j that expresses the same or a stronger constraint.

The order of the reductions affects the final result. Reductions are usually attempted in a given order, until nothing changes any more (i.e., the covering matrix has been reduced to a cyclic core). The reductions and order implemented in our reduction algorithm are summarized in Figure 7.
In the reduction, there are two cases when no solution is generated:

```
$Reduce(R, C, U)$ {
    repeat {
        $Collapse\_Columns(C)$
        $Column\_Dominance(R, C)$
        $Sol = Sol \cup Essential\_Columns(R, C)$
        if ($|Sol| \geq U$) return $empty\_solution$
        $Unacceptable\_Columns(R, C)$
        $Unnecessary\_Columns(R, C)$
        if ($C$ does not cover $R$) return $empty\_solution$
        $Collapse\_Rows(R)$
        $Row\_Dominance(R, C)$
    } until (both $R$ and $C$ unchanged)
    return $(R, C)$
}
```

Figure 7: Implicit reduction loop.

1. The added cardinality of the set of essential columns, and of the partial solution computed so far, Sol, is larger or equal than the upper bound U. In this case, a better solution is known than the one that can be found from now on and so the current computation branch can be bounded away.
2. After having eliminated essential, unacceptable and unnecessary columns and covered rows, it may happen that the rest of the rows cannot be covered by the remaining columns. In this case, the current partial solution cannot be extended to any full solution.

We are going to describe how the reduction operations are performed implicitly using BDD's on the three table representations described in the previous section.

8.1 Duplicated Columns

It is possible that more than one column (row) label is associated with columns (rows) that coincide element by element. We need to identify such duplicated columns (rows) and collapse them into a single column (row). This avoids the problem of columns (rows) dominating each other when performing implicitly column (row) dominance. The following computations can be seen as finding the equivalence relation of duplicated columns (rows) and selecting one representative for each equivalence class.

Definition 8.1 *Two columns are duplicates, if on every row, their corresponding table entries are identical.*

Theorem 8.1 Duplicated columns *can be computed as:*

$$dup\_col(c', c) \overset{1}{=} \forall r \{R(r) \Rightarrow [(0(r, c') \Leftrightarrow 0(r, c)) \cdot (1(r, c') \Leftrightarrow 1(r, c))]\}$$
$$dup\_col(c', c) \overset{2}{=} \forall r \{R(r) \Rightarrow [\neg 0(r, c') \cdot \neg 0(r, c) \cdot (1(r, c') \Leftrightarrow 1(r, c))]\}$$
$$dup\_col(p', p) \overset{3}{=} \nexists d\, R(p', d) \cdot \nexists d\, R(p, d) \cdot \forall d \{[\exists c\, R(c, d)] \Rightarrow [(p' \supseteq d) \Leftrightarrow (p \supseteq d)]\}$$

Proof. As discussed at the end of Section 7.3, the first equation computes the duplicated columns relation for the most general binate table, and the second equation for the binate table with the assumption that there is at most one 0 in each row, and the third equation is for the specialized binate table for state minimization, assuming the columns are prime compatibles p, and the rows are pairs (c, d).

For the column labels c' and c to be in the relation *dup_col*, the first equation requires the following conditions to be met for every row label $r \in R$: (1) the entry (r, c) is a 0 if and only if the entry (r, c') is a 0, (i.e., $0(r, c') \Leftrightarrow 0(r, c)$), and (2) the entry (r, c) is a 1 if and only if the entry (r, c') is a 1, (i.e., $1(r, c') \Leftrightarrow 1(r, c)$). Assuming each row has at most one 0 for the second equation, condition 2 requires that the row labeled r cannot intersect either column at a 0, (i.e., $\neg 0(r, c') \cdot \neg 0(r, c)$). □

Theorem 8.2 *Duplicated columns can be collapsed by:*

$$\begin{aligned} C(c) \quad & {}^{1,2}= \quad C(c) \cdot \not\exists c' \, [C(c') \cdot (c' \prec c) \cdot dup\_col(c', c)] \\ C(p) \quad & {}^{3}= \quad C(p) \cdot \not\exists p' \, [C(p') \cdot (p' \prec p) \cdot dup\_col(p', p)] \end{aligned}$$

Proof. This computation picks a representative column label out of a set of column labels corresponding to duplicated columns. A column label c is deleted from C if and only if there is another column label c' which has a smaller binary value than c (denoted by $c' \prec c$) and both label the same duplicated column. Here we exploit the fact that any positional-set c can be interpreted as a binary number. Therefore, a unique representative from a set can be selected by picking the one with the smallest binary value. [14] □

8.2 Duplicated Rows

Definition 8.2 *Two rows are duplicates if, on every column, their corresponding table entries are identical.*

Detection of duplicated rows, selection of a representative row, and table updating are performed by the following equations as in the case of duplicated columns.

Theorem 8.3 Duplicated rows *can be computed as:*

$$\begin{aligned} dup\_row(r', r) \quad & {}^{1,2}= \quad \forall c \, \{C(c) \Rightarrow [(0(r', c) \Leftrightarrow 0(r, c)) \cdot (1(r', c) \Leftrightarrow 1(r, c))]\} \\ dup\_row(c', d', c, d) \quad & {}^{3}= \quad (c' = c) \cdot \not\exists p \, [C(p) \cdot ((p \supseteq d') \not\Leftrightarrow (p \supseteq d))] \end{aligned}$$

Proof. Similar to the proof for Theorem 8.1. For the row labels r' and r to be in the relation *dup_row*, the first equation requires the following conditions to be met for every column label $c \in C$: (1) the entry (r, c) is a 0 if and only if the entry (r', c) is a 0, (i.e., $0(r', c) \Leftrightarrow 0(r, c)$), and (2) the entry (r, c) is a 1 if and only if the entry (r', c) is a 1, (i.e., $1(r', c) \Leftrightarrow 1(r, c)$). □

Theorem 8.4 *Duplicated rows can be collapsed by:*

$$\begin{aligned} R(r) \quad & {}^{1,2}= \quad R(r) \cdot \not\exists r' \, [R(r') \cdot (r' \prec r) \cdot dup\_row(r', r)] \\ R(c, d) \quad & {}^{3}= \quad R(c, d) \cdot \not\exists c', d' \, [R(c', d') \cdot (d' \prec d) \cdot dup\_row(c', d', c, d)] \end{aligned}$$

Proof. The proof is similar to that for Theorem 8.2, except we are delete all duplicating rows here except the representative ones. □

From now on, sometimes we will blur the distinction between a column (row) label and the column (row) itself, but the context should say clearly which one it is meant.

[14] Alternatively, one could have used the *cproject* BDD operator introduced in [38] to pick a representative column out of each set of duplicated columns.

8.3 Column Dominance

Some columns need not be considered in a binate table, if they are dominated by others. Classically, there are two notions of column dominance: α-dominance and β-dominance.

Definition 8.3 *A column c'* **α-dominates** *another column c if c' has all the 1's of c, and c has all the 0's of c'.*

Theorem 8.5 *The α-dominance relation can be computed as:*

$$\begin{aligned}
\alpha\_dom(c',c) &\overset{1}{=} \not\exists r\,\{R(r)\cdot[1(r,c)\cdot\neg 1(r,c')]+[0(r,c')\cdot\neg 0(r,c)]\}\\
\alpha\_dom(c',c) &\overset{2}{=} \not\exists r\,\{R(r)\cdot[1(r,c)\cdot\neg 1(r,c')+0(r,c')]\}\\
\alpha\_dom(p',p) &\overset{3}{=} \not\exists c,d\,[R(c,d)\cdot(p\supseteq d)\cdot(p'\not\supseteq d)]\cdot \not\exists d\,R(p',d)
\end{aligned}$$

Proof. For column c' to α-dominate c, the first equation ensures that there doesn't exists a row $r \in R$ such that either (1) the table entry (r,c) is a 1 but the table entry (r,c') is not, *or* (2) the table entry (r,c') is a 0 but the table entry (r,c) is not. Assuming each row has at most one 0, condition 2 can be simplified to the second equation that table entry (r,c') is a 0. □

Definition 8.4 *A column c'* **β-dominates** *another column c if (1) c' has all the 1's of c, and (2) for every row r' in which c' contains a 0, there exists another row r in which c has a 0 such that disregarding entries in column c', r' has all the 1's of r.*

Theorem 8.6 *The β-dominance relation can be computed by:*

$$\begin{aligned}
\beta\_dom(c',c) &\overset{1,2}{=} \not\exists r'\,\{R(r')\cdot[1(r',c)\cdot\neg 1(r',c')\\
&\qquad + 0(r',c')\cdot \not\exists r\,[R(r)\cdot 0(r,c)\cdot \not\exists c''\,[C(c'')\cdot(c''\neq c')\cdot 1(r,c'')\cdot\neg 1(r',c'')]\,]]\}\\
\beta\_dom(p',p) &\overset{3}{=} \not\exists d'\,\{\exists c'\,(R(c',d'))\cdot(p\supseteq d')\cdot(p'\not\supseteq d')\}\\
&\qquad\cdot \not\exists d'\,\{R(p',d')\cdot \not\exists d\,[R(p,d)\cdot \not\exists q\,[C(q)\cdot(q\neq p')\cdot(q\supseteq d)\cdot(q\not\supseteq d')]\,]\}\}
\end{aligned}$$

Proof. According to the definition, the table should *not* contain a row $r' \in R$ if either of the following two cases is true at that row: (1) table entry at column c is a 1 while entry at column c' is not a 1 (i.e., $1(r',c)\cdot\neg 1(r',c')$), *or* (2) c' has a 0 in row r' (i.e., $0(r',c')$) but there does not exist a row $r \in R$ such that its column c is a 0 and disregarding entries in column c', row r' has all the 1's of row r. Rephrasing the last part of the condition 2, the expression $\not\exists c''\,[C(c'')\cdot(c''\neq c')\cdot 1(r,c'')\cdot\neg 1(r',c'')]$ requires that there is no column $c'' \in C$ apart from column c' such that c'' has a 1 in row r, but not in row r'. □

The conditions for α-dominance are a strict subset of those for β-dominance, but α-dominance is easier to compute implicitly. Either of them can be used as the column dominance relation *col_dom*.

Theorem 8.7 *The set of dominated columns in a table (R,C) can be computed as:*

$$\begin{aligned}
D(c) &\overset{1,2}{=} C(c)\cdot\exists c'\,[C(c')\cdot(c'\neq c)\cdot col\_dom(c',c)]\\
D(p) &\overset{3}{=} C(p)\cdot\exists p'\,[C(p')\cdot(p'\neq p)\cdot col\_dom(p',p)]
\end{aligned}$$

Proof. A column $c \in C$ is dominated if there is another $c' \in C$ different from c (i.e., $c' \neq c$) which column dominates c (i.e., $col\_dom(c',c)$). □

Theorem 8.8 *The following computations delete a set of columns $D(c)$ from a table (R, C) and all rows intersecting these columns in a 0.*

$$\begin{aligned} C(c) &\overset{1,2}{=} C(c) \cdot \neg D(c) \\ R(r) &\overset{1,2}{=} R(r) \cdot \nexists c\,[D(c) \cdot 0(r,c)] \end{aligned}$$

$$\begin{aligned} C(p) &\overset{3}{=} C(p) \cdot \neg D(p) \\ R(c,d) &\overset{3}{=} R(c,d) \cdot \neg D(c) \end{aligned}$$

Proof. The first computation removes columns in $D(c)$ from the set of columns $C(c)$. The expression $\exists c\,[D(c) \cdot 0(r,c)]$ defines all rows r intersecting the columns in D in a 0. They are deleted from the set of rows R. □

8.4 Row Dominance

Definition 8.5 *A row r'* **dominates** *another row r if r has all the 1's and 0's of r'.*

Theorem 8.9 *The row dominance relation can be computed by:*

$$\begin{aligned} row\_dom(r', r) &\overset{1,2}{=} \nexists c\,\{C(c) \cdot [1(r',c) \cdot \neg 1(r,c) + 0(r',c) \cdot \neg 0(r,c)]\} \\ row\_dom(c', d', c, d) &\overset{3}{=} \nexists p\,[C(p) \cdot (p \supseteq d') \cdot (p \not\supseteq d)] \cdot [unate\_row(c') + (c' = c)] \end{aligned}$$

Proof. For r' to dominate r, the equation requires that there is no column $c \in C$ such that either (1) the table entry (r', c) is a 1 but the entry (r, c) is not, *or* (2) the entry (r', c) is a 0 but the entry (r, c) is not. □

Theorem 8.10 *Given a table $(R(r), C(c))$, the set of unate row labels r can be computed as*

$$unate\_row(r) \overset{1,2}{=} \nexists c\,[C(c) \cdot 0(r,c)].$$

Given a table $(R(c,d), C(p))$, the set of unate row labels c can be computed as

$$unate\_row(c) \overset{3}{=} \nexists p\,[C(p) \cdot (p = c)] = \nexists c\, C(c).$$

Theorem 8.11 *The set of rows not dominated by other rows can be computed as:*

$$\begin{aligned} R(r) &\overset{1,2}{=} R(r) \cdot \nexists r'\,[R(r') \cdot (r' \neq r) \cdot row\_dom(r', r)] \\ R(c,d) &\overset{3}{=} R(c,d) \cdot \nexists c', d'\,\{R(c',d') \cdot [(c',d') \neq (c,d)] \cdot row\_dom(c',d',c,d)]\} \end{aligned}$$

Proof. The equation expresses that any row $r \in R$, dominated by another *different* row $r' \in R$, is deleted from the set of rows $R(r)$ in the table. □

8.5 Essential Columns

Definition 8.6 *A column c is an* **essential column** *if there is a row having a 1 in column c and 2 everywhere else.*

Theorem 8.12 *The set of essential columns can be computed by:*

$$\begin{aligned}
ess\_col(c) &\overset{1}{=} C(c)\cdot\exists r\,\{R(r)\cdot 1(r,c)\cdot \nexists c'[C(c')\cdot(c'\neq c)\cdot(0(r,c')+1(r,c'))]\}\\
ess\_col(c) &\overset{2}{=} C(c)\cdot\exists r\,\{R(r)\cdot 1(r,c)\cdot unate\_row(r)\cdot \nexists c'\,[C(c')\cdot(c'\neq c)\cdot 1(r,c')]\}\\
ess\_col(p) &\overset{3}{=} C(p)\cdot\exists c,d\,\{R(c,d)\cdot(p\supseteq d)\cdot unate\_row(c)\cdot \nexists p'\,[C(p')\cdot(p'\neq p)\cdot(p'\supseteq d)]\}
\end{aligned}$$

Proof. For a column $c\in C$ to be essential, there must exist a row $r\in R$ which (1) contains a 1 in column c (i.e., $1(r,c)$), and (2) there is not another *different* column intersecting the row in a 1 or 0 (i.e., $\nexists c'\,[C(c')\cdot(c'\neq c)\cdot(0(r,c')+1(r,c'))]$).

Assuming that a row can have at most one 0, a column $c\in C$ is essential if and only if there is a row $r\in R$ which (1) contains a 1 in column c (i.e., $1(r,c)$), and (2) does not contain any 0 (i.e., $unate\_row(r)$), and (3) there is not another *different* column intersecting the row in a 1 (i.e., $\nexists c'\,[C(c')\cdot(c'\neq c)\cdot 1(r,c')]$). □

Theorem 8.13 *Essential columns must be in the solution. Each essential column must then be deleted from the table together with all rows where it has 1's.*

The following computations add essential columns to the solution, delete them from the set of columns and delete all rows in which they have 1's:

$$\begin{aligned}
solution(c) &\overset{1,2}{=} solution(c)+ess\_col(c)\\
C(c) &\overset{1,2}{=} C(c)\cdot\neg ess\_col(c)\\
R(r) &\overset{1,2}{=} R(r)\cdot\nexists c\,[ess\_col(c)\cdot 1(r,c)]
\end{aligned}$$

$$\begin{aligned}
solution(p) &\overset{3}{=} solution(p)+ess\_col(p)\\
C(p) &\overset{3}{=} C(p)\cdot\neg ess\_col(p)\\
R(c,d) &\overset{3}{=} R(c,d)\cdot\neg ess\_col(c)
\end{aligned}$$

Proof. The first two equations move the essential columns from the column set to the solution set. The third equation deletes from the set of rows R all rows intersecting an essential column c in a 1. □

8.6 Unacceptable Columns

Definition 8.7 *A column c is an* **unacceptable column** *if there is a row having a 0 in column c and 2 everywhere else.*

Theorem 8.14 *The set of unacceptable columns can be computed by:*

$$\begin{aligned}
unacceptable\_col(c) &\overset{1}{=} C(c)\cdot\exists r\,\{R(r)\cdot 0(r,c)\cdot\nexists c'\,[C(c')\cdot(c'\neq c)\cdot 0(r,c')]\}\\
&\quad\cdot\nexists c'\,[C(c')\cdot 1(r,c')]\}\\
unacceptable\_col(c) &\overset{2}{=} C(c)\cdot\exists r\,\{R(r)\cdot 0(r,c)\cdot\nexists c'\,[C(c')\cdot 1(r,c')]\}\\
unacceptable\_col(p) &\overset{3}{=} C(p)\cdot\exists d\,\{R(p,d)\cdot\nexists p'\,[C(p')\cdot(p'\supseteq d)]\}
\end{aligned}$$

Proof. For column $c \in C$ to be unacceptable, there must be a row $r \in R$ such that (1) it intersects the column c at a 0, and (2) there does not exists another column c' different from c which intersects that row r at a 0 (i.e., $\not\exists c'\ [C(c') \cdot (c' \neq c) \cdot 0(r, c')]$), and (3) no column c' intersects that row r in a 1 (i.e., $\not\exists c'\ [C(c') \cdot 1(r, c')]$). Condition 2 is not needed if we assume that each row contains at most one 0. □

8.7 Unnecessary Columns

Definition 8.8 *A column is an* **unnecessary column** *if it does not have any 1 in it.*

Theorem 8.15 *The set of unnecessary columns can be computed as:*

$$\begin{aligned} unnecessary\_col(c) &\overset{1,2}{=} C(c) \cdot \not\exists r\, [R(r) \cdot 1(r, c)] \\ unnecessary\_col(p) &\overset{3}{=} C(p) \cdot \not\exists c, d\, [R(c, d) \cdot (p \supseteq d)] \end{aligned}$$

Proof. A column $c \in C$ is unnecessary if no row $r \in R$ intersects it in a 1. □

Theorem 8.16 *Unacceptable and unnecessary columns should be eliminated from the table, together with all the rows in which such columns have 0's.*

The table (R, C) is updated according to Theorem 8.8 by setting

$$\begin{aligned} D(c) &\overset{1,2}{=} unacceptable\_col(c) + unnecessary\_col(c) \\ D(p) &\overset{3}{=} unacceptable\_col(p) + unnecessary\_col(p) \end{aligned}$$

Proof. Obvious. □

9 Other Implicit Table Manipulations

To have a fully implicit binate covering algorithm as described in Section 7, we must also compute implicitly a branching column and a lower bound. These computations as well as table partitioning involve solving a common subproblem of finding columns in a table which have the maximum number of 1's.

9.1 Selection of Columns with Maximum Number of 1's

Given a binary relation $F(r, c)$ as a BDD, the abstracted problem is to find a subset of c's each of which relates to the maximum number of r's in $F(r, c)$. An inefficient method is to cofactor F with respect to c taking each possible values c_i, count the number of onset minterms of each $F(r, c)|_{c=c_i}$, and pick the c_i's with the maximum count. Instead our algorithm, $Lmax$, traverses each node of F exactly once as shown by the pseudo-code in Figure 8.

$Lmax$ takes a relation $F(r, c)$ and the variables set r as arguments and returns the set G of c's which are related to the maximum number of r's in F, together with the maximum count. Variables in c are required to be ordered before variables in r. Starting from the root of BDD F, the algorithm traverses down the graph by recursively calling $Lmax$ on its *then* and *else* subgraphs. This recursion stops when the top variable v of F is within the variable set r. In this case, the BDD rooted at v corresponds to a cofactor $F(r, c)|_{c=c_i}$ for some c_i. The minterms in its onset are counted and returned as *count*, which is the number of r's that are related to c_i.

```
$Lmax(F, r)$ {
    $v = bdd\_top\_var(F)$
    if ($v \in r$)
        return ($\mathbf{1}, bdd\_count\_onset(F)$)
    else { /* $v$ is a $c$ variable */
        $(T, count\_T) = Lmax(bdd\_then(F), r)$
        $(E, count\_E) = Lmax(bdd\_else(F), r)$
        $count = max(count\_T, count\_E)$
        if ($count\_T = count\_E$)
            $G = ITE(v, T, E)$
        else if ($count = count\_T$)
            $G = ITE(v, T, \mathbf{0})$
        else if ($count = count\_E$)
            $G = ITE(v, \mathbf{0}, E)$
        return ($G, count$)
    }
}
```

Figure 8: Pseudo-code for the $Lmax$ operator.

During the upward traversal of F, we construct a new BDD G in a bottom up fashion, representing the set of c's with maximum count. The two recursive calls of $Lmax$ return the sets $T(c)$ and $E(c)$ with maximum counts $count\_T$ and $count\_E$ for the $then$ and the $else$ subgraphs. The larger of the two counts is returned. If the two counts are the same, the columns in T and E are merged by $ITE(v, T, E)$ and returned. If $count\_T$ is larger, only T is retained as the updated columns of maximum count. And symmetrically for the other case. To guarantee that each node of BDD $F(r, c)$ is traversed once, the results of $Lmax$ and bdd_count_onset are memoized in computed tables. Note that $Lmax$ returns a set of c's of maximum count. If we need only one c, some heuristic can be used to break the ties.

Example 9.1 *To understand how* Lmax *works consider the explicit binate table:*

	00	01	10	11
00	1	2	1	1
01	2	1	1	2
10	2	1	2	1
11	2	1	2	1

with four rows and four columns. The columns that maximize the number of 1*'s are the second and the fourth. If the rows and columns are encoded by 2 Boolean variables each, using the encodings given on top of each column and to the left of each row, the 1 entries of the table are represented implicitly by the relation $F(c, r)$* [15] *whose minterms are:*

$$\{0000, 1000, 1100, 0101, 1001, 0110, 1110, 0111, 1111\}.$$

[15] r and c are swapped in F so that minterms are listed in the order of the BDD variables.

The BDD representing F is shown in Figure 9. The result of invoking Lmax *on $F(r,c)$ is a BDD representing the relation $G(c)$ whose minterms are: $\{01, 11\}$, corresponding to the encodings of the second and fourth column.*

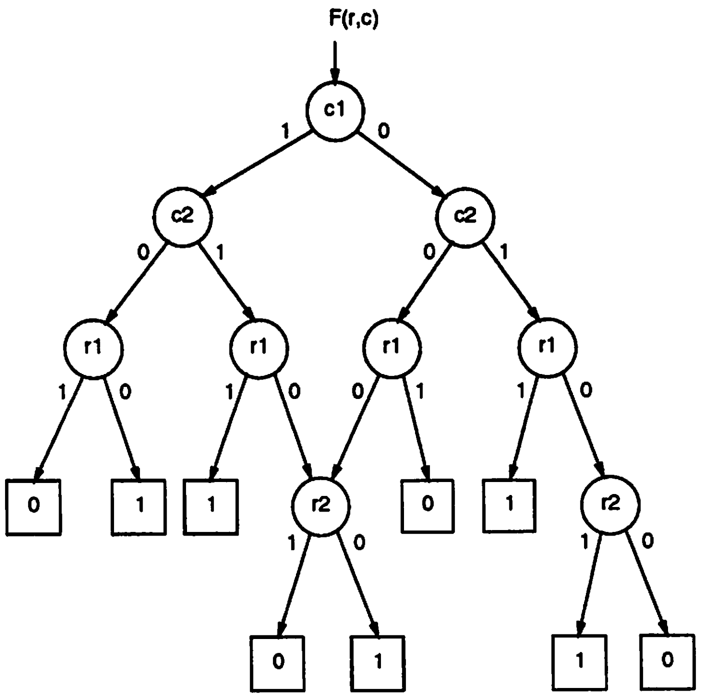

Figure 9: BDD of $F(r,c)$ to illustrate the routine $Lmax$.

9.2 Implicit Selection of a Branching Column

The selection of a branching column is a key ingredient of an efficient branch-and-bound covering algorithm. A good choice reduces the number of recursive calls, by helping to discover more quickly a good solution. We adopt a simplified selection criterion: select a column with a maximum number of 1's. By defining $F'(r,c) = R(r) \cdot C(c) \cdot 1(r,c)$ which evaluates true if and only table entry (r,c) is a 1, our column selection problem reduces to one of finding the c related to the maximum number of r's in the relation $F'(r,c)$, and so it can be found implicitly by calling $Lmax(F', r)$. A more refined strategy is to restrict our selection of a branching column to columns intersecting rows of a maximal independent set, because a unique column must eventually be selected from each independent row. A maximal independent set can be computed as follows.

9.3 Implicit Selection of a Maximal Independent Set of Rows

Usually a lower bound is obtained by computing a maximum independent set of the unate rows. A maximum independent set of rows is a (maximum) set of rows, no two of which intersect the same column at a 1. Maximum independent set is an NP-hard problem and an approximate one (only maximal) can be computed by a greedy algorithm. The strategy is to select *short unate rows* from the table, so we construct a relation $F''(c,r) = R(r) \cdot unate\_row(r) \cdot C(c) \cdot 1(r,c)$. Variables in r are ordered *before* those in c. The rows with the minimum number of 1's in F'' can be computed by $Lmin(F'', c)$, by replacing in $Lmax$ the expression

$max(count\_T, count\_E)$ with $min(count\_T, count\_E)$. Once a shortest row, $shortest(r)$, is selected, all rows having 1-elements in common with $shortest(r)$ are discarded from $F''(c,r)$ by:

$$F''(c,r) = F''(c,r).\ \not\exists c'\ \{\exists r'\ [shortest(r') \cdot F''(c',r')] \cdot F''(c',r)\}$$

Another shortest row can then be extracted from the remaining table F'' and so on, until F'' becomes empty. The maximum independent set consists of all $shortest(r)$ so selected.

9.4 Implicit Covering Table Partitioning

If a covering table can be partitioned into n disjoint blocks, the minimum covering for the original table is the union of the minimum coverings for the n sub-blocks. Let us define the nonempty-entry relation $\mathtt{01}(r,c) = \mathtt{0}(r,c) + \mathtt{1}(r,c)$. The implicit algorithm in Figure 10 takes a table description in terms of its set of rows $R(r)$, its set of columns $C(c)$ and the nonempty-entry relation $\mathtt{01}(r,c)$, partitions it into n disjoint sub-blocks, and return them as n pairs of (R^i, C^i), each corresponding to the rows and columns for the i-th sub-block.

n-way partitioning can be accomplished by successive extraction of disjoint blocks from the table. When the following iteration reaches a fixed point, (R_k, C_k) corresponds to a disjoint sub-block in (R, C).

$$\begin{aligned} R_0(r) &= Lmax(R(r) \cdot C(c) \cdot \mathtt{01}(r,c)\ ,\ c) \\ C_k(c) &= C(c) \cdot \exists r\ \{R_{k-1}(r) \cdot \mathtt{01}(r,c)\} \\ R_k(r) &= R(r) \cdot \exists c\ \{C_k(c) \cdot \mathtt{01}(r,c)\} \end{aligned}$$

This sub-block is extracted from the table (R, C) and the above iteration is applied again to the remaining table, until the table becomes empty. [30] provides a more detailed explanation.

Given a covering table, a single row $R_0(r)$, which has the maximum number of nonempty entries, is first picked using $Lmax()$. The set of columns $C_1(c)$ intersecting this row at 0 or 1 entries is given by $C(c) \cdot \exists r\ [R_0(r) \cdot \mathtt{01}(r,c)]$ (we want $c \in C$ such that there is a row $r \in R_0$ which intersects c at a 0 or 1). Next we find the set of rows R_1 intersecting the columns in C_1 via nonempty entries, by a similar computation $R(r) \cdot \exists c\ [C_1(c) \cdot \mathtt{01}(r,c)]$. Then we can extract all the rows $R_2(r)$ which intersects $C_1(c)$, and so on. This pair of computations is iteratively applied within the *repeat* loop in Figure 10 until no new connected row or column can be found (i.e., $R_k = R_{k-1}$). Effectively, starting from a row, we have extracted a disjoint block (R^1, C^1) from the table, which will later be returned. The remaining table after bi-partition simply contains the rows $R - R^1$ and the columns $C - C^1$. If the remaining table is not empty, we will extract another partition (R^2, C^2) by passing through the outer while loop a second time. If the original table contains n disjoint blocks, the algorithm is guaranteed to return exactly the n sub-blocks by passing through the outer *while* loop n times.

10 Quantifier-Free Table Reductions

The implicit computations presented to manipulate a binate table are valid *a fortiori* when the table is unate. In the latter case, however, more specialized algorithms can be designed to better exploit the features of the problem. Historically speaking, an implicitization of covering problems has been accomplished first for the case of unate tables generated in the minimization of two-level logic functions. A long list of papers has been produced on the subject [24, 12, 37, 14, 13, 11, 15, 17, 18, 8]. Here we will outline some key points.

Given a Boolean function f, consider the problem of finding a minimum two-level cover. A classical exact algorithm by Quine and McCluskey reduces it to a unate covering problem where the rows of the table are minterms and the columns of the table are primes of the function. There is a 1 at the intersection

```
$n\_way\_partition(R(r), C(c), 01(r,c))$ {
    $n = 0$
    while ($R$ not empty) {
        $k = 0$
        $R_0(r) = Lmax(R(r) \cdot C(c) \cdot 01(r,c))$
        repeat {
            $k = k + 1$
            $C_k(c) = C(c) \cdot \exists r \{R_{k-1}(r) \cdot 01(r,c)\}$
            $R_k(r) = R(r) \cdot \exists c \{C_k(c) \cdot 01(r,c)\}$
        } until ($R_k = R_{k-1}$)
        $R^n = R_k$
        $C^n = C_k$
        $R = R - R_k$
        $C = C - C_k$
        $n = n + 1$
    }
    return $\{(R^i, C^i) : 0 \leq i \leq n-1\}$
}
```

Figure 10: Implicit n-way partitioning of a covering table.

of a row and column, if the prime associated to the column contains the minterm associated to the row. A routine that solves explicitly a unate table is available in the program ESPRESSO. In that implementation an improvement has been introduced, namely there is only one row for each set of minterms that are covered by the same set of primes. In other words, the table is constructed in such a way that there are no equal rows in it.

This problem can be implicitized as a special case of the scheme presented in the binate case. Basically one assigns labels to primes and minterms in such a way that there is a 1 at the intersection of a column and a row if and only if the corresponding prime label contains the corresponding minterm label. What is new here is that in this special case the computation to reduce a unate table can be made quantifier-free, i.e., one can define recursive computations that work directly on binary decision graphs representations and build the graphs representing the reduced sets of column and row labels, without quantifications. To achieve the same goal with quantified computations one may incur more easily in the danger of trying to build intermediate graphs too large to be stored in memory. This is considered a winning technique that accounts for part of the success of the application. We will illustrate the point made in the case of row dominance, referring to [8] for a complete treatment.

We remind that a literal is a propositional variable x_k or its negation $\overline{x_k}$. P_n is the set of products that can be built from the set of variables $\{x_1, \ldots, x_n\}$. The subset relation $\subseteq$ is a partial order on the set P_n. P is maximal if and only if there do not exist two products p and p' of P such that $p \subset p'$. A product p is an implicant of a Boolean function f if and only if $p \subseteq \{x \in \{0,1\}^n \mid f(x) \neq 0\}$. A product p is a prime implicant of f if and only if it is a maximal element of the set of implicants of f with respect to $\subseteq$. Any subset P of P_n can be partitioned in the following way:

$$P = P_{1_k} \cup (\{\overline{x_k}\} \times P_{\overline{x_k}}) \cup (\{x_k\} \times P_{x_k})$$

where P_{1_k} is the set of products of P where neither the variable x_k nor $\overline{x_k}$ occurs; $P_{\overline{x_k}}$ (respectively P_{x_k}) is the set of products of P where $\overline{x_k}$ (x_k) occurs, after dropping $\overline{x_k}$ (x_k).

A Boolean space to represent all products can be obtained by a number of variables double with respect to the number of input variables of f. It is the *metaproduct* representation in the literature by researchers at Bull and the *extended space* in the literature by researchers at UCB. The basic idea is to encode the presence of x_k or $\overline{x_k}$ or both (i.e., neither literal appears explicitly in the product) with two bits. The table covering problem can now be described by the triple $< Q, P, \subseteq >$, where Q is the set of minterms of f, P is the set of primes of f and $\subseteq$ describes the table building relation.

A unate table is reduced by applying row and column dominance and detection of essential primes. Consider row dominance, a row R' dominates another row R if and only if R has all the 1's of R'. In the terminology of $< Q, P, \subseteq >$, if q is the label of R and q' is the label of R', this translates into:

Definition 10.1 $q \preceq_Q q' \Leftrightarrow (\forall p \in P\ (q' \subseteq p) \Rightarrow (q \subseteq p))$.

Moreover, if there are rows that intersect exactly the same set of columns, i.e., are equivalent, one should compute this equivalence relation and then replace each equivalence class with one representative (sometimes called *c-projection* operation [36]). Row dominance should then be applied to these representatives only.

Instead of using such a projection and then applying the definition of dominance relation, one can define a row transposing function that maps the rows on objects whose manipulation can be done more efficiently. The maximal elements of the transposed objects are the dominating rows.

The basic idea is that each row of a covering table corresponds to a cube, called *signature cube*, that is the intersection of the primes covering the minterm associated to the row. This was noticed first in [45]. A rigorous theory and an efficient algorithm were presented in [43]. The steps of the algorithm follow:

1. Compute the signature cube of the each cube of an arbitrary initial cover and make irredundant the resulting cover.

2. Since for each cube of an arbitrary irredundant cover of signature cubes there is some essential signature cube contained by it, obtain the irredundant cover of essential signature cubes (called minimum canonical cover).

3. For each cube of the minimum canonical cover, generate the set of primes containing it (the essential signature set).

4. Solve the resulting unate covering problem as usual.

The resulting unate covering problem is exactly what one could get by applying row domination to the minterms/primes table.

One can define a row transposing function $\tau_Q : Q \longrightarrow P_n$ based on the idea of signature cubes.

Definition 10.1 $\tau_Q(q) = \bigcap_{\{p \in P | q \subseteq p\}} p$.

In other words, each element of $\tau_Q(Q)$ is obtained by an element q of Q, by intersecting all elements of P that cover q. The following theorem relates row dominance to the row transposing function.

Theorem 10.1 *The function τ_Q is such that $q \preceq_Q q' \Leftrightarrow \tau_Q(q) \subseteq \tau_Q(q')$.*

Given a set covering problem $(Q, P, \subseteq)$, the function $max_{\subseteq}\tau_Q(Q)$ computes the maximal elements of the set $\tau_Q(Q)$, i.e., the dominating rows.

Since the range τ_Q is P_n, the computation of τ_Q can be easily transposed to the case of the extended space or metaproducts representation. The most obvious implementation would use quantified Boolean

```
$MaxTauQ(Q, P, k)$ {
    if $Q = \emptyset$ or $Q = \emptyset$ {
    if $P = \{1\}$ return $\{1\}$
    $K0 = Subset(Q_{\overline{x_k}}, P_{\overline{x_k}})$
    $K1 = Subset(Q_{x_k}, P_{x_k})$
    $K0 = Q_{1k} \cup (Q_{\overline{x_k}} \setminus K0) \cup (Q_{x_k} \setminus K1)$
    $R = MaxTauQ(K, P_{1_k}, k+1)$
    $R0 = MaxTauQ(K0, P_{1_k} \cup P_{\overline{x_k}}), k+1)$
    $R1 = MaxTauQ(K1, P_{1_k} \cup P_{x_k}), k+1)$
    return $R \cup$
            $\{\overline{x_k}\} \times Subset(R0, R)) \cup$
            $\{x_k\} \times Subset(R1, R)) \cup$
}
```

Figure 11: Recursive computation of $max_{\subseteq}\tau_Q(Q)$

formulas, but in practice they tend to produce huge intermediate ROBDD's. A quantifier free recursive computation of $max_{\subseteq}\tau_Q(Q)$ has given better experimental results.

We present now a pseudo-code description from [17] of $MaxTauQ(Q, P, k)$, a quantifier-free recursive procedure that compute $max_{\subseteq}\tau_Q(Q)$. It uses two auxiliary functions $Supset(P, Q) \equiv \{p \in P \mid \exists q \in Qp \supseteq q$, and $Subset(P, Q) \equiv \{p \in P \mid \exists q \in Qp \subseteq q\}$.

Theorem 10.2 *$MaxTauQ(Q, P, 1)$ computes $max_{\subseteq}\tau_Q(Q)$.*

Proof. The terminal cases are easy. Consider a variable x_k. One can divide the set P in three subsets: P_{x_k}, the products of P in which x_k occurs, $P_{\overline{x_k}}$, the products of P in which $\overline{x_k}$ occurs and P_{1_k}, the products of P in which neither x_k nor $\overline{x_k}$ occurs. Similarly, one can divide the set Q in three subsets: Q_{x_k}, the products of Q in which x_k occurs, $Q_{\overline{x_k}}$, the products of Q in which $\overline{x_k}$ occurs and Q_{1_k}, the products of Q in which neither x_k nor $\overline{x_k}$ occurs.

The products of $Q_{\overline{x_k}}$ can be contained by products of $P_{\overline{x_k}}$ or by products of P_{1_k}. The products of Q_{x_k} can be contained by products of P_{x_k} or by products of of P_{1_k}. The products of Q_{1k} can be contained only by products of P_{1_k}. $K0$ has the products of $Q_{\overline{x_k}}$ contained by products of $P_{\overline{x_k}}$. $K1$ has the products of Q_{x_k} contained by products of P_{x_k}. K has the products of Q_{1_k}, the products of $Q_{\overline{x_k}}$ that are not contained by products of $P_{\overline{x_k}}$ and the products of Q_{x_k} that are not contained by products of P_{x_k}.

Also the set $MaxTauQ(Q, P, 1)$ can be divided in three subsets: the set of products in which x_k occurs, the set of products in which $\overline{x_k}$ occurs and the set of products of P in which neither x_k nor $\overline{x_k}$ occurs. The last set is given by R, that is $MaxTauQ(K, P_{1_k}, k+1)$. Indeed in R the second argument is P_{1_k}, the set of products of P where neither x_k nor $\overline{x_k}$ occurs. The first argument is K that includes the products of Q where x_k nor $\overline{x_k}$ occurs and so can be contained only by products of P_{1_k}, and the products of Q where either x_k or $\overline{x_k}$ occurs but they are not covered by P_{x_k} or $P_{\overline{x_k}}$ and so they can be covered only by P_{1_k}. The second set is obtained from $R0$, that is $MaxTauQ(K0, P_{1_k} \cup P_{\overline{x_k}}, k+1)$, by the following modification. In the first argument of $R0$ there are the products of Q where $\overline{x_k}$ occurs, which are contained by the products of P in the second argument. A product in $R0$ must be multiplied by $\{\overline{x_k}\}$ because for sure each $q \in K0$ is contained by a product of $P_{\overline{x_k}}$, and by definition of $\tau_Q(q)$ one must intersect all the products that contain q. But before multiplying by $\{\overline{x_k}\}$ we must subtract from $R0$ the products contained in R ($Subset(R0, R)$), because if a product $r0$ of $R0$ is contained by a product r of R (or is equal to) it means that there are $q \in K$ and $q0 \in K0$ such that $\tau_Q(q) \supseteq \tau_Q(q0)$ (because r contains $r0$ and $r0$ is multiplied by $\{\overline{x_k}\}$) and we want to keep only $\tau_Q(q)$ because we are computing $max_{\subseteq}\tau_Q$. Instead if a product of R is contained by a product of $R0$, the fact that the product of $R0$ must be multiplied by $\{\overline{x_k}\}$ makes the two products not comparable. Therefore $\{x_k\} \times (R0 \setminus Subset(R0, R))$ is the set of products of $MaxTauQ(Q, P, 1)$ in which $\overline{x_k}$ occurs. Replacing verbatim $\{\overline{x_k}\}$ with x_k, the same reasoning applies for the addition coming from $R1$, from which the first set is obtained. □

After the set $Q' = max_{\subseteq}\tau_Q(Q)$ has been computed, the problem $< Q, P, \subseteq >$ transforms to $< Q', P, R' >$, where $q'R'p$ if and only if $q' = \tau_Q(q)$ and $q \subset p$. $R' \equiv \subseteq$, since $q \subseteq p$ if and only if $\tau_Q(q) \subseteq p$. Therefore the new covering problem is $< Q', P, \subseteq >$.

A similar development allows to compute column dominance by finding the maximal elements of a set to which columns are mapped by a column transposing function. We refer for details to [17].

In [18] it is stated that the usage of Zero-Suppressed BDD's by Minato [44] instead of ROBDD's [6] resulted in more efficient implicit representations of the computations of the problem.

11 Experimental Results of Binate Covering

We implemented a specialized solver where the table is specified as in variant 3. of Section 7.3 (Specialized binate covering table for exact state minimization and similar problems) and we applied it to the problem of exact state minimization of incompletely specified FSM's (ISFSM's) [31].

We implemented also a more general solver that does not rely on a hard-wired rule to determine 0 and 1 entries, but instead works with relations $0(r, c)$ and $0(r, c)$ for 0 entries and 1 entries. It corresponds to variant 2. of Section 7.3 (Binate covering table assuming each row has at most one 0). The difference between variant 1. and 2. is not in the specification of the table, but in the computations for table reduction that can be simplified in the latter case. We applied it to the problem of exact state minimization of pseudo-deterministic FSM's [32]. The same binate solver was applied also to the problem of selection of generalized prime implicants [57].

In this section we report results of two applications of the previous implicit binate covering algorithms. We will concentrate on the experimental performance of binate covering, referring to the original papers for a full-fledged description of the specific applications.

11.1 State Minimization of ISFSM's

Here we provide data for a subset of them, sufficient to characterize the capabilities of our prototype program.

Comparisons of our program ISM are made with STAMINA. The binate covering step of STAMINA was run with no row consensus, because row consensus has not been implemented in our implicit binate solver. Our implicit binate program does not feature Gimpel's reduction rule, that was instead invoked in the version of STAMINA used for comparison. This might sometimes favour STAMINA, but for simplicity we will not elaborate further on this effect. Missing from our package is also table partitioning. All run times are reported in CPU seconds on a DECstation 5000/260 with 440 Mb of memory.
The following explanations refer to the tables of results:

- Under table size we provide the dimensions of the original binate table and of its cyclic core, i.e., the dimensions of the table obtained when the first cycle of reductions converges.
- # mincov is the number of recursive calls of the binate cover routine.
- α and β mean, respectively, α and β dominance.
- Data are reported with a $*$ in front, when only the first solution was computed.

- Data are reported with a † in front, when only the first table reduction was performed.
- # cover is the cardinality of a minimum cost solution (when only the first solution has been computed, it is the cardinality of the first solution).
- CPU time refers only to the binate covering algorithm. It does not include the time to find the prime compatibles.

11.1.1 Minimizing Small and Medium Examples

With the exception of *ex2*, *ex3*, *ex5*, *ex7*, the examples from the MCNC and asynchronous benchmarks do not require prime compatibles for exact state minimization and yield simple covering problems[16]. Table 1 reports those few non-trivial examples. They were all run to full completion, with the exception of *ex2*. In the case of *ex2*, we stopped both programs at the first solution.

FSM	table size (rows x columns)		# mincov				# cover				CPU time (sec)			
	before	after first	ISM		STAMINA		ISM		STAMINA		ISM		STAMINA	
	reduction	α reduction	α	β	α	β	α	β	α	β	α	β	α	β
ex2	4418 x 1366	3425 x 1352	*6	*14	*6	*4	*10	*12	*10	*9	*58	*293	*116	*91
ex2	4418 x 1366	3425 x 1352	*6	*14	*6	286	*10	*12	*10	5	*58	*293	*116	2100
ex3	243 x 91	151 x 84	201	37	91	39	4	4	4	4	78	33	0	0
ex5	81 x 38	47 x 31	16	6	10	6	3	3	3	3	4	3	0	0
ex7	137 x 57	62 x 44	38	31	37	6	3	3	3	3	8	12	0	0

Table 1: Examples from the MCNC benchmark.

These experiments suggest that

- the number of recursive calls of the binate cover routine (*# mincov*) of ISM and STAMINA is roughly comparable, showing that our implicit branching selection routine is satisfactory. This is an important indication, because selecting a good branching column is a more difficult task in the implicit frame.
- the running times are better for STAMINA except in the largest example, *ex2*, where ISM is slightly faster than STAMINA. This is to be expected because when the size of the table is small the implicit approach has no special advantage, but it starts to pay off scaling up the instances. Moreover, our implicit reduction computations have not yet been fully optimized.

11.1.2 Minimizing Constructed Examples

Table 2 presents a few randomly generated FSM's. They generate giant binate tables. The experiments show that ISM is capable of reducing those table and of producing a minimum solution or at least a solution. This is beyond reach of an explicit technique and substantiates the claim that implicit techniques advance decisively the size of instances that can be solved exactly.

11.1.3 Minimizing FSM's from Learning I/O Sequences

Examples in Table 2 demonstrate dramatically the capability of implicit techniques to build and solve huge binate covering problems on suites of contrived examples. Do similar cases arise in real synthesis

[16]Moreover, in the case of the asynchronous benchmark a more appropriate formulation of state minimization requires all compatibles and a different set-up of the covering problem.

FSM	table size (rows x columns)		# mincov				# cover				CPU time (sec)			
	before	after first	ISM		STAMINA		ISM		STAMINA		ISM		STAMINA	
	reduction	α reduction	α	β	α	β	α	β	α	β	α	β	α	β
ex2.271	95323 x 96382	0 x 0	1	1	-	-	2	2	-	-	1	55	fails	fails
ex2.285	1 x 121500	0 x 0	1	1	-	-	2	2	-	-	0	0	fails	fails
ex2.304	1053189 x 264079	1052007 x 264079	2	-	-	-	2	-	-	-	463	fails	fails	fails
ex2.423	637916 x 160494	636777 x 160494	*2	-	-	-	*3	-	-	-	*341	fails	fails	fails
ex2.680	757755 x 192803	756940 x 192803	2	-	-	-	2	-	-	-	833	fails	fails	fails

Table 2: Random FSM's.

applications? The examples reported in Table 3 answer in the affirmative the question. They are the from the suite of FSM's described in [46]. It is not possible to build and solve these binate tables with explicit techniques. Instead we can manipulate them with our implicit binate solver and find a solution. In the example *fourr.40*, only the first table reduction was performed.

FSM	table size (rows x columns)		# mincov				# cover				CPU time (sec)			
	before	after first	ISM		STAMINA		ISM		STAMINA		ISM		STAMINA	
	reduction	α reduction	α	β	α	β	α	β	α	β	α	β	α	β
threer.20	6977 x 3936	6974 x 3936	*4	*6	*5	*3	*5	*5	*6	*6	*13	*26	*1996	*677
threer.25	35690 x 17372	34707 x 17016	*3	*6	-	-	*5	*6	-	-	*69	*192	fails	fails
threer.30	68007 x 33064	64311 x 32614	*4	*9	-	-	*8	*8	-	-	*526	*770	fails	fails
threer.35	177124 x 82776	165967 x 82038	*8	*9	-	-	*12	*10	-	-	*2296	*2908	fails	fails
threer.40	1209783 x 529420	1148715 x 526753	*8	-	-	-	*12	-	-	-	*6787	fails	fails	fails
fourr.16	6060 x 3266	5235 x 3162	*2	*3	*3	*3	*3	*3	*4	*4	*6	*23	*1641	*513
fourr.16	6060 x 3266	5235 x 3162	*2	623	*3	377	*3	3	*4	3	*6	9194	*1641	1459
fourr.20	26905 x 12762	26904 x 12762	*2	*4	-	-	*4	*4	-	-	*31	*68	fails	fails
fourr.30	1396435 x 542608	1385809 x 542132	*2	*5	-	-	*4	*5	-	-	*1230	*1279	fails	fails
fourr.40	6.783e9 x 2.388e9	6.783e9 x 2.388e9	†1	-	-	-	†-	-	-	-	†723	fails	fails	fails

Table 3: Learning I/O sequences benchmark.

11.1.4 Minimizing FSM's from Synthesis of Interacting FSM's

Prime compatibles are required only for the state minimization of $ifsm1$ and $ifsm2$. For $ifsm1$, ISM can find a first solution faster than STAMINA using α-dominance. But as the table sizes are not very big, the run times ISM take are usually longer than those for STAMINA.

FSM	table size (rows x columns)		# mincov				# cover				CPU time (sec)			
	before	after first	ISM		STAMINA		ISM		STAMINA		ISM		STAMINA	
	reduction	α reduction	α	β	α	β	α	β	α	β	α	β	α	β
ifsm1	17663 x 8925	16764 x 8829	*4	2	*10	3	*14	14	*15	14	*388	864	*17582	805
ifsm1	17663 x 8925	16764 x 8829	*4	2	24	3	*14	14	14	14	*388	864	40817	805
ifsm2	1505 x 774	1368 x 672	4	3	41	44	9	9	9	9	136	230	49	3

Table 4: Examples from synthesis of interactive FSM's.

11.2 Selection of Generalized Prime Implicants

Tables 5 and 6 report the results of running our program ISA to select a minimal encodeable cover of generalized prime implicants (GPI's). GPI's are an extension of the concept of prime implicants to the case of multi-valued input and multi-valued output Boolean functions. An encodeable selection of GPI's translates into a two-valued implementation of the same size. Details can be found in [19, 57]. For these experiments ISA has been run with option $-m$, that computes a subset of the GPI's, to generate smaller tables. The tables provide the following information:

- Under the column "table size" we provide the dimensions of the original table and of its cyclic core, i.e., the dimensions of the table obtained when the first cycle of reductions converges.
- The column "mincov calls" is the number of recursive calls of the implicit table solver.
- The column "table sol." is the cardinality of the cover of GPI's returned by the table solver.
- The column "CPU time table red." gives the time for the binate table solver. The time to compute the prime compatibles is not included.

The part of ISA that computes an encodeable cover of GPI's and gets the codes by a second call to an implicit table solver is not reported here.

Out of the examples in Table 5, ISA fails to complete the first table reduction of *slave* because of timeout at 18000 seconds, during collapse columns. Ouf of the examples in Table 6, ISA fails to complete some of them, again due to timeout or no more memory in the collapse column step of the first table reduction. FSM's *cse, dk512, keyb, ex2, maincont, pkheader, markl* were run on a DEC 7000 Model 610 AXP with 1Gb of memory. There is no program against which to compare.

We underline that the covering problems faced to select covers of GPI's, even though they are unate, are often harder than those encountered to select covers of prime implicants in the ESPRESSO benchmark [24, 8], a reason being the larger variable support of the BDD representations of columns and rows. To be able to solve the examples of the previous tables, the package described in [31] had to be further optimized and inadequacies still remain to be addressed.

References

[1] K. Brace, R. Rudell, and R. Bryant. Efficient implementation of a BDD package. In *The Proceedings of the Design Automation Conference*, pages 40–45, June 1990.

[2] R. Brayton, G. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, 1984.

[3] R. Brayton, A. Sangiovanni-Vincentelli, and G. Hachtel. Multi-level logic synthesis. *The Proceedings of the IEEE*, february 1990.

[4] R. Brayton, A. Sangiovanni-Vincentelli, G. Hachtel, and R. Rudell. *Multi-level logic synthesis*. Unpublished book, 1992.

[5] R. Brayton and F. Somenzi. An exact minimizer for Boolean relations. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 316–319, November 1989.

[6] R. Bryant. Graph based algorithm for Boolean function manipulation. In *IEEE Transactions on Computers*, pages C–35(8):667–691, 1986.

FSM	table size (row x col) before red.	after red.	mincov calls	table sol.	CPU time (sec.) table red.
bbara	187 x 4124	98 x 35	9	8	329
bbtas	28 x 107	9 x 6	3	4	3
beecount	153 x 176	0 x 0	1	6	44
chanstb	169216 x 525	0 x 0	1	11	1218
cpab	208896 x 1892	683 x 73	4	8	7774
dk14	157 x 199	0 x 0	1	17	129
dk15	88 x 68	0 x 0	1	14	9
dk17	64 x 164	0 x 0	1	9	46
dk27	20 x 71	0 x 0	1	4	5
dol2	20 x 113	19 x 25	2	2	8
es	23 x 45	0 x 0	1	5	1
ex3	42 x 495	0 x 0	1	5	563
ex5	50 x 301	0 x 0	1	3	139
ex6	908 x 423	0 x 0	1	22	645
ex7	48 x 583	0 x 0	1	4	106
fstate	5360 x 1605	11 x 11	2	8	12770
leoncino	21 x 22	0 x 0	1	5	0
lion	25 x 29	0 x 0	1	4	0
lion9	42 x 175	0 x 0	1	2	10
mc	96 x 71	0 x 0	1	7	5
ofsync	300 x 97	48 x 24	18	12	69
opus	914 x 2830	0 x 0	1	14	704
s8	40 x 206	0 x 0	1	1	8
scud	2966 x 2533	0 x 0	1	57	15633
shiftreg	24 x 89	8 x 6	5	3	6
slave	2207744 x 16845	-[a]	-	-	timeout
tav	100 x 81	4 x 4	5	10	10
test	8 x 5	0 x 0	1	3	0
virmach	4992 x 144	0 x 0	1	16	778

[a] timeout 18000 in collapse columns

Table 5: Selection of a minimal encodeable GPI cover

FSM	table size (row x col) before red.	after red.	mincov calls	table sol.	CPU time (sec.) table red.
bbsse	3480 x 34727	-[a]	-	-	timeout
cf	30208 x 102781	-[b]	-	-	-
cse	2588 x 21798	0 x 0	1	23	6534
dk512	43 x 1777	0 x 0	1	6	4150
ex2	86 x 38410	0 x 0	1	3	830
ex4	1072 x 26759	0 x 0	1	10	803
keyb	2666 x 361240	0 x 0	1	8	1706
kirkman	100252 x 1081088	-[a]	-	-	timeout
maincont	67586 x 245784	0 x 0	1	4	115
mark1	1936 x 50258	5 x 5	3	7	1313
modulo12	24 x 9039	24 x 36	17	2	50
pkheader	140288 x 29099	0 x 0	1	19	5850
ricks	31232 x 16561	14 x 14	18	27	3301
s1	15336 x 586240	-[b]	-	-	-
s1a	5120 x 586240	-[b]	-	-	-
saucier	18496 x 7106239	0 x 0	1	15	6802
tma	2028 x 287558	-[b]	-	-	-
train11	43 x 583	0 x 0	1	2	177

[a] timeout 18000 in collapse columns
[b] out-of-memory in collapse columns

Table 6: Selection of a minimal encodeable GPI cover

[7] E. Cerny. Characteristic functions in multivalued logic systems. *Digital Processes*, vol. 6:167–174, June 1980.

[8] O. Coudert. Two-level logic minimization: an overview. *Integration*, 17-2:97–140, October 1994.

[9] O. Coudert. On solving binate covering problems. *Manuscript*, May 1995.

[10] O. Coudert, C. Berthet, and J. C. Madre. Verification of sequential machines using functional Boolean vectors. *IFIP Conference*, November 1989.

[11] O. Coudert, H.Fraisse, and J.C. Madre. Towards a symbolic logic minimization algorithm. In *The Proceedings of the VLSI Design 1993 Conference*, pages 329–334, January 1993.

[12] O. Coudert and J.C. Madre. Implicit and incremental computation of prime and essential prime implicants of Boolean functions. In *The Proceedings of the Design Automation Conference*, pages 36–39, June 1992.

[13] O. Coudert and J.C. Madre. A new implicit graph based prime and essential prime computation technique. In *Proceedings of the International Symposium on Information Sciences*, pages 124–131, July 1992.

[14] O. Coudert and J.C. Madre. A new method to compute prime and essential prime implicants of boolean functions. In *Advanced Research in VLSI and Parallel Systems*, pages 113–128. The MIT Press, T. Knight and J. Savage Editors, March 1992.

[15] O. Coudert and J.C. Madre. A new viewpoint on two-level logic minimization. *Bull Research Report N. 92026*, November 1992.

[16] O. Coudert and J.C. Madre. New ideas for solving covering problems. In *The Proceedings of the Design Automation Conference*, pages 641–646, June 1995.

[17] O. Coudert, J.C. Madre, and H.Fraisse. A new viewpoint on two-level logic minimization. In *The Proceedings of the Design Automation Conference*, pages 625–630, June 1993.

[18] O. Coudert, J.C. Madre, H.Fraisse, and H. Touati. Implicit prime cover computation: an overview. In *The Proceedings of the SASIMI Conference*, pages 413–422, 1993.

[19] S. Devadas and R. Newton. Exact algorithms for output encoding, state assignment and four-level Boolean minimization. *IEEE Transactions on Computer-Aided Design*, pages 13–27, January 1991.

[20] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. W. H. Freeman and Company, 1979.

[21] J. Gimpel. A reduction technique for prime implicant tables. *IRE Transactions on Electronic Computers*, EC-14:535–541, August 1965.

[22] A. Grasselli and F. Luccio. A method for minimizing the number of internal states in incompletely specified sequential networks. *IRE Transactions on Electronic Computers*, EC-14(3):350–359, June 1965.

[23] A. Grasselli and F. Luccio. Some covering problems in switching theory. In *Networks and Switching Theory*, pages 536–557. Academic Press, New York, 1968.

[24] G.Swamy, R.Brayton, and P.McGeer. A fully implicit Quine-McCluskey procedure using BDD's. *Tech. Report No. UCB/ERL M92/127*, 1992.

[25] R. W. House and D.W. Stevens. A new rule for reducing cc tables. *IEEE Transactions on Computers*, C-19:1108–1111, November 1970.

[26] S. Robinson III and R. House. Gimpel's reduction technique extended to the covering problem with costs. *IRE Transactions on Electronic Computers*, EC-16:509–514, August 1967.

[27] S.-W. Jeong and F. Somenzi. A new algorithm for 0-1 programming based on binary decision diagrams. In *Proceedings of ISKIT-92, International symposium on logic synthesis and microprocessor architecture, Iizuka, Japan*, pages 177–184, July 1992.

[28] S.-W. Jeong and F. Somenzi. A new algorithm for the binate covering problem and its application to the minimization of boolean relations. In *The Proceedings of the International Conference on Computer-Aided Design*, November 1992.

[29] T. Kam. *State Minimization of Finite State Machines using Implicit Techniques*. PhD thesis, U.C. Berkeley, Electronics Research Laboratory, University of California at Berkeley, May 1995.

[30] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. A fully implicit algorithm for exact state minimization. *Tech. Report No. UCB/ERL M93/79*, November 1993.

[31] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. A fully implicit algorithm for exact state minimization. In *The Proceedings of the Design Automation Conference*, pages 684–690, June 1994.

[32] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. Implicit state minimization of non-deterministic fsm's. In *The Proceedings of the International Conference on Computer Design*, October 1995.

[33] Y.-T. Lai, M. Pedram, and S.B.K. Vrudhula. FGILP: An integer linear program solver based on function graphs. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 685–689, November 1993.

[34] Y.-T. Lai, M. Pedram, and S.B.K. Vrudhula. EVBDD-based algorithms for integer linear programming, spectral transformation, and function decomposition. *IEEE Transactions on Computer-Aided Design*, pages CAD–13(8):959–975, August 1994.

[35] L. Lavagno. Heuristic and exact methods for binate covering. *EE290ls Report*, May 1989.

[36] B. Lin. Synthesis of VLSI designs with symbolic techniques. *Tech. Report No. UCB/ERL M91/105*, November 1991.

[37] B. Lin, O. Coudert, and J.C. Madre. Symbolic prime generation for multiple-valued functions. In *The Proceedings of the Design Automation Conference*, pages 40–44, June 1992.

[38] B. Lin and A.R. Newton. Implicit manipulation of equivalence classes using binary decision diagrams. In *The Proceedings of the International Conference on Computer Design*, pages 81–85, September 1991.

[39] B. Lin and F. Somenzi. Minimization of symbolic relations. In *The Proceedings of the International Conference on Computer-Aided Design*, November 1990.

[40] B. Lin and F. Somenzi. Minimization of symbolic relations. In *The Proceedings of the International Conference on Computer-Aided Design*, November 1990.

[41] H.-J. Mathony. Universal logic design algorithm and its application to the synthesis of two-level switching circuits. *IEE Proceedings*, pages 171–177, May 1989.

[42] E. McCluskey. Minimization of Boolean functions. *Bell Laboratories Technical Journal*, November 1956.

[43] P. McGeer, J. Sanghavi, R. Brayton, and A. Sangiovanni-Vincenetelli. Espresso-signature: a new exact minimizer for logic functions. *IEEE Transactions on VLSI Systems*, pages 432–440, December 1993.

[44] S. Minato. Zero-suppressed BDD's for set manipulation in combinatorial problems. In *The Proceedings of the Design Automation Conference*, pages 272–277, June 1993.

[45] L. Nguyen, M. Perkowski, and N. Goldstein. Palmini - fast boolean minimizer for personal computers. In *The Proceedings of the Design Automation Conference*, pages 615–621, July 1987.

[46] Arlindo L. Oliveira and Stephen A. Edwards. Inference of state machines from examples of behavior. Technical report, UCB/ERL Technical Report M95/12, Berkeley, CA, 1995.

[47] C. H. Papadimitriou, J.D. Ullman, and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice Hall, 1982.

[48] J.-K. Rho and F. Somenzi. Stamina. *Computer Program*, 1991.

[49] R. Rudell. Espresso. *Computer Program*, 1987.

[50] R. Rudell. Logic synthesis for VLSI design. *Tech. Report No. UCB/ERL M89/49*, April 1989.

[51] A. Saldanha, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. A uniform framework for satisfying input and output encoding constraints. *The Proceedings of the Design Automation Conference*, June 1991.

[52] S.C. De Sarkar, A.K. Basu, and A.K. Choudhury. Simplification of incompletely specified flow tables with the help of prime closed sets. *IEEE Transactions on Computers*, pages 953–956, October 1969.

[53] M. Servit and J. Zamazal. Exact approaches to binate covering problem. *Manuscript*, October 1992.

[54] F. Somenzi. Cookie. *Computer Program*, 1989.

[55] F. Somenzi. Gimpel's reduction technique extended to the binate covering problem. *Unpublished manuscript*, 1989.

[56] H. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using BDD's. *The Proceedings of the International Conference on Computer-Aided Design*, pages 130–133, November 1990.

[57] T. Villa. *Encoding Problems in Logic Synthesis*. PhD thesis, University of California, Berkeley, May 1995.

[58] Ming Huei Young and S. Muroga. Symmetric minimal covering problem and minimal PLA's with symmetric variables. *IEEE Transactions on Computers*, C-34:523–541, June 1985.