Chapter 1

# AURA II: COMBINING NEGATIVE THINKING AND BRANCH-AND-BOUND IN UNATE COVERING PROBLEMS


Luca P. Carloni
*EECS Department, University of California at Berkeley, Berkeley, CA 94720*

Evguenii I. Goldberg
*Cadence Berkeley Laboratories, Berkeley, CA 94704*

Tiziano Villa
*PARADES, 00186 Roma*

Robert K. Brayton
*EECS Department, University of California at Berkeley, Berkeley, CA 94720*

Alberto L. Sangiovanni-Vincentelli
*EECS Department, University of California at Berkeley, Berkeley, CA 94720*



**Abstract** Recently a novel technique has been published to augment traditional Branch-and-Bound (B&B) while solving exactly a discrete optimization problem [Goldberg et al., 1997]. This technique is based on the *negative thinking* paradigm and has been applied to develop AURA, a Unate Covering Problem (UCP) solver which reportedly was able to deal efficiently with some time-consuming benchmark problems. However, on average AURA was not able to compete with SCHERZO, a classical UCP solver based on several new bounding techniques proposed by O. Coudert in his breakthrough paper [Coudert, 1996]. This fact left open the question on the practical impact of the negative thinking paradigm. The present work is meant to settle this question. The paper discusses the details of AURA II, a new implementation of the negative thinking paradigm for UCP which combines the best of SCHERZO and AURA. Experimental results show the dramatic impact of the negative thinking paradigm in searching the solution space and propose AURA II as the most efficient available tool for unate covering.

**Keywords:** Combinatioral optimization, branch-and-bound, covering problem.

# 1. INTRODUCTION

The Unate Covering Problem (UCP) [Kam et al., 1997] occurs often in logic synthesis and operations research and is defined as:

- Given a Boolean matrix $A$ (all entries are 0 or 1), with $m$ rows, denoted as $Row(A)$, and $n$ columns, denoted as $Col(A)$, and a cost vector $c$ of the columns of $A$ ($c_i$ is the cost of the $i$-th column), minimize the cost $x^T c = \sum_{j=1}^{n} x_j c_j$, over all $x \in \{0,1\}^n$, subject to $A\, x \geq (1, 1, \cdots, 1)^T$.

Informally the minimum unate covering problem requires to find a set of columns of minimum cost, such that each row intersects - "is covered by" - at least once a column in the set (i.e., the entry at the intersection is a 1). For simplicity assume that all columns have the same cost. An instance of UCP with matrix $A$ is denoted $UCP(A)$.

In [Goldberg et al., 1997] the authors applied to UCP a novel technique to augment Branch-and-Bound (B&B) by a new way of exploring solutions, inspired by a paradigm called negative thinking. An algorithm named *raiser* realizing negative thinking by means of incremental problem solving was implemented in a computer program called AURA. This paper discusses the details of the *raiser* algorithm and reports the results obtained with AURA II, a new UCP solver which combines the best techniques of the traditional B&B with the negative thinking paradigm.

An exact solution of UCP may be obtained by a B&B recursive algorithm, variants of which have been implemented in successful computer programs [Coudert, 1994, Coudert, 1996, Coudert and Madre, 1995, Rudell and Sangiovanni-Vincentelli, 1987]. Branching is done by columns, i.e., subproblems are generated by considering whether a chosen branching column is or is not in the solution. A run of the algorithm, say *mincov*, can be described by a computation tree, where the root is the input of the problem, an edge represents a call to *mincov* and 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, which is the current path for that node. The path leading to the node gives a partial solution and a submatrix $A_N$ obtained from $A$ by removing some rows and columns. On the path some columns are included in the partial solution and they are denoted by $path(A_N)$.

Suppose that we know that any minimal cover of $A_N$ is greater or equal to a value $L(A_N)$. The value is called a lower bound of the solutions of $UCP(A_N)$; e.g., a Maximal Set of Independent Rows (MSIR) is a lower bound (independent means that they have at most 1 one per column). So the size of any solution of $UCP(A)$ including the columns in $path(A_N)$ is greater or equal to $L(A_N)$ + $|path(A_N)|$. Hence, if we found before a solution $best$ with the same or a smaller number of columns, i.e., $|best| \leq L(A_N) + path(A_N)$ we can stop the recursion and backtrack to the parent node of $A_N$. Denote by $K(A_N)$ the value $|best| - L(A_N) - |path(A_N)|$. The condition to stop the recursion is given by $K(A_N) \leq 0$. On the other hand, if $K(A_N)$ has a large positive value, usually it means that $L(A_N)$ is far from the size of a minimal solution to $UCP(A_N)$ and so a lot of branching is expected from $A_N$ before a leaf can be reached.

Suppose that there is no way of improving the solution $best$ in the search tree rooted at $A_N$, yet $K(A_N)$ is positive. Usually a B&B algorithm must continue branching. However, there is another way of making $K(A_N)$ negative or zero: it is to improve the lower bound $L(A_N)$. The first way is "*positive*", in the sense that the algorithm tries to construct a better solution, and branching columns are chosen in the hope of improving the current best solution. The second way is "*negative*", in the sense that the algorithm tries to prove that there is no better solution in the tree rooted at $A_N$. Often in the first leaf a solution very close to a minimum one is found, so only few improvements are required to get a minimum solution. Therefore "positive" search will succeed and yield a new better solution only in a few of the potential $2^n$ subproblems at the $n$-th level of the computation tree. In the overwhelming majority of the subproblems "negative" search is more natural. The less frequently the best current solution is improved during the search, the more the "negative" search is justified.

To exploit both "positive" and "negative" search, B&B was modified in [Goldberg et al., 1997] as follows: start solving the initial problem with "positive thinking" in the ordinary column branching mode, called PT-mode. Then, when the number of subproblems generated in the column branching mode becomes large "enough", solve each subproblem in the "negative thinking" mode, called NT-mode. Modes are switched depending on the ratio of the expected number of improvements to the number of subproblems generated at this level of the search tree. The smaller the ratio, the more appropriate it is to switch to the NT-mode.

In [Goldberg et al., 1997] the results of comparing AURA against ESPRESSO [Rudell and Sangiovanni-Vincentelli, 1987] and SCHERZO [Coudert, 1994, Coudert, 1996, Coudert and Madre, 1995] were reported. AURA could outperform ESPRESSO on every benchmark, but was not always able to beat the performance of SCHERZO, due to its improvements in the computation of the lower bounds; partition-based pruning and further modifications in the organization of the B&B scheme. In principle, these features are orthogonal to the introduction of the negative thinking paradigm.

To assess further the strength of *raiser*, an approach (only partially explored in [Goldberg et al., 1997]) would have been to reproduce systematically all the features of SCHERZO within AURA. This paper reports the results of the alternative choice to re-implement *raiser* on top of SCHERZO, yielding the program AURA II, in order to exploit the algorithmic and programming virtues of SCHERZO together with the power of negative thinking available through *raiser*. In Section 3. the results of this comparison are reported, showing that AURA II is faster than SCHERZO, especially in the most time-consuming examples. As far as we know AURA II is currently the most efficient available tool for unate covering. AURA II combines the best of both worlds, and settles some experimental questions left open in [Goldberg et al., 1997].

## 2. THE RAISING ALGORITHM

Figure 1.1 shows how the traditional branch-and-bound algorithm *mincov* [Villa et al., 1997a] is modified to incorporate the technique of incrementally raising the lower bound. After the computation of the lower bound, if the gap *difference* between the upper and lower bound is small, i.e., less than a global parameter $maxRaiser$, a new procedure $raiser$ is invoked with a parameter $n$ set to the value of *difference*. The parameter $maxRaiser$ currently is decided a-priori, but ideally it should be adapted dynamically. Intuitively if the gap is small, we conjecture that a search in this subtree will not improve the best solution and so we trigger the procedure $raiser$ that may either confirm the conjecture and prove that no better solution can be found here or disprove the conjecture and improve the best solution, updating the current one.

### 2.1 RAISING ALGORITHM: OVERVIEW

As discussed in [Goldberg et al., 1997] we developed an $n$-*raiser* procedure, based on row branching. Given a covering matrix $A$, let $A'$ be a submatrix of $A$ and $A_p$ a row from $Row(A) \setminus Row(A')$. Let $S$ be a solution of $UCP(A')$. Denote by $O(A_p)$ the set $\{j \mid A_{pj} = 1\}$, i.e., the set of all columns covering $A_p$ and by $Rec(A' + A_p, S)$ a set of solutions of $UCP(A' + A_p)$ obtained according to the following rules:

1. if $S$ is a solution of $UCP(A' + A_p)$, then $Rec(A' + A_p, S) = \{S\}$;
2. if $S$ is not a solution of $UCP(A' + A_p)$, i.e., no column of $S$ covers $A_p$ then $Rec(A' + A_p, S) = \{S \cup \{j\} \mid j \in O(A_p)\}$.

So $Rec(A' + A_p, S)$ gives the solutions of $UCP(A' + A_p)$ that can be obtained from the solution $S$ of $UCP(A')$. According to 2., if $S$ is not a solution of $UCP(A' + A_p)$, then we obtain $|O(A_p)|$ solutions of $UCP(A' + A_p)$ by adding to $S$ the columns covering $A_p$.

As discussed in [Goldberg et al., 1997], we represent the solutions of $UCP(A)$ by sets with a structure of multi-valued cubes [Rudell and Sangiovanni-Vincentelli, 1987]. We define a *cube* to be the set $C = D_1 \times \cdots \times D_d$ where

$AuraMincov(A, path, weight, lbound, ubound)$ {
/* Apply row dominance, column dominance, and select essentials */ (1)
if (not $reduce(A, path, weight, ubound)$) return $empty\_solution$
/* See if Gimpel's reduction technique applies */ (2)
if ($gimpel\_reduce(A, path, weight, lbound, ubound, best)$) return $best$
/* Find lower bound from here to final solution by independent set */ (3)
$MSIR = maximal\_independent\_set(A, weight)$
/* Make sure the lower bound is monotonically increasing */ (4)
$lbound\_new = max(cost(path) + cost(MSIR), lbound)$
$\boldsymbol{difference = ubound - lbound\_new}$
**/* Bounding based on no better solution possible */ (5)**
**if** $(\boldsymbol{difference \leq 0})$ $\boldsymbol{best = empty\_solution}$
**else if** $(\boldsymbol{difference \leq maxRaiser})$ { **/* Apply raiser with** $\boldsymbol{n = difference}$ ***/ (16)**
$\boldsymbol{SolCube = cover\_MSIR(MSIR)}$ **(17)**
$\boldsymbol{lowerBound = |SolCube|}$ **(18)**
$\boldsymbol{a = raiser(SolCube, difference, A, lowerBound, bestSol, ubound)}$ **(19)**
**if** $(\boldsymbol{a = 1})$ $\boldsymbol{best = empty\_solution}$ **(20)**
**else** $\boldsymbol{best = path \cup bestSol}$ **/* (answer = 0) */ (21)**
}
else if ($A$ is empty) { /* New best solution at current level */ (6)
$best = solution\_dup(path)$
} else if ($block\_partition(A, A_1, A_2)$ gives non-trivial bi-partitions) { (7)
$path1 = empty\_solution$
$best1 = mincov(A_1, path1, weight, 0, ubound - cost(path))$ (8)
/* Add best solution to the selected set */ (9)
if ($best1 = empty\_solution$) $best = empty\_solution$
else { (10)
$path = path \cup best1$
$best = mincov(A_2, path, weight, lbound\_new, ubound)$
}
} else { /* Branch on cyclic core and recur */ (11)
$branch = select\_column(A, weight, MSIR)$
$path1 = solution\_dup(path) \cup branch$
let $A_{branch}$ be the reduced table assuming $branch$ in solution (12)
$best1 = mincov(A_{branch}, path1, weight, lbound\_new, ubound)$
/* Update the upper bound if we found a better solution */ (13)
if ($best1 \neq empty\_solution$) /* It implies ($ubound > cost(best1)$) */
$ubound = cost(best1)$
/* Do not branch if lower bound matched */ (14)
if ($best1 \neq empty\_solution$) and ($cost(best1) = lbound\_new$) return $best1$
let $A_{\overline{branch}}$ be the reduced table assuming $branch$ not in solution (15)
$best2 = mincov(A_{\overline{branch}}, path, weight, lbound\_new, ubound)$
$best = best\_solution(best1, best2)$
}
return $best$
}

*Figure 1.1* AuraMincov: Traditional *mincov* algorithm enhanced by incremental raising.

$D_i \cap D_j = \emptyset$, $i \neq j$ and $D_i \subset Col(A)$, $1 \leq i, j \leq d$. The subsets $D_i$ are the *domains* of cube $C$. So cube $C$ denotes a set of sets consisting of $d$ columns.

Let $A'$ be a submatrix of $A$. The set of all irredundant (and minimum) solutions of $UCP(A')$ can be represented as the cube $O(A_{i_1}) \times \cdots \times O(A_{i_d})$, where $A_{i_1}, \cdots, A_{i_d}$ are the rows forming $A'$. Let $C = D_1 \times \cdots \times D_d$ be a cube of solutions of $UCP(A')$. Then, choose a "good" row $A_p$ from from

$Row(A) \setminus Row(A')$. From the definition of the $Rec$ operator [1] it follows that

$$Rec(A' + A_p, C) \;=\; part1(C) \;\cup\; part2(C) \times O(A_p) \qquad (1.1)$$

where $part1(C)$ is the set of solutions contained in $C$ which cover $A_p$ and $part2(C)$ is the set of solutions contained in $C$ which do not cover $A_p$. Hence, $Rec(A' + A_p, C)$ can be represented by $r + 1$ cubes where $r$ is the number of rows of the $MSIR(A)$ intersecting $A_p$. Then, perform recursively the process for each of the $r + 1$ cubes, i.e., choose a new row from those not yet selected for each of the $r + 1$ cubes of solutions and split each cube according to the rule explained in [Goldberg et al., 1997].

The entire process can be described by a search tree, called **cube branching tree**. The initial cube of solutions $C$ corresponds to the root node, to which we associate also a pair of matrices $MSIR(A)$ and $A\ MSIR(A)$. In each node a choice of an unselected row from the second matrix of the node is made. The chosen row is removed from the second matrix of the pair and added to the first matrix of the pair. The number of branches leaving a node is equal to the number of cubes in which the cube corresponding to the node is partitioned by the $Rec$ operation, and each child of a node gets one of the cubes obtained after splitting. So the cube corresponding to a node represents a set of solutions covering the first matrix of the pair (that is a "lower bound submatrix" for the node).

Some useful facts are:

- When applying an $n$-*raiser*, the branches corresponding to cubes of more than $|MSIR(A)| + n$ domains are pruned.

- If at a node, a row $A_p$ is chosen such that no solution from the cube $C$ of the node covers $A_p$, then there is no splitting of the cube, since $Rec$ yields only one cube $C \times [O(A_p) \setminus (D_1 \cup \cdots \cup D_d)]$.

- At each node, the following reduction rule can be applied to the second matrix of the pair: if a row of the second matrix is covered by every solution of the cube $C$ corresponding to the node, then the row can be removed from the matrix since, if we add it to the lower bound submatrix of the pair, then the recomputed cube will be equal to $C$.

The recursion terminates if one of the two following conditions hold:

1. There is a node such that there are no rows left in the second matrix of the pair and the corresponding cube has $k$ domains, where $k < |MSIR| + n$. This means that the lower bound $|MSIR|$ cannot be improved by $n$.

[1] With the natural extension that $Rec(A, C) = \bigcup_{c \in C} Rec(A, c)$.

Any solution from the cube can be taken as the best current solution of $UCP(A)$.

2. From all branches, nodes are reached corresponding to cubes with a number of domains greater than $|MSIR| + n$. In this case the lower bound has been raised to $|MSIR| + n$, since no solution $S$ of $UCP(A)$ exists such that $|S| \leq |MSIR| + n$.

The correctness of the $n$-raiser procedure, applied to matrix $A$ with lower bound $|MSIR(A)|$, has been proved in [Goldberg et al., 1997].

## 2.2 RAISING ALGORITHM: IMPLEMENTATION

The procedure *raiser* returns 1 if the lower bound can be raised by $n$, otherwise it returns 0, which means that the current best solution has been improved at least once by *raiser*. The following parameters are needed:

- $A$ is the matrix of rows not yet considered. Initially $A = A' \setminus MSIR$, where $A'$ is the covering matrix at the node (of the column branching tree) that called *raiser*, and $MSIR$ is the maximal independent set of rows, found at the node (of the column branching tree) that called *raiser*. Hence, $A'$ is the covering matrix related to the subproblem obtained by choosing the columns in the path from the root to the node that called *raiser*. The set of chosen columns is denoted by $path$.

- $SolCube$ is a cube which encodes a set of partial solutions of the covering matrix $A'$. Initially $SolCube$ is equal to the set of solutions covering the $MSIR$.

- $n$ is number by which the lower bound $lbound$ must be raised. $n$ is an input-output parameter initially equal to $ubound - |MSIR| - |path|$, which is decreased if *raiser* decreases the best current solution.

- $lbound$ is an input parameter for $raiser$ equal to $|MSIR|$. Notice that $lbound$ differs from the original lower bound [2] by a quantity equal to $|path|$, for consistency with the previous definition of $n$.

- $ubound$ is the cardinality of the best solution known at the time of the current call of *raiser*.

- $bestSolution$ is an output parameter which contains the new best solution found by $raiser$, if the lower bound could not be raised by $n$.

[2] $lbound\_new = |MSIR| + |path|$.

$raiser(SolCube, n, A, lbound, bestSolution, ubound)$ {
    /* returns 1 if solutions in $SolCube$ raise lower bound of $A$ by $n$ */
    $stillToRaise = lbound + n - number\_domains(SolCube)$
    if $(stillToRaise \leq 0)$ return **1**
    /* If $A = \emptyset$ then $path$ + solutions of $A$ in $SolCube$ beats upper bound */
    if $(A = \emptyset)$ return $found\_solution(SolCube, n, bestSolution, ubound)$
    /* consider rows of $A$ not covered by any solution from $SolCube$ */
    $\mathcal{BSONIR} = find\_best\_set\_of\_non\_intersecting\_rows(A, SolCube)$
    foreach row $r_i \in \mathcal{BSONIR}$ {
        /* add a new domain for the columns covering $r_i \in A$ */
        $SolCube = add\_domain(SolCube, A, r_i)$
        $stillToRaise = stillToRaise - 1$
        if $(stillToRaise \leq 0)$ return **1**
    }
    /* Remove the covered rows from $A$ and check again if $A$ is empty */
    $A = A \setminus \mathcal{BSONIR}$
    if $(A = \emptyset)$ return $found\_solution(SolCube, n, bestSolution, ubound)$
    if $(stillToRaise = 1)$ {
        /* Cover with $SolCube$ and remove from $A$ the 1-intersecting rows */
        /* If 2 rows intersect 2 different cols in the same domain, prune the branch */
        if $(add\_set\_of\_1intersecting\_rows(A, SolCube) = 1)$ return **1**
        if $(A = \emptyset)$
            return $found\_solution(SolCube, n, bestSolution, ubound)$
    }
    /* select next "best" row to be covered with $SolCube$ and remove it from $A$ */
    $r_i = select\_best\_uncovered\_row(A, SolCube)$
    $A = A \setminus \{r_i\}$
    /* Splitting: $part1 = \{SolCube_1, \cdots, SolCube_k\}; part2 = \{SolCube_{k+1}\}$ */
    $split\_cubes(SolCube, A, r_i, part1, part2)$
    /* add to $SolCube_2 \in part2$ new domain of the columns covering $r_i$ */
    $SolCube_{k+1} = add\_domain(SolCube_{k+1}, A, r_i)$
    /* branching on cubes of $part1$ and $part2$ */
    $returnValue = \mathbf{1}$
    while $(part1 \cup part2 \neq \emptyset)$ {
        /* select first cubes from $part1$, then cube from $part2$ */
        $SolCube_j = get\_next\_cube(part1 \cup part2)$
        /* if a better global solution has been found set $returnValue$ to **0** */
        if $(raiser(SolCube_j, n, A, lbound, bestSolution, ubound) = \mathbf{0})$
            $returnValue = \mathbf{0}$
    }
    return $returnValue$
}

$found\_solution(SolCube, n, bestSolution, ubound)$ {
    /* extract any solution from $SolCube$ by picking a column from each domain */
    $bestSolution = get\_solution(SolCube)$
    $newUbound = cost(bestSolution)$
    $newN = n - (ubound - newUbound)$
    $n = newN$
    $ubound = newUbound$
    return **0**
}

*Figure 1.2* Algorithm to raise the lower bound.

Fig. 1.2 shows the flow of $raiser$, the procedure that attempts to raise the lower bound of $A$. Notice that it requires a routine *split_cubes* which, for a selection of a row $r_i$ covered by $k$ of the $d$ domains of $SolCube$, partitions $SolCube$ in $k+1$ disjoint cubes, each of $d$ domains; so $part1$ has $k$ cubes of solutions from $SolCube$ covering $r_i$, whereas $part2$ has one cube of solutions from $SolCube$ not covering $r_i$. The number of domains of $SolCube$ is computed by *number_domains*.

$raiser$ is a recursive procedure which starts by handling two terminal cases. The first one occurs when the variable $stillToRaise$ [3], which measures the gap between the upper bound and the current lower bound, is less or equal to zero. If so, we know that the solutions in $SolCube$ raise the lower bound of $A$ by at least $n$, so that no solutions of $A$ can beat the current upper bound. The second terminal case occurs when, after some recursive calls, $A$ has become empty, and so any solution obtained as the union of a solution of $A$ in $SolCube$ together with the columns in the current $path$ is the new best solution.

After these preliminary checks, $find\_best\_set\_of\_non\_intersecting\_rows$ is called. This routine, reported in Figure 1.3, implements a fast heuristic to find a good subset of rows of $A$ which do not intersect any domain of $SolCube$ and which do not intersect each other. Ideally, we would like to get the best $\mathcal{BSONIR}$ set, which is a sort of "maximum set of independent rows" related to $SolCube$, but this would require the solution of another NP-complete problem. We implemented instead the heuristic to insert first in the set $\mathcal{BSONIR}$ the largest row that intersects neither a domain of $SolCube$ nor a row previously inserted into $\mathcal{BSONIR}$.

Thereafter, since each row $r_i$ in $\mathcal{BSONIR}$ is not covered by any solution encoded in $SolCube$, we must add a new domain to $SolCube$ made by the columns which cover $r_i$. While we are adding these new domains, we keep decreasing the variable $stillToRaise$ and checking if its value becomes equal to zero. Finally, we can remove the set $\mathcal{BSONIR}$ from $A$ because the rows have been covered by the new added domains. Notice that during the first call of $raiser$ the set $\mathcal{BSONIR}$ is empty because $SolCube$ encodes the $MSIR$ and, by definition, every row not in the $MSIR$ must intersect at least one row in the $MSIR$. However, during the following recursive calls of $raiser$ the original domains of $SolCube$ may change, namely decrease in cardinality due to $split\_cubes$ and $add\_set\_of\_1intersecting\_rows$. Hence, at some node of

[3] By definition,

$$
\begin{aligned}
stillToRaise &= lbound + n - numberDomains(SolCube) \\
&= |MSIR| + ubound - |MSIR| - |path| - numberDomains(SolCube) \\
&= ubound - |path| - numberDomains(SolCube)
\end{aligned}
$$

.

```
find_best_set_of_non_intersecting_rows(A, SolCube) {
    /* Heuristic to find best set of rows non intersecting SolCube domains. */
    emptyInterRows = ∅
    bestRow = ∅
    foreach row r ∈ A {
        /* D is the set of SolCube domains intersected by r */
        D = compute_set_of_intersected_domains(SolCube, r)
        if (D = ∅) {
            emptyInterRows = emptyInterRows ∪ r
            if (length(bestRow) < length(r))
                bestRow = r
        }
    }
    /* If every row intersects domains of SolCube then return the empty set */
    if (emptyInterRows = ∅)
        return ∅
    else {
        /* Build BSONIR starting from bestRow */
            BSONIR = ∅
        do {
            BSONIR = BSONIR ∪ bestRow
            emptyInterRows = emptyInterRows \ bestRow
            /* Find the new bestRow within emptyInterRows*/
            foreach row r ∈ emptyInterRows {
                if ((r ∩ BSONIR) ≠ ∅)
                    emptyInterRows = emptyInterRows \ r
                else if (length(bestRow) < length(r))
                    bestRow = r
            }
        } while (emptyInterRows ≠ ∅)
    }
    return BSONIR
}
```

*Figure 1.3* Algorithm to find the best set of rows not intersecting $SolCube$.

the recursion tree, it may happen that a row of $A$ is not covered anymore by any domain of $SolCube$.

After having removed the rows belonging to $\mathcal{BSONIR}$, another optimization step can be applied successively before splitting $SolCube$. If at this point $stillToRaise$ is equal to 1, it means that we have already raised the lower bound by $n-1$. Therefore, if we are forced to add one more domain to $SolCube$, then we can prune the current branch. Hence, a simple condition which leads immediately to pruning is the following: consider two rows $r_1$ and $r_2$ of $A$ which intersect $SolCube$ only in one domain $d = \{c^1, c^2, \cdots, c^l\}$, and suppose that $r_1$ intersects only the column $c^i$, while $r_2$ intersects only the column $c^j$. This fact allows us to prune the current branch because to cover one of the rows we may choose either one of the two distinct columns of the domain. Say w.l.o.g that we cover $r_1$ with $c^i$, then to cover $r_2$ we must use a column which does not belong to any domain of $SolCube$ and so we are forced to add one more domain to $SolCube$, thereby raising the lower bound by $n$.

Figure 1.4 illustrates the procedure $add\_set\_of\_1intersecting\_rows$, which exploits the previous situation and, in practice, is invoked often because the condition $stillToRaise = 1$ happens very commonly in hard problems. Basically, the routine is based on two nested cycles. The external cycle is repeated until the internal cycle does not modify $SolCube$ anymore. The internal cycle computes, for each row $r$ of $A$, the set $D$ of the domains of $SolCube$ intersected by $r$. If the cardinality of $D$ is equal to 1, e.g., $D = \{d\}$, we remove from $d$ all the columns which are not intersected by $r$ and then we remove $r$ from $A$, since $r$ has been covered.

Notice that $add\_set\_of\_1intersecting\_rows$ is called just after having removed from $A$ the set of non-intersecting rows $\mathcal{BSONIR}$ and therefore when all the remaining rows of $A$ intersect at least one domain of $SolCube$. However, after cycling inside this routine and removing some columns (thereby making "leaner" some domains), it is possible that a row of $A$ is not covered anymore, i.e., $|D| = 0$. As discussed above, this happens, e.g., when two 1-intersecting rows intersect two different columns in the same domain $D$. In this case the routine returns $1$ in order to inform the caller to prune the current branch. If this fact does not happen before the end of both cycles, a 0 is returned but, at least a certain number of rows have been removed from $A$ and the corresponding intersected domains of $SolCube$ have been made "leaner". After calling $add\_set\_of\_1intersecting\_rows$ and removing 1-intersecting rows, it is possible that $A$ has become empty. If so, $raiser$ calls $found\_solution$ to update the variables $bestSolution$, $ubound$ and $n$.

After all these special cases have been addressed, we must select a new row $r_i$ to be covered with $SolCube$. The row $r_i$ is removed from $A$ and drives the splitting of $SolCube$. The strategy to select the best row in order to split the current $SolCube$, before calling recursively $raiser$, is to look for the row of $A$ which intersects the minimum number of domains of $SolCube$. The reason is to reduce the number of branches from the node [4]. Notice that at this stage each row of $A$ intersects at least 2 domains of $SolCube$. In case of ties between different rows, the row having the highest weight is chosen. The weight of a row $A_p$ is defined as $\prod_{k=1}^{m} \frac{|D'_{i_k}|}{|D_{i_k}|}$, where $m$ is the number of domains of $SolCube$ intersecting $A_p$, $D_{i_k}$ is a domain intersected by $A_p$ and $D'_{i_k} = D_{i_k} \setminus O(A_p)$. So the weight of $A_p$ is just the fraction of solutions from $SolCube$ that do not cover $A_p$, that is the quantity that we want to maximize when selecting a new

[4]Recall that there is a branch for each domain intersecting the row plus one more branch for the non-intersecting domains.

```
add_set_of_1intersecting_rows(A, SolCube) {
    /* This routine is called only if stillToRaise = 1. It covers */
    /* with SolCube and removes from A the 1-intersecting rows, */
    /* i.e., the rows intersecting only one domain of SolCube. */
    /* If 2 rows intersect 2 different columns in the same domain, */
    /* return 1 to the caller to prune the current branch */
    do {
        reducingDomains = FALSE
        foreach row r ∈ A {
            /* D is the set of SolCube domains intersected by r */
            D = compute_set_of_intersected_domains(SolCube, r)
            if (| D |= 1) {
                reducingDomains = TRUE
                /* Get the domain d of SolCube covering r and */
                /* remove from d all the cols which do not cover r */
                d = get_covering_domain(SolCube, r)
                simplify_domain(d, r)
                /* Remove the covered row r from A */
                A = A \ {r}
            }
            else if (| D |= 0) {
                /* After removing some columns, a row may not be */
                /* covered anymore, so current branch must be pruned. */
            }
            /* else (| D |> 1): do nothing */
            /* because r is not a 1-intersecting row */
        }
    } while (reducingDomains)
    return 0
}
```

*Figure 1.4* Algorithm to handle the 1-intersecting rows.

row. If $D'_{i_k} = \emptyset$, for some $k$, this means that $A_p$ is covered by any solution from $SolCube$. Such a row is simply removed from $A''$ and added to $A'$.

After performing the splitting of $SolCube$ as explained in [Goldberg et al., 1997], $raiser$ is called recursively on the disjoint cubes of the recomputed solution. If the current best solution is not improved in any of the calls, then raiser returns 1, meaning that the lower bound has been raised by $n$. If instead the current best solution has been improved once or more times, $raiser$ returns 0 after having updated the current best solution and upper bound.

## 3. EXPERIMENTAL RESULTS

In [Goldberg et al., 1997], AURA was compared against the routine *mincov* available in ESPRESSO, and against the results of SCHERZO [Coudert, 1994, Coudert, 1996, Coudert and Madre, 1995], the most effective UCP solver available then. Compared to ESPRESSO, SCHERZO features a collection of new lower bounds (easy lower bound, logarithmic lower bound, left hand side lower bound, limit lower bound), and partition-based pruning. In this paper we

| | | | SCHERZO | | AURA II | | | time |
|---|---|---|---|---|---|---|---|---|
| matrix | $R \times C(S\%)$ | Sol. | nodes | time | nodes/A-nodes | time | r | ratio |
| ex5 | $831 \times 2428\ (2)$ | 37 | 614631 | 11397.1 | 614510/156 | 11066.5 | **1** | 0.97 |
| ex5 | $831 \times 2428\ (2)$ | 37 | 614631 | 11397.1 | 31185/243184 | 1346.67 | **2** | 0.12 |
| ex5 | $831 \times 2428\ (2)$ | 37 | 614631 | 11397.1 | 1905/195190 | 746.85 | **3** | 0.06 |
| max1024 | $1090 \times 1264\ (0.5)$ | 245 | 533635 | 5535.67 | 533632/52 | 5244.54 | **1** | 0.95 |
| max1024 | $1090 \times 1264\ (0.5)$ | 245 | 533635 | 5535.67 | 91345/667471 | 2994.88 | **2** | 0.54 |
| max1024 | $1090 \times 1264\ (0.5)$ | 245 | 533635 | 5535.67 | 15353/1624827 | 5967.92 | **3** | 1.10 |
| prom2 | $1924 \times 2611\ (0.3)$ | 278 | 26143 | 1506.75 | 26143/16 | 1454.81 | **1** | 0.97 |
| prom2 | $1924 \times 2611\ (0.3)$ | 278 | 26143 | 1506.75 | 6115/115460 | 1685.36 | **2** | 1.10 |
| prom2 | $1924 \times 2611\ (0.3)$ | 278 | 26143 | 1506.75 | 1389/754564 | 10162 | **3** | 6.70 |
| saucier | $171 \times 6207\ (47)$ | 6 | 187089 | 11876.1 | 7/36 | 24.0 | **1** | 0.002 |
| saucier | $171 \times 6207\ (47)$ | 6 | 187089 | 11876.1 | 7/36 | 24.0 | **2** | 0.002 |
| saucier | $171 \times 6207\ (47)$ | 6 | 187089 | 11876.1 | 7/36 | 24.0 | **3** | 0.002 |

*Table 1.1* Results on *Espresso* benchmarks (SCHERZO VS. AURA II).

compare AURA II, that is $raiser$ implemented in SCHERZO, against SCHERZO. The benchmarks used belong to three classes: Table 1.1 contains difficult cases from the collection of ESPRESSO (we start from the matrix obtained by ESPRESSO after removing the essential primes) and some matrix encoding constraints satisfaction problems from [Villa et al., 1997b]; Table 1.2 contains random generated matrices with varying row/column ratios and densities (e.g., $m$200_100_30_70 means a matrix with 200 rows, 100 columns, and each column having a number of ones between 30 and 70). For each of these matrices, their size ($R \times C$ in the tables) and sparsity ($S$ expressed as a percentage in the tables) are reported. The experiments were performed with a 1GB 625Mhz Alpha with timeout set to 4 hours of cpu time. Tables 1.1 and 1.2 report two types of data for comparison: the number of nodes of the column branching computation tree and the running time. Concerning the number of nodes we clarify the following points:

1. AURA II has two types of nodes: those of the column branching computation tree and those of the cube branching computation tree (called A-nodes in the tables). Indeed AURA II follows a dual strategy: it builds the column branching computation tree, but when at a node the difference between the upper bound and the lower bound is less than or equal to the raising parameter $r$ (or $maxRaiser$), AURA II calls the procedure $raiser$ which builds a cube branching computation tree (appended at the node where $raiser$ was called). So we need to report both numbers of nodes to measure a run of AURA II.

2. Nodes of the cube branching computation tree usually take much less computing time than those of the column branching computation tree, even though a time ratio between the two types of nodes is not known a-priori. The reason is that expensive procedures for finding dominance relations and $MSIR$ are applied in each node of the column branching tree.

| matrix | $R \times C(S\%)$ | Sol. | SCHERZO nodes | SCHERZO time | AURA II nodes/A-nodes | AURA II time | r | time ratio |
|---|---|---|---|---|---|---|---|---|
| m100_100_10_10 | $100 \times 100\ (10)$ | 12 | 95086 | 36.87 | 3180/121892 | 20.33 | 3 | 0.55 |
| m100_100_10_15 | $100 \times 100\ (12)$ | 10 | 10335 | 6.12 | 269/11071 | 2.41 | 3 | 0.39 |
| m100_100_10_30 | $100 \times 100\ (20)$ | 8 | 4618 | 4.05 | 84/2726 | 0.78 | 3 | 0.19 |
| m100_100_30_30 | $100 \times 100\ (30)$ | 5 | 1752 | 2.44 | 49/1288 | 0.64 | 3 | 0.26 |
| m100_100_50_50 | $100 \times 100\ (50)$ | 4 | 4015 | 6.1 | 5/857 | 0.69 | 3 | 0.11 |
| m100_100_70_70 | $100 \times 100\ (70)$ | 3 | 171 | 2.21 | 3/112 | 0.19 | 3 | 0.09 |
| m100_100_90_90 | $100 \times 100\ (90)$ | 2 | 2 | 0.02 | 2/0 | 0.02 | 3 | 1 |
| m100_300_10_10 | $100 \times 293\ (3)$ | 21 | 351183 | 235.16 | 10144/612753 | 175.37 | 3 | 0.75 |
| m100_300_10_14 | $100 \times 297\ (4)$ | 19 | 1906835 | 1257.62 | 70998/3453419 | 993.83 | 3 | 0.79 |
| m100_300_10_15 | $100 \times 297\ (4)$ | 19 | 11596849 | 7066.57 | 329794/16381322 | 4385.16 | 3 | 0.62 |
| m100_300_10_20 | $100 \times 299\ (5)$ | 17 | 5240615 | 3641.41 | 138572/6904928 | 2036.72 | 3 | 0.56 |
| m100_50_10_10 | $100 \times 50\ (20)$ | 8 | 2079 | 0.92 | 85/2411 | 0.42 | 3 | 0.46 |
| m100_50_20_20 | $100 \times 50\ (40)$ | 5 | 1825 | 1.02 | 23/889 | 0.27 | 3 | 0.26 |
| m100_50_30_30 | $100 \times 50\ (60)$ | 3 | 63 | 0.34 | 3/24 | 0.03 | 3 | 0.09 |
| m100_50_40_40 | $100 \times 50\ (80)$ | 2 | 2 | 0.01 | 2/0 | 0.01 | 3 | 1 |
| m50_100_10_10 | $50 \times 99\ (10)$ | 8 | 92 | 0.02 | 12/133 | 0.02 | 3 | 1 |
| m50_100_30_30 | $50 \times 100\ (30)$ | 4 | 65 | 0.06 | 5/61 | 0.02 | 3 | 0.33 |
| m50_100_50_50 | $50 \times 100\ (50)$ | 3 | 107 | 0.22 | 3/32 | 0.02 | 3 | 0.09 |
| m50_100_70_70 | $50 \times 100\ (70)$ | 2 | 2 | 0.01 | 2/0 | 0.01 | 3 | 1 |
| m50_100_90_90 | $50 \times 100\ (90)$ | 2 | 2 | 0.01 | 2/0 | 0.01 | 3 | 1 |
| m100_200_10_30 | $100 \times 200\ (10)$ | 12 | 281845 | 242.65 | 2915/161571 | 45.61 | 3 | 0.19 |
| m100_200_10_50 | $100 \times 200\ (10)$ | 12 | 281845 | 241.06 | 2915/161571 | 45.36 | 3 | 0.19 |
| m100_200_10_70 | $100 \times 200\ (20)$ | 8 | 19135 | 22.8 | 82/6538 | 2.36 | 3 | 0.10 |
| m100_200_30_30 | $100 \times 200\ (15)$ | 8 | 154475 | 117.5 | 31499/775717 | 220.05 | 3 | 1.90 |
| m100_200_30_50 | $100 \times 200\ (19)$ | 7 | 50613 | 78.03 | 4019/136979 | 59.58 | 3 | 0.76 |
| m100_200_30_70 | $100 \times 200\ (25)$ | 6 | 30577 | 61.55 | 707/15289 | 10.43 | 3 | 0.17 |
| m100_200_50_50 | $100 \times 200\ (25)$ | 6 | 32214 | 63.84 | 3753/78023 | 44.67 | 3 | 0.70 |
| m100_200_50_70 | $100 \times 200\ (29)$ | 5 | 4867 | 17.19 | 163/5581 | 4.94 | 3 | 0.29 |
| m100_200_70_70 | $100 \times 200\ (35)$ | 5 | 26588 | 63.73 | 245/22860 | 16.47 | 3 | 0.26 |
| m200_100_10_10 | $200 \times 100\ (10)$ | 16 | 13889095 | 10776.6 | 464553/16098542 | 3830.34 | 3 | 0.36 |
| m200_100_10_100 | $200 \times 100\ (54)$ | 6 | 317 | 1.79 | 9/250 | 0.21 | 3 | 0.12 |
| m200_100_10_30 | $200 \times 100\ (19)$ | 11 | 564302 | 584.54 | 9156/371430 | 115.52 | 3 | 0.20 |
| m200_100_10_50 | $200 \times 100\ (28)$ | 8 | 29803 | 46.64 | 528/17689 | 8.91 | 3 | 0.19 |
| m200_100_10_70 | $200 \times 100\ (40)$ | 7 | 1735 | 4.87 | 37/1046 | 1.01 | 3 | 0.21 |
| m200_100_30_100 | $200 \times 100\ (64)$ | 4 | 1725 | 11.09 | 5/185 | 0.38 | 3 | 0.03 |
| m200_100_30_30 | $200 \times 100\ (30)$ | 6 | 65468 | 115.44 | 883/31293 | 18 | 3 | 0.16 |
| m200_100_30_50 | $200 \times 100\ (39)$ | 6 | 123621 | 170.09 | 1177/51624 | 33.41 | 3 | 0.20 |
| m200_100_30_70 | $200 \times 100\ (51)$ | 4 | 2036 | 17.07 | 7/190 | 0.39 | 3 | 0.02 |
| m200_100_50_100 | $200 \times 100\ (74)$ | 3 | 145 | 7.08 | 3/52 | 0.33 | 3 | 0.05 |
| m200_100_50_50 | $200 \times 100\ (50)$ | 4 | 8076 | 35.4 | 9/1607 | 1.79 | 3 | 0.05 |
| m200_100_50_70 | $200 \times 100\ (60)$ | 4 | 5413 | 32.48 | 5/1302 | 2.31 | 3 | 0.07 |
| m200_100_70_100 | $200 \times 100\ (84)$ | 2 | 2 | 0.03 | 2/0 | 0.03 | 3 | 1 |
| m200_100_70_70 | $200 \times 100\ (70)$ | 3 | 169 | 10.89 | 3/90 | 0.46 | 3 | 0.04 |
| m200_200_100_100 | $200 \times 200\ (50)$ | 4 | 16313 | 259.45 | 5/2642 | 7.11 | 3 | 0.03 |

*Table 1.2* Results on random benchmarks (SCHERZO vs. AURA II).

3. The raising parameter $r$ is an input to AURA II. The higher the raising parameter, the fewer column branching nodes compared to cube branching nodes there will be. With a value that is high enough, there will be a single column node and the rest will be all row nodes.

The experiments show that AURA II is faster than SCHERZO, especially in the most time-consuming examples. For each of the difficult cases of Table 1.1, we have run AURA II with $r = 1, 2, 3$. There is always a value of $r$ which allows AURA II to solve the problem faster than SCHERZO and in general this value is either $2$ or $3$. However, for the problem *prom2* the higher is the value of $r$ the lower is the performance of AURA II: in fact, since this problem presents an highly diversified solution space, the raising procedure often terminates only after it has found a better solution (and, therefore, without having been able to prune rapidly the current branch). On the other hand, in the case of the problem *saucier*, whose solution space is poorly diversified, AURA II finds the solution in $24$ second with any possible value of $r$ while SCHERZO takes $11876$ seconds. These results are in concord with the philosophy of "negative thinking" as discussed in Section 1.: the less frequently the best current solution is improved during the search, the more the "negative" search is justified. Now, when we are running a very time-consuming problem, the overwhelming majority of the subproblems do not lead to a solution improvement and, therefore, "negative" search is more natural and, if applied, leads to spectacular savings in total time. This is confirmed by the experiments with the random generated matrices of Table 1.2, for which we have kept the raising parameter $r$ constantly equal to $3$. In the most time-consuming of these examples AURA II takes between $36\%$ and $75\%$ of the time of SCHERZO.

## 3.1 OTHER COMPARISONS

We do not have a systematic comparison with the results by BCU, a very efficient recently-developed ILP-based covering solver [Liao and Devadas, 1997]. However, the intuition is that an algorithm based on linear programming is better suited for problems with a solution space diversified in the costs, i.e., for problems which are "closer" to numerical ones. To test the conjecture we asked the authors of [Liao and Devadas, 1997] to run BCU on *saucier.t*, whose solution space is poorly diversified (a minimum solution has 6 columns, while most of the irredundant solutions cost in the range from 6 to 8). BCU ran out of memory after 20000 seconds of computations (the information was kindly provided by S.Liao), while AURA II completes the example in 24 seconds. It would be of interest to study if the virtues of an ILP-based solver and of *raiser* could be combined in a single algorithm.

# 4. CONCLUSIONS

In [Goldberg et al., 1997] the authors applied to UCP a novel technique to augment Branch-and-Bound (B&B) using a new way of exploring solutions, inspired by a paradigm called negative thinking. Traditional UCP solvers are based on the mincov algorithm [Rudell and Sangiovanni-Vincentelli, 1987] which keep searching the solution space in the hope of finding a better solution (positive thinking mode) The new paradigm led to the development of the *raiser* algorithm which can be coupled with *mincov* to better guide the exploration of the binary tree representing the solution space: in fact, the search for a better solution can be appropriately interleaved with the attempt to prove that no better solution can be found in the current branching node (negative thinking mode). This paper discusses the details of the *raiser* algorithm. Moreover, by reporting experimental results obtained with AURA II, a new state-of-the-art UCP solver which combines the best of both worlds, we settle some experimental questions left open in [Goldberg et al., 1997]. Future work includes the extension of AURA II to solve the binate covering problem.

## Acknowledgments

We gratefully thank Dr. Olivier Coudert (Monterey Design Systems) who kindly provided us a version of Scherzo and was always available for technical discussions.

## References

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

[Coudert, 1996] Coudert, O. (1996). On solving binate covering problems. In *The Proceedings of the Design Automation Conference*, pages 197–202.

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

[Goldberg et al., 1997] Goldberg, E., Carloni, L. P., Villa, T., Brayton, R. K., and Sangiovanni-Vincentelli, A. (1997). Negative thinking by incremental problem solving: application to unate covering. In *The Proceedings of the International Conference on Computer-Aided Design*, pages 91–99.

[Kam et al., 1997] Kam, T., Villa, T., Brayton, R. K., and Sangiovanni-Vincentelli, A. (1997). *Synthesis of FSMs: functional optimization.* Kluwer Academic Publishers.

[Liao and Devadas, 1997] Liao, S. and Devadas, S. (1997). Solving covering problems using LPR-based lower bounds. In *The Proceedings of the Design Automation Conference*.

[Rudell and Sangiovanni-Vincentelli, 1987] Rudell, R. and Sangiovanni-Vincentelli, A. (1987). Multiple-valued minimization for PLA optimization. *IEEE Transactions on Computer-Aided Design*, CAD-6:727–750.

[Villa et al., 1997a] Villa, T., Kam, T., Brayton, R. K., and Sangiovanni-Vincentelli, A. (1997a). Explicit and implicit algorithms for binate covering problems. *IEEE Transactions on Computer-Aided Design*, 16(7):677–691.

[Villa et al., 1997b] Villa, T., Kam, T., Brayton, R. K., and Sangiovanni-Vincentelli, A. (1997b). *Synthesis of FSMs: logic optimization*. Kluwer Academic Publishers.