Negative Thinking in Branch-and-Bound: the Case of Unate Covering

Evguenii I. Goldberg, Luca P. Carloni, *Member, IEEE*, Tiziano Villa, Robert K. Brayton, *Fellow, IEEE*, Alberto L. Sangiovanni-Vincentelli, *Fellow, IEEE*

Abstract— **We introduce a new technique for solving some discrete optimization problems exactly. The motivation is that when searching the space of solutions by a standard Branch-and-Bound (B&B) technique, often a good solution is reached quickly and then improved only a few times before the optimum is found: hence most of the solution space is explored to certify optimality, with no improvement in the cost function. This suggests that more powerful lower bounding would speed up the search dramatically. More radically, it would be desirable to modify the search strategy with the goal of proving that the given subproblem cannot yield a solution better than the current best one (negative thinking), instead of branching further in search for a better solution (positive thinking).**

For illustration we applied our approach to the Unate Covering Problem. The algorithm starts in the positive-thinking mode by a standard B&B procedure that generates recursively smaller subproblems. If the current subproblem is "deep" enough, the algorithm switches to the negative thinking mode where it tries to prove that solving the subproblem does not improve the solution. The latter is achieved by a new search procedure invoked when the difference between the upper and lower bound is "small". Such a procedure is complete: either it yields a lower bound that matches the current upper bound, or it yields a new solution better than the current one. We implemented our new search procedure on top of ESPRESSO and SCHERZO, two state-of-art covering solvers used for CAD applications, showing that in both cases we obtain new search engines (respectively, AURA and AURA II) much more efficient than the original ones.

Keywords— **Logic Optimization, Combinatorial Optimization, Branch&Bound Techniques, Covering Problems.**

I. Introduction

Branch-and-bound (B&B) is a common search technique to solve exactly problems in combinatorial optimization. B&B improves over exhaustive enumeration, because it avoids the exploration of those regions of the solution space, where it can be certified (by means of lower bounds) that no solution improvement can be found.

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. The exploration of the solution space can be represented by a search tree, whose nodes represent sets of solutions, which can be further partitioned in mutually exclusive sets. Each subset in the partition is represented by a child of the original node. An algorithm that computes a lower bound on the cost of any solution in a given subset prevents 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 no children need 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 a node x, then we say that y *dominates* x, and y can kill x. Figure 1 shows the standard algorithm [1]. An *activeset* holds the live nodes at any point. A variable U is an upper bound on the optimum cost (cost of the best complete solution obtained so far).

```
branch_and_bound () {
  activeset = original problem
  U = ∞
  currentbest = anything
  while (activeset is not empty) {
    choose a branching node k ∈ activeset
    remove node k from activeset
    generate the children of node k: child i = 1, . . . , n_k
      and the corresponding lower bounds z_i
    for i = 1 to n_k {
      if (z_i ≥ U) kill child i
      else if (child i is a complete solution) {
        U = z_i
        currentbest = child i
        else add child i to activeset
      }
    }
  }
}
```

Fig. 1. Structure of branch-and-bound.

An important feature of many practical discrete optimization problems is that the current best solution can be improved only very few times. In turn this is related to how much the solution space is "diversified", i.e., different solutions have different costs. For example, if the first solution found for an instance of the graph coloring problem has 20 colors and an optimum solution takes 15 colors, we can have no more than 5 improvements to the current best solution. On the other hand, the number of subproblems generated at a "deep enough" level of the search tree is very large. For instance even at level 10 of a B&B search tree, as many as 2^{10} subproblems may be generated. This means that only for a tiny fraction of 2^{10} subproblems can a better solution be found, whereas in the overwhelming majority of subproblems the solution is not improved. Therefore, for

Evguenii I. Goldberg is with Cadence Berkeley Laboratories, Berkeley, CA 94704-1103. E-mail: egold@cadence.com.

Luca P. Carloni, Robert K. Brayton, and Alberto L. Sangiovanni-Vincentelli are with the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, Berkeley, CA 94720-1772. E-mail: {lcarloni,brayton,alberto}@eecs.berkeley.edu.

Tiziano Villa is with PARADES, Via di S. Pantaleo 66, 00186 Roma, Italy. E-mail: villa@parades.rm.cnr.it.

a "deep" subproblem it is reasonable to be negative, trying to prove in the first place that the current best solution cannot be improved.

A branch-and-bound procedure may be seen as consisting of positive and negative thinking search modes. The positive thinking mode looks for a better solution by branching, while the negative thinking mode tries to prune the current path by lower bounding. Intuitively, when solving a subproblem A the relation between positive and negative modes should be "proportional" to the ratio between the probability of finding a better solution and that of proving that the current best solution cannot be improved. However, in traditional B&B the boundary between positive and negative modes is rigid and depends solely on the power of the lower bounding procedure. So if the latter fails to prune the path leading to a node associated with a subproblem A, then B&B tries to solve A in the positive thinking mode, even though the chances of improving the solution by means of A are very small.

The key point of our approach is to shift the boundary between the two modes of B&B in order to exercise more negative thinking. Namely, when the lower bound procedure fails to prune the current branch, while being "close" to do it, we apply negative thinking by invoking a special incremental problem solving procedure on the subproblem A.

The incremental problem-solving procedure is based on the following observation. Typically a lower bound on optimal solutions to a problem A is computed by extracting a subproblem A' for which: 1) finding an exact solution is very easy, 2) the cost of an exact solution to A' is not more than the cost of an exact solution to A. For instance, when solving the graph coloring problem, a maximum size complete subgraph is extracted, since optimal coloring of a complete graph is trivial. When solving UCP, a maximum subset of independent rows is extracted, because finding an optimal covering of independent rows is trivial. In both cases the ease of finding an optimal solution is due to the solution space "regularity": e.g., any coloring of a complete graph can be obtained from another by permutation; the set of all irredundant coverings of a set of independent rows can be represented as a single Cartesian product.

Let A' be a subproblem of A with a regular solution space. If the cost of the optimal solutions of A' is not large enough to prune the current path of the search tree, we can augment A' to make it "closer" to A. Let A'' be such an augmented problem. Then, instead of solving A'' anew, we can find the optimal solutions of A'' by refining the set of optimum solutions of A'. This should not be hard to do because the set of solutions of A' by hypothesis can be represented in a compact form. The set of optimum solutions of A'' is in general less regular than for A', but their cost has increased. In the negative thinking mode, A' is augmented to increase as much as possible the cost of the optimum solutions of the augmented problem A''; to that purpose, we look first for the most difficult "obstacles" in the sequence from A' to A, trying to prove that no solution of A' can overcome the obstacles and be extended to a solution of A that is better than the current best one. This is achieved by clustering similar solutions: i.e., we group in a cluster those solutions of A' which have the same reason for not producing solutions of A costing less than *ubound*.

In this paper we introduce a Unate Covering Problem (UCP) solver working in two modes. In the positive thinking mode it uses a standard B&B procedure like *mincov* in ESPRESSO [2] or the one available in SCHERZO [3]. When the lower bound on optimal coverings for the current submatrix A is close to bounding the search, the solver switches to the negative thinking mode, by invoking an incremental problem solving procedure termed *raiser*. The procedure *raiser* starts with a *Maximal Set of Independent Rows* ($MSIR$) of A of size L (that failed to bound the search) and constructs the set of irredundant coverings of the $MSIR$. Then, *raiser* adds to the $MSIR$ new rows from A, which are the most difficult to cover by solutions of the $MSIR$. The solutions of the augmented set of rows are computed, possibly increasing the minimum solution cost; if all solutions with cost less than L are eliminated, then *raiser* proved that the current subtree can be pruned away.

The paper is organized as follows. Section II shows how an incremental solver is incorporated into the standard B&B procedure for UCP. Section III describes how to represent and recompute efficiently the solutions of UCP in the negative thinking search mode. The raising procedure is explained in detail in Section IV. Experimental results are discussed in Section V. Conclusions are given in Section VI.

II. Incorporating an Incremental Solver into B&B for UCP

A. *Revisiting the Procedure* mincov

In this paper we apply the proposed search technique to UCP, a problem of wide interest in logic synthesis and operations research [4]. UCP can be stated as follows.

Definition II.1: 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 $x^T c = \sum_{j=1}^{n} x_j c_j$, where $x \in \{0,1\}^n$, subject to:

$$A\ x \geq (1, 1,\ \cdots\ , 1)^T. \tag{1}$$

The constraint $A\ x \geq (1, 1,\ \cdots\ , 1)^T$, ensures that the nonzero elements of x determine a column set $S = \{j\ |x_j = 1\}$, which *covers* all rows of A, that is, $\forall i\ ,\ \ \exists j \in S$ such that $A_{i,j} = 1$. Thus the minimum unate covering problem is to find a column set of minimum cost, which satisfies the constraint of (1). For simplicity we will assume that $c_j = 1\ ,\ \forall j$. We will also say that two rows are independent or non-intersecting when there is no column that covers both. We will denote an instance of UCP with matrix A by the notation $\boldsymbol{UCP(A)}$. Notice that UCP can be seen as a matrix formulation of the *MINIMUM COVER* problem [5].

An exact solution is obtained by a B&B recursive algorithm, *mincov*, which has been implemented in successful computer programs such as ESPRESSO and STAMINA.

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 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 *mincov*, an internal node is a reduced input. A leaf is reached when a complete solution is found or the search is bounded. From the root to any internal node there is a unique path, which is the current path for that node. The inputs of the *mincov* algorithm are:

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

The best column cover for input A extended from the partial solution *path* is returned as the best current solution, if it costs less than *ubound*. An empty solution is returned if a solution cannot be found which beats *ubound*. When *mincov* is called on A with an empty partial solution *path* and initial *lbound* and *ubound*, it returns a best global solution.

The flow of a UCP solver based on branch-and-bound enhanced by an incremental solver is shown in Figure 2. The parts of text not in bold font correspond to the original *mincov* algorithm, stripped away of some additional features like matrix partitioning and Gimpel's reduction. Given a matrix A, most existing UCP solvers employ column branching to decompose the problem and use an $MSIR$ to compute a lower bound of $UCP(A)$ (since no column covers two rows from an $MSIR$) [1].

The parts of text in bold font refer to the added incremental solver, whose main search engine is the procedure *raiser*. The *raiser* procedure performs "negative thinking" and is invoked when the following situation occurs: $MSIR$ is a lower bound not sufficient to prune the subtree rooted at the current node, whereas increasing the lower bound by a small integer n would allow such pruning. In this case, *raiser* starts from the subproblem $UCP(MSIR)$, whose solution space is very regular, and tries gradually to extend it to the entire problem $UCP(A)$: as a result, *raiser* either returns a minimum cost solution of $UCP(A)$ (if the lower bound cannot be raised by n) or returns the empty solution.

The value n corresponds to the *difference* between the current upper bound and the current lower bound. If *difference* ≤ 0, the current branch can be pruned because it can not lead to a solution improvement. If *difference*

[1]Notice that ILP-based covering solvers, such as BCU [6], do not need to compute the MSIR.

```
AuraMincov(A, path, w, lbound, ubound) {
  /* Apply row/column dominance, and select essentials */
  if (not reduce(A, path, w, ubound)) return empty_solution
  /* Find lower bound from here to final solution */
  MSIR = maximal_independent_set(A, w)
  /* Make sure the lower bound is monotonically increasing */
  lbound_new = max(cost(path) + cost(MSIR), lbound)
  difference = ubound − lbound_new
  /* Bounding based on no better solution possible */
  if (difference ≤ 0)
    best = empty_solution
  else if (difference ≤ maxRaiser){
    /* Apply raiser with n = difference */
    SolCube = cover_MSIR(MSIR)
    lowerBound = |SolCube|
    answer = raiser(SolCube, difference, A,
    lowerBound, bestSolution, ubound)
    if (answer = 1)
      best = empty_solution
    else
      best = path ∪ bestSolution   /* (answer = 0) */
  }
  else if (A is empty) { /* New best solution at current level */
    best = path
  } else { /* Branch on cyclic core and recur */
    branch = select_column(A, w, MSIR)
    path1 = solution_dup(path) ∪ branch
    /* A_branch: reduced table assuming branch in solution */
    best1 = AuraMincov(A_branch, path1, w, lbound_new, ubound)
    /* Update the upper bound if better solution is found*/
    if (best1 ≠ empty_solution) /* i.e., (ubound > cost(best1)) */
      ubound = cost(best1)
    /* Do not branch if lower bound matched */
    if (best1 ≠ empty_solution) and (cost(best1) = lbound_new)
      return best1
    /* A_(not branch): reduced table assuming branch not in solution */
    best2 = AuraMincov(A_(not branch), path, w, lbound_new, ubound)
    best = best_solution(best1, best2)
  }
  return best
}
```

Fig. 2. AuraMincov: *mincov* enhanced by incremental raising.

$= n > 0$, the search can be continued within *raiser* instead of marching on with column branching. However, practically, *raiser* is invoked only if $0 < n \leq maxRaiser$, where $maxRaiser$ is a parameter fixed a-priori. The value of $maxRaiser$ is usually a small number in the range from 1 to 3 for two reasons:

1. if n is small then the node is deep enough to warrant the application of negative thinking,
2. if n is small then the fact that $UCP(MSIR)$ has a regular solution space can be used.

In the following section the basic idea behind the *raiser* is presented (the details are given in Section IV). A discussion on the impact of different values of $maxRaiser$ is given in Section V.

B. Introducing raiser to Improve the Lower Bound

To introduce *raiser* we need the following notation:

- $min(UCP(A))$ is the size of a minimum solution of $UCP(A)$.
- Let A' be a submatrix of A, where the set of columns and rows of A' are defined respectively as $Col(A') = Col(A)$ and $Row(A') \subseteq Row(A)$. A' is a *lower bound submatrix* if its minimum solution is a lower bound for $UCP(A)$.
- An $MSIR$ of A is usually chosen as a lower bound sub-

matrix A', denoted also as $A' = MSIR(A)$. If A' is a $MSIR$ then $min(UCP(A')) = |Row(A')|$.

- $A' + A_r$ denotes the submatrix obtained by adding a row $A_r \in Row(A) \setminus Row(A')$ to A'.
- Let S be a solution of $UCP(A)$. A column $j \in S$ is *redundant* if $S \setminus \{j\}$ is also a solution. A solution of $UCP(A)$ that does not contain redundant columns is said *irredundant.*
- $Sol(A', n)$ denotes the set of all irredundant solutions of $UCP(A')$ consisting of n or fewer columns. $Sol(A', m)$, where $m = min(UCP(A'))$, is the set of all minimum solutions of $UCP(A')$.
- The solutions of $UCP(A)$ are represented by sets with the structure of multi-valued cubes [2]. We define a *cube* to be the set $C = D_1 \times \cdots \times D_d$ where $D_i \cap D_j = \emptyset$, $i \neq j$ and $D_i \subset Col(A)$, $1 \leq i, j \leq d$. The subsets D_i are the *domains* of cube C. C denotes a set of sets consisting of d columns. In contrast to common cubes used for the representation of multi-valued functions, cubes here may have different numbers of domains. For example, if $|Col(A)| = 10$, then sets $C_1 = \{1, 5\} \times \{2, 6, 7\} \times \{3, 4\}$ and $C_2 = \{1\} \times \{2, 4\} \times \{3, 7\} \times \{5, 6, 10\}$ are both cubes.
- $O(A_i)$ is the set of all columns covering row A_i.
- The cost of every set of columns in C is the number of domains of C, denoted by $cost(C)$, since the cost of a set of columns is its cardinality.
- A set $S' \subseteq Col(A)$ is a *partial solution* of $UCP(A)$ if it is not a solution of $UCP(A)$.
- A set P of partial solutions is *complete* if for any solution S of $UCP(A)$ there is a partial solution S' in P with $S' \subseteq S$.

We now describe the idea underlying the method for an incremental improvement of the lower bound. Suppose that for a lower bound submatrix A' of A we know a set of solutions $Sol(A', n)$. The lower bound given by A' is equal to $m = min(UCP(A'))$. Now add a row A_p of A to A'. Obviously $Sol(A' + A_p, m) \subseteq Sol(A', m)$, since in general some solutions from $Sol(A', m)$ do not cover A_p and so are not contained in $Sol(A' + A_p, m)$. So after adding a set of rows $A_{i_1}, .., A_{i_k}$ of A to A', if $Sol(A' + A_{i_1} + .. + A_{i_k}, m) = \emptyset$ then the lower bound for $UCP(A)$ has improved by 1. If $Sol(A' + A_{i_1} + .. + A_{i_k}, n) = \emptyset, n \geq m$, then the lower bound has improved by $n - m + 1$.

We start from a submatrix A' which is an $MSIR$ (since the solutions of an $MSIR$ can be represented compactly) and then we add rows to the $MSIR$ with the goal to improve the initial lower bound given by $|MSIR|$. The proposal relies on the fact that, knowing $Sol(A', n)$, it is not difficult to recalculate $Sol(A' + A_p, n)$. In Section III we explain how to represent and update efficiently the set of solutions of a matrix.

The previous discussion motivates the *raiser* procedure. At any given node N in the search tree, the $MSIR$ for the corresponding matrix A_N is computed. If $|MSIR| + |path(A_N)| + n \geq |best|$, where *best* is the best current solution, then the *raiser* procedure is applied to $UCP(A_N)$, otherwise branching on columns continues. The outcome of *raiser* may be one of the following: (1) the lower bound $|MSIR|$ can be increased by $|best| - |MSIR| - |path(A_N)|$ and the recursion in the node stops, or (2) a minimum solution $S(A_N)$ of $UCP(A_N)$ is found such that $S(A_N) \cup path(A_N)$ is the new best current solution of $UCP(A)$.

Notice that improving the lower bound even by a small amount may lead to considerable runtime reductions. For example, in [7] the limit lower bound is defined, which allows some branches of the search tree to be pruned. The effect is to reduce the runtimes for some examples by a factor of 10 or more. It can be shown that the limit lower bound technique prunes no more branches of the search tree than the application of *raiser* when n is equal to 1.

The challenge is to design an efficient procedure to implement *raiser*. In fact, a "naive" implementation where one stores the set of solutions $Sol(A', |MSIR(A)| + n)$ may require too much memory. Indeed, if *raiser* fails to raise the lower bound then A itself will be taken as a lower bound submatrix and we will have to store all irredundant solutions of $UCP(A)$ with $|MSIR| + n$ or fewer columns. Our solution to the potential memory problem relies on using a data structure called "cubes" and a new scheme of branching on rows. Before embarking on a detailed description of the *raiser* procedure (found in Sections III and IV) we illustrate the idea with an example.

C. Example: Lower Bound by raiser

Consider the following matrix A:

	1	2	3	4	5	6	7	8	9	10	11
1	1	0	1	0	1	0	0	0	0	1	1
2	0	0	1	0	0	0	0	0	1	0	0
3	1	0	1	0	0	0	0	1	0	0	0
4	0	1	0	0	0	0	1	0	0	0	1
5	0	0	0	0	0	0	0	1	1	1	0
6	1	1	0	0	0	0	0	0	0	0	0
7	0	0	0	1	1	1	0	0	0	0	0

Since $MSIR(A) = \{A_5, A_6, A_7\}$, the lower bound is 3. Suppose this value is not sufficient to prune the current path of the search tree, but a lower bound of 4 would suffice. We show how *raiser* can increase the lower bound by 1 by means of incremental problem solving. The set of all irredundant solutions of the subproblem $UCP(MSIR(A))$ is given by the cube $C = \{8, 9, 10\} \times \{1, 2\} \times \{4, 5, 6\}$, i.e., any set of 3 columns c_1, c_2, c_3 such that $c_1 \in A_5$, $c_2 \in A_6$ and $c_3 \in A_7$ is an irredundant solution of $UCP(MSIR(A))$.

Consider $UCP(MSIR(A) + A_4)$, whose irredundant solutions can be obtained from $UCP(MSIR(A))$ given by cube C. We partition C into two cubes: $C_1 = \{8, 9, 10\} \times \{2\} \times \{4, 5, 6\}$ and $C_2 = \{8, 9, 10\} \times \{1\} \times \{4, 5, 6\}$. All sets of columns specified by C_1 are solutions of $UCP(MSIR(A) + A_4)$, since they cover row A_4, but none of the sets specified by C_2 is a solution because they do not cover A_4. For sets of columns from C_2 to be extended to solutions of $UCP(MSIR(A) + A_4)$, one must add to C_2 one more domain, $\{7, 11\}$, since $C_2' = C_2 \times \{7, 11\} = \{8, 9, 10\} \times \{1\} \times \{4, 5, 6\} \times \{7, 11\}$ are solutions of $UCP(MSIR(A) + A_4)$.

Since we want to raise the lower bound from 3 to 4, we can discard the solutions specified by C_2' (they cost 4) and focus only on the solutions specified by C_1 (they cost 3). The goal is to increase by 1 their cardinality. Now consider the solutions of $UCP(MSIR(A)+A_4+A_3)$ that can be obtained from C_1. Cube C_1 can be partitioned into two cubes: $C_{11}=\{8\}\times\{2\}\times\{4,5,6\}$ and $C_{12}=\{9,10\}\times\{2\}\times\{4,5,6\}$. Cube C_{11} specifies solutions of $UCP(MSIR(A)+A_4)$ that are also solutions of $UCP(MSIR(A)+A_4+A_3)$, whereas cube C_{12} does not contain any solution of $UCP(MSIR(A)+A_4+A_3)$. For sets of columns from C_{12} to be extended to solutions also of $UCP(MSIR(A)+A_4+A_3)$, one must add domain $\{1,3\}$ to C_{12}. The solutions specified by $C_{12}\times\{1,3\}$ consist of 4 columns and so are discarded. Finally none of the sets of columns specified by C_{11} covers row A_2, so $Sol(UCP(MSIR(A)+A_4+A_3+A_2),3)=\emptyset$. Hence a stronger lower bound of 4 is obtained on the solutions of $UCP(A)$.

III. Representation and Recomputation of the Solutions

As a stepping stone to the algorithm for raising the lower bound the method for representing and updating the set of solutions of a matrix is presented first.

Let A' be a submatrix of A and A_p a row from $Row(A)\setminus Row(A')$. Let S be a solution of $UCP(A')$.

Definition III.1: $Rec(A'+A_p,S)$ denotes the 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.

Rec is a complete operator, i.e., every irredundant solution S^* of $UCP(A'+A_p)$ can be obtained by the recomputation of a corresponding irredundant solution S of $UCP(A')$:

Theorem III.1: For any irredundant solution S^* of $UCP(A'+A_p)$ there is an irredundant solution S of $UCP(A')$ such that S^* is an element of $Rec(A'+A_p,S)$.

A proof can be found in the Appendix A-A. From Theorem III.1 follows directly:

Corollary III.1: Let Sol be a set containing all irredundant solutions S of $UCP(A')$. Let

$$Sol^*=\bigcup_{S\in Sol} Rec(A'+A_p,S),$$

then Sol^* contains all solutions S^* of $UCP(A'+A_p)$ [2].

[2] There are examples showing that $Rec(A'+A_p,S)$ may contain also redundant solutions.

We give an operational definition of $Rec(A'+A_p,S)$ when solutions are represented by multi-valued cubes defined in II-B. Applying the operator *Rec* to a cube of solutions leads to a collection of cubes of solutions, thereby providing a natural clustering of the recomputed solutions. This supports the design of a raising algorithm based on branching by subsets of solutions, each subset being one of the recomputed cubes of solutions. Let A' be a $MSIR$ 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 of A'.

Let A' be a submatrix of A and A_p be a row from $Row(A)\setminus Row(A')$. Let $C=D_1\times\cdots\times D_d$ be a cube of solutions of $UCP(A')$. Using

$$Rec(A,C)=\bigcup_{c\in C} Rec(A,c),$$

$Rec(A'+A_p,C)$ can be rewritten as

$$Rec(A'+A_p,C)\;=\;part1(C)\;\cup\;part2(C)\times O(A_p),\quad(2)$$

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.

Now we want to rewrite $part1(C)$ and $part2(C)$ as unions of disjoint cubes. There are three cases:
1. $D_i\subseteq O(A_p)$ for some i, $1\le i\le d$. Then any solution from C covers the row A_p and so $Rec(A'+A_p,C)=C$.
2. $O(A_p)\cap D_i=\emptyset$ for any $i, 1\le i\le d$. Then no solution from C covers A_p and so $Rec(A'+A_p,C)=C\times O(A_p)=D_1\times\cdots\times D_d\times O(A_p)$.
3. Cases 1. and 2. are not true, i.e., no D_i is a subset of $O(A_p)$, $O(A_p)$ intersects at least one domain, and we assume w.l.o.g. that A_p intersects the first r domains $D_1,\cdots,D_r$. Then cube C can be partitioned into the following $r+1$ pairwise non-intersecting cubes:

$$\begin{aligned}
C_1 &= D_1\cap O(A_p)\times D_2\times\cdots\times D_d\\
C_2 &= D_1\setminus O(A_p)\times D_2\cap O(A_p)\times D_3\times\cdots\times D_d\\
C_3 &= D_1\setminus O(A_p)\times D_2\setminus O(A_p)\times D_3\cap O(A_p)\times\\
&\quad\times D_4\times\cdots\times D_d \qquad (3)\\
&\vdots\\
C_r &= D_1\setminus O(A_p)\times\cdots\times D_{r-1}\setminus O(A_p)\times\\
&\quad\times D_r\cap O(A_p)\times D_{r+1}\times\cdots\times D_d\\
C_{r+1} &= D_1\setminus O(A_p)\times\cdots\times D_{r-1}\setminus O(A_p)\times\\
&\quad\times D_r\setminus O(A_p)\times D_{r+1}\times\cdots\times D_d.
\end{aligned}$$

It is not hard to check that

$$C=C_1\cup\cdots\cup C_{r+1}$$

and that for any pair $C_i,C_j,i\ne j,C_i\cap C_j=\emptyset$. Moreover, the first r cubes give the solutions of $UCP(A')$ from C which cover A_p and the cube C_{r+1} gives the solutions of $UCP(A')$ from C which do not cover A_p. Therefore

$$part1(C)=C_1\cup\cdots\cup C_r,\;\;part2(C)=C_{r+1}.\qquad(4)$$

In summary, equations. 2–4 define operationally the *Rec* operator over the cubes of solutions, consistently with definition. III.1. Although generating non-intersecting cubes of solutions $C_i, i = 1, \dots, r+1$ is not required by definition III.1 of *Rec*, it avoids the occurrence of the same partial solution in more than one branch.

The following revised operational definition of *Rec* avoids the generation of some redundant solutions, namely, of any solution S' of $UCP(A' + A_p)$ from $part2(C) \times O(A_p)$ that strictly contains a solution S'' of $UCP(A' + A_p)$ from $part1(C)$. The next theorem tightens the operational definition of *Rec* and states that no irredundant solutions are lost.

Theorem III.2: If the computation of the *Rec* operator is modified as follows:

$$\begin{aligned} Rec(A' + A_p, C) &= part1(C) \cup \{\, part2(C) \times \\ &\times \; [O(A_p) \setminus (D_1 \cup \cdots \cup D_d)] \,\} \end{aligned} \quad (5)$$

no irredundant solution of $A' + A_p$ is discarded.
A proof can be found in the Appendix A-A. In addition, another method for avoiding the repeated generation of some solutions is discussed in the Appendix A-B. In practice, avoiding generation of repeated solutions gives a 30-40% speed-up on the overall computation. In the following section we discuss the raising procedure in detail, showing how these techniques are embedded in the algorithm.

IV. The Raising Procedure

Fig. 2 shows how a traditional UCP solver is enhanced by the technique to raise incrementally the lower bound. After the computation of the lower bound, if the gap *difference* between the upper and lower bound is a small positive number, i.e., less than a global parameter *maxRaiser*, the *raiser* procedure is invoked with a parameter n set to the value of *difference*. In this case, we say that a *n-raiser* has been invoked. Intuitively if the gap is small, we conjecture that a search in this subtree will not improve the best solution. *n-raiser* either confirms the conjecture and proves that no better solution can be found, or disproves the conjecture and improves the best solution by at least 1.

```
raiser(SolCube, n, A, lbound, bestSolution, ubound) {
  /* return 1 if solutions in SolCube raise lower bound of A by n */
  stillToRaise = lbound + n − number_domains(SolCube)
  if (stillToRaise ≤ 0) return 1
  /* If A = ∅ then (path ∪ solutions in SolCube) beats ubound */
  if (A = ∅)
    return found_solution(SolCube, n, bestSolution, ubound)
  /* find rows of A not covered by any solution from SolCube */
  BSONIR = find_best_set_of_non_intersecting_rows(A, SolCube)
  foreach row r_i ∈ BSONIR {
    /* add a new domain for the columns covering r_i ∈ A */
    SolCube = add_domain(SolCube, A, r_i)
    stillToRaise = stillToRaise − 1
    if (stillToRaise ≤ 0) return 1
  }
  /* Remove the covered rows and check if A is empty */
  A = A \ BSONIR
  if (A = ∅)
    return found_solution(SolCube, n, bestSolution, ubound)
  if (stillToRaise = 1) {
    /* Cover with SolCube and remove the 1-intersecting rows */
    /* If 2 rows intersect different cols in a domain, prune branch */
    if (add_set_of_1intersecting_rows(A, SolCube) = 1) return 1
    if (A = ∅)
      return found_solution(SolCube, n, bestSolution, ubound)
  }
  /* select (and remove from A) next row to cover with SolCube */
  r_i = select_best_uncovered_row(A, SolCube)
  A = A \ {r_i}
  /* Split: part1 = {SolCube_1, ···, SolCube_k} and ...
  /*... part2 = {SolCube_k+1} */
  split_cubes(SolCube, A, r_i, part1, part2)
  /* add to SolCube_2 ∈ part2 new domain of columns covering r_i */
  SolCube_k+1 = add_domain(SolCube_k+1, A, r_i)
  /* branching on cubes of part1 and part2 */
  returnValue = 1
  while (part1 ∪ part2 ≠ ∅) {
    /* select first cubes from part1, then cube from part2 */
    SolCube_j = get_next_cube(part1 ∪ part2)
    /* if a better global solution is found set returnValue = 0 */
    if (raiser(SolCube_j, n, A, lbound, bestSolution, ubound) = 0)
      returnValue = 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
}
```

Fig. 3. Algorithm to raise the lower bound.

A. Overview of the Raising Algorithm

The *n-raiser* procedure is based on row branching. Given a covering matrix A, for which $A' = MSIR(A)$, the irredundant solutions of $UCP(A')$ are represented by the cube $C = O(A_{i_1}) \times \cdots \times O(A_{i_d})$, in which $A_{i_1}, \cdots, A_{i_d}$ are the rows in the $MSIR$. A "good" row A_p is chosen from A. According to Equations (2–5), $Rec(MSIR(A) + A_p, C)$ is represented by $r+1$ cubes where r is the number of rows of the $MSIR(A)$ intersecting A_p. This is applied recursively for each of the $r+1$ cubes.

The 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 $A \setminus MSIR(A)$. 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 and added to the first. The number of branches exiting a node is the number of cubes generated by the *Rec* operator; each child node gets one of the cubes obtained after splitting. Thus 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 *n-raiser*, the branches corresponding to cubes of more than $|MSIR(A)| + n$ domains are pruned.
- If - at a node - 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 $C \times$

$[O(A_p) \setminus (D_1 \cup \cdots \cup D_d)]$.

• The following reduction rule can be applied to the second matrix of the pair: if a row is covered by every solution of the cube C corresponding to the node, then the row can be removed from the matrix.

The recursion terminates if either:

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. 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 $\geq |MSIR|+n$. In this case the lower bound has been raised to $|MSIR| + n$, since no solution S of $UCP(A)$ exists such that $|S| < |MSIR| + n$.

Theorem IV.1: The *n-raiser* procedure is correct.

Proof: The *n-raiser* procedure starts with the set of solutions of $UCP(MSIR)$, which is a complete set of partial solutions of $UCP(A)$. Since by Theorem III.1 the *Rec* operator preserves completeness of a set of partial solutions, the set of cubes of any cut of the *n-raiser* search tree is a complete set of partial solutions, where a cut is a set of nodes that intersects any path from the root to a leaf (nodes in a cut can be either leaf nodes of the search tree or nodes that can still be split). The invariant that any cut set of nodes is a complete set of partial solutions guarantees that all solutions of $UCP(A)$ eventually are explored, explicitly or implicitly. The procedure *n-raiser*, applied to A, attempts to find a complete set of partial solutions each containing at least $|MSIR(A)| + n$ columns. If such a set is found, then no solution of $UCP(A)$ has less than $|MSIR(A)| + n$ columns, and so the procedure *n-raiser* succeeds in increasing the lower bound by n.

If there is no complete set of partial solutions consisting of at least $|MSIR(A)| + n$ columns, then by construction the *n-raiser* procedure creates a leaf node with a cube containing solutions of $|MSIR(A)|+n'$ columns, where $n' < n$. If so, the procedure *n-raiser* is tightened to be the procedure *n'-raiser*, $n' < n$, and the search is continued. If the lower bounding goal of *n'-raiser* is achieved, it returns a solution of $|MSIR(A)| + n'$ columns, which is the minimum computed so far. If instead *n'-raiser* fails to raise the lower bound by n', then by construction it exhibits a solution of $UCP(A)$ consisting of $|MSIR(A)| + n''$ columns, where $n'' < n'$. So *n'-raiser* is tightened again to be the procedure *n''-raiser* and the search is continued, until eventually all solutions of $UCP(A)$ are enumerated. Notice that by construction, at any given node of a cut set, a lower bound lb, an upper bound ub and a k-raiser procedure with $k = ub-lb$ are defined. ■

B. Complexity of the Raising Algorithm

The complexity of the raising algorithm is dictated by the size of the cube branching tree, which, in the worst case, is exponential in the cardinality of the set of rows $Rows(A) \setminus MSIR(A)$, i.e., the set of rows that are different from $MSIR(A)$ and not covered yet when *n-raiser* is invoked. However, the following considerations can also be made:

1. If the number of uncovered rows is extremely large and no better solution exists in the current branch of the column branching tree (i.e., the procedure *raiser* succeeds in raising the lower bound), then the size of the cube branching tree is usually small. This is due to the fact that there is a large choice of rows which can be selected to improve the lower bound and, therefore, usually it is easy to find quickly witnesses that no better solution can be found in the current branch.
2. If the number of uncovered rows is small, we have also an easy case because the size of the cube branching tree is exponential in the small number of uncovered rows. This situation may happen regardless of whether *n-raiser* improves the solution or not, but, in practice, the former case is more common.
3. The worst case is when *n-raiser* ends up disproving solutions which are "close" in quality to the current best. In this case, the number of rows in the set $Rows(A) \setminus MSIR(A)$ is neither big nor small and the size of the cube branching tree can grow fairly large.

C. Example: Upper Bound by raiser

Consider *1-raiser* applied to the following matrix A:

	1	2	3	4	5	6	7
1	0	0	0	1	1	0	1
2	1	0	1	0	0	0	0
3	0	1	1	1	0	1	0
4	0	0	0	0	0	1	1
5	1	1	0	0	0	0	0
6	0	0	1	0	1	0	0

Suppose that the set of rows $A' = \{A_4, A_5, A_6\}$ is chosen as $MSIR(A)$. The set of irredundant solutions of $UCP(A')$ is

$$C = \{1,2\} \times \{3,5\} \times \{6,7\}$$

for which

$$D = D_1 \cup D_2 \cup D_3 = \{1,2\} \cup \{3,5\} \cup \{6,7\} = \{1,2,3,5,6,7\}.$$

C gives a lower bound of 3 (C has 3 domains). The root node of the search tree is specified by C and the pair A', A'' where $Row(A'') = Row(A) \setminus Row(A')$. The aim of applying *1-raiser* to A is to improve the lower bound from 3 to 4.

Choose row A_3 from A'' to be added to A'. Since $O(A_3) = \{2,3,4,6\}$ then row A_3 intersects all three rows of A'. Therefore by Equations (2–4) the set of all irredundant solutions of no more than 4 columns of $A' + A_3$ is obtained as follows:

$$\begin{aligned}
C_1 &= \{2\} \times \{3,5\} \times \{6,7\} \\
C_2 &= \{1\} \times \{3\} \times \{6,7\} \\
C_3 &= \{1\} \times \{5\} \times \{6\} \\
C_4 &= \{1\} \times \{5\} \times \{7\} \\
part1\,(C) &= C_1 \cup C_2 \cup C_3 \\
part2\,(C) &= C_4 \\
Rec(A' + A_3, C) &= part1\,(C) \cup C_4 \times (O(A_3) \setminus D)
\end{aligned}$$

so that [3]

$$\begin{array}{rcl} Rec(A' + A_3, C) & = & (\ \{2\} \times \{3,5\} \times \{6,7\} \\ & \cup & \{1\} \times \{3\} \times \{6,7\} \\ & \cup & \{1\} \times \{5\} \times \{6\}\ ) \\ & \cup & \{1\} \times \{5\} \times \{7\} \times \{4\} \end{array}$$

Cube C_1 describes the set of solutions from C covering $A' + A_3$ in which A_3 is necessarily covered by a column of the first domain of C (and maybe by columns of other domains) and so $C_1 = \{1,2\} \cap O(A_3) \times \{3,5\} \times \{6,7\}$. Cube C_2 describes the set of solutions not contained in C_1 in which row A_3 is necessarily covered by a column of the second domain and so $C_2 = \{1,2\} \setminus O(A_3) \times \{3,5\} \cap O(A_3) \times \{6,7\} = \{1\} \times \{3\} \times \{6,7\}$. Cube C_3 describes the set of solutions from C not contained in C_1 or C_2 in which A_3 is necessarily covered by a column of the third domain. Finally, cube C_4 describes the set of solutions of $UCP(A')$ from C which do not cover row A_3 and so are not solutions of $UCP(A' + A_3)$.

Hence the root node has four children nodes, each specified by one of the four cubes $C_1, C_2, C_3, C_4 \times (O(A_3) \setminus D)$ and by the pair of matrices $A' + A_3, A'' - A_3$. Consider the branch corresponding to $C_1 = \{2\} \times \{3,5\} \times \{6,7\}$. Suppose A_2 is chosen from $A'' - A_3$ to be added to $A' + A_3$. Since $O(A_2) = \{1,3\}$ intersects only the second domain of C_1, cube C_1 splits in: $C_{11} = part1\,(C_1) = \{2\} \times \{3\} \times \{6,7\}$, $C_{12} = part2\,(C_1) = \{2\} \times \{5\} \times \{6,7\}$.

Hence the node corresponding to C_1 has two branches whose pair of matrices are $A' + A_3 + A_2$ and $A'' - A_3 - A_2$ and whose cubes are respectively C_{11} and $C_{12} \times (O(A_2) \setminus D_{C_1}) = \{2\} \times \{5\} \times \{6,7\} \times \{1\}$ [4]. Consider the branch corresponding to the cube C_{11}. Only row A_1 is left in $A'' - A_3 - A_2$. Since $O(A_1) = \{4,5,7\}$ intersects the third domain of C_{11}, cube C_{11} splits in: $C_{111} = part1\,(C_{11}) = \{2\} \times \{3\} \times \{7\}$, $C_{112} = part2\,(C_{11}) = \{2\} \times \{3\} \times \{6\}$.

Thus the node corresponding to C_{11} has two branches whose pair of matrices are $A' + A_3 + A_2 + A_1$ and $A'' - A_3 - A_2 - A_1$ and whose cubes are respectively C_{111} and $C_{112} \times (O(A_1) \setminus D_{C_{11}}) = \{2\} \times \{3\} \times \{6\} \times \{4,5\}$. The branch corresponding to the cube C_{111} leads to a node at which the first matrix of the pair is equal to A and the second is empty. Moreover cube C_{111} has 3 domains. This means that cube C_{111} contains solutions of A of 3 columns (in this case only one solution): the lower bound cannot be raised to 4 and the solution $\{2\} \times \{3\} \times \{7\}$ is returned as the current best solution.

[3] Notice that we used Equation 5. Applying instead Equation 2, we would obtain:

$$C_4 \times O(A_3) = \{1\} \times \{5\} \times \{7\} \times \{2,3,4,6\},$$

which includes the following additional solutions: $\{1\} \times \{5\} \times \{7\} \times \{2\}$, $\{1\} \times \{5\} \times \{7\} \times \{3\}$, $\{1\} \times \{5\} \times \{7\} \times \{6\}$. In fact, they are all redundant; their irredundant counterparts are respectively: $\{5\} \times \{7\} \times \{2\}$, $\{1\} \times \{7\} \times \{3\}$, $\{1\} \times \{5\} \times \{6\}$, which already appear in $part1(C)$.

[4] We denote by D_{C_1} the union of the domains of cube C_1; we write only D when it is clear which cube is being considered, as before when using D for cube C.

```
find_best_set_of_non_intersecting_rows(A, SolCube) {
  /* Heuristic to find "best" set of rows, */
  /* which do not intersect domains of SolCube. */
  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 every domain 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
}
```

Fig. 4. Algorithm to find the best set of rows not intersecting *SolCube*.

D. Detailed Description of the Raising Algorithm

The procedure *raiser* returns 1 if the lower bound can be raised by n, otherwise it returns 0, meaning 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, for the node 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 the number by which the lower bound *lbound* must be raised. n is an input-output parameter which is initially equal to $ubound - |MSIR| - |path|$ and is decreased every time *raiser* improves the current best solution.
- *lbound* is an input parameter for *raiser* equal to $|MSIR|$. Notice that *lbound* differs from the original lower bound [5] by a quantity equal to $|path|$, for consistency with the previous definition of n.
- *ubound* is the cardinality of the current best solution.
- *bestSolution* is an output parameter, containing a new best solution found by *raiser* if the lower bound could not

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

be raised by n.

Figure 3 shows the flow of *raiser*. 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; thus $part1$ 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* [6], which measures the gap between the upper bound and the current lower bound, is non-positive. If so, then the solutions in $SolCube$ raise the lower bound of A by at least n; hence no solutions of A can beat the current upper bound. The second terminal case occurs when, after some recursive calls, A is empty. Then any solution obtained as the union of a solution of A in $SolCube$ with the columns in the current *path* is the new best solution.

Routine $find\_best\_set\_of\_non\_intersecting\_rows$ is invoked after these preliminary checks and it returns a set of rows of A denoted by $\mathcal{BSONIR}$. This routine, shown in Figure 4, implements a heuristic to find a large 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 $\mathcal{BSONIR}$ 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 longest row that intersects neither a domain of $SolCube$ nor a row previously inserted into $\mathcal{BSONIR}$.

Thereafter, since no row r_i in $\mathcal{BSONIR}$ is covered by any solution encoded in $SolCube$, for each such r_i 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*, checking if its value becomes 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*, $\mathcal{BSONIR}$ is empty because $SolCube$ encodes the $MSIR$ and, by definition, every row not in the $MSIR$ intersects at least one row in the $MSIR$. However, during the recursive calls of *raiser* the original domains of $SolCube$ may decrease in cardinality due to *split_cubes* and $add\_set\_of\_1intersecting\_rows(A, SolCube)$. Hence, it may happen that a row of A is not covered 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 raised already the lower bound by $n-1$. Therefore, if we are forced to add one more domain

[6] By definition $stillToRaise = lbound + n - numberDomains(SolCube) = |MSIR| + ubound - |MSIR| - |path| - numberDomains(SolCube) = ubound - |path| - numberDomains(SolCube)$.

```
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
}
```

Fig. 5. Algorithm to handle the 1-intersecting rows.

to $SolCube$, then we can prune the current branch. For example, 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 $d = \{c^1, c^2, \cdots, c^l\}$. Suppose r_1 intersects only c^i, while r_2 intersects only c^j. This fact allows us to prune the current branch. Indeed assume to cover r_1 by means of column 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.

Routine $add\_set\_of\_1intersecting\_rows$, which exploits the previous situation, is illustrated in Figure 5. In practice, it is invoked often because the condition $stillToRaise = 1$ happens very commonly in hard problems. The routine is based on two nested cycles. The external cycle is repeated until the internal cycle does not modify $SolCube$. 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 equals 1, e.g., $D = \{d\}$, we remove from d all 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 removing from A the set $\mathcal{BSONIR}$ of non-intersecting rows, 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, a new row r_i is selected to be covered with $SolCube$. Row r_i is removed from A and drives the splitting of $SolCube$. The strategy to select the best row is to look for the row of A which intersects the minimum number of domains of $SolCube$. The rationale is to reduce the number of branches from the node [7]. 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 $D'_{i_k} = D_{i_k} \setminus O(A_p)$. Thus the weight is just the fraction of solutions from $SolCube$ that do not cover A_p. If $D'_{i_k} = \emptyset$ for some k, then row A_p is covered by any solution from $SolCube$. Hence, A_p is simply removed from A'' and added to A'.

The splitting of $SolCube$ is done as explained in Section III. Then, *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 one or more times, *raiser* returns 0 after having updated the current best solution and upper bound.

V. Experimental Results

As discussed in the previous sections, *raiser* can be implemented on top of any existing standard B&B procedure. We made two distinct implementations of *raiser* starting from two well-known UCP solvers, namely *mincov* (used in ESPRESSO) and SCHERZO. The *mincov* routine has represented the *state-of-the-art* in solving UCP problems for over ten years and was strongly outperformed only recently by the arrival of SCHERZO [7], [3], [8]. SCHERZO exploits a collection of new lower bounds (easy lower bound, logarithmic lower bound, left hand side lower bound, limit lower bound), and partition-based pruning. By enhancing both of these programs with the negative thinking idea, we obtained two new search engines, which are much more efficient than the original ones [9], [10]: they are called respectively AURA = ESPRESSO + RAISER and AURA II = SCHERZO + RAISER.

In this section we show the dramatic impact of the negative thinking paradigm in both cases. Table I gives the results obtained comparing ESPRESSO and AURA, while Tables II and III report experiments comparing SCHERZO and AURA II. The benchmarks used belong to three classes: (1) a set of difficult cases from the collection of ESPRESSO two-level minimization problems (we consider as input the unate matrix which is obtained after removing the essential primes), (2) three matrices encoding constraints satisfaction problems from [11], and (3) a set of random generated matrices with varying row/column ratios and densities (e.g., $m200\_100\_30\_70$ 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, the size ($R \times C$ in the tables) and sparsity (S in the tables) are reported. The experiments were performed on a 1GB 625Mhz Alpha with timeout set to 24 hours of cpu time.

The tables report two types of data for comparison: the number of nodes of the column branching computation tree and the running time in seconds. There are several points to be explained concerning the number of nodes:

1. Both AURA and AURA II have 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). As explained in Section II, these search engines apply the negative thinking approach by following a dual strategy: they start building the column branching computation tree, but when at a given node the difference between the upper bound and the lower bound is less or equal to the raising parameter *maxRaiser* they call the *raiser* procedure, which builds a cube branching computation tree, appended at the node where *raiser* was called. Thus to measure correctly a run of AURA or AURA II, both numbers of nodes need to be reported.
2. Nodes for cube branching usually take much less computing time than those for column branching, even though it is not known a-priori a time ratio between the two types of nodes. The reason is that expensive procedures for finding dominance relations and the $MSIR$ are applied in column branching.
3. The raising parameter *maxRaiser* (label r in the tables) is an input to both AURA and 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 is always faster than ESPRESSO and similarly for AURA II and SCHERZO. Indeed the speedup increases for the most time-consuming examples.

Table I reports the experimental results for ESPRESSO versus AURA with the raising parameter set always to 3: ESPRESSO is not able to compute the solutions of benchmarks *ex5, max1024, prom2* in the allotted time, while for the benchmark *saucier* the computation does not complete with the available memory. These benchmarks represent the most difficult problems in our benchmark suite and for all of them AURA completes. Considering the random benchmarks, the comparison between AURA and ESPRESSO illustrates the strong superiority of the former.

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

For each of the difficult cases reported in Table II, 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 a 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 seconds 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 I: 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 III, for which we set the raising parameter r always to 3. In the most time-consuming of these examples AURA II takes between 36% and 75% of the time of SCHERZO.

A. Other Comparisons

We do not have a systematic comparison with the results by BCU, a very efficient recently-developed ILP-based covering solver [6]. 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 [6] to run BCU on *saucier*, whose solution space is poorly diversified (a minimum solution has 6 columns, while most of the irredundant solutions have costs 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 AURAII 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.

VI. Conclusions

We introduced a new technique to solve exactly a discrete optimization problem. The motivation is that often a good solution is reached quickly and then it is improved only a few times before the optimum is found; hence most of the solution space is explored to certify optimality, with no improvement of the cost function. This suggests that more powerful lower bounding would speed up the search dramatically. Therefore, the search strategy was modified with the goal of proving that the given subproblem cannot yield a solution better than the current best one (negative-thinking), instead of branching further in search for a better solution (positive thinking).

For illustration we have applied our technique to UCP, usually solved exactly by a branch-and-bound procedure, with an independent set of columns as a lower bound, and branches on columns. We designed a dual search technique, called *raiser*, which is invoked when the difference between the upper bound and the lower bound is within a parameter *maxRaiser*, set by the user. The procedure *raiser* tries to detect a hard core of the matrix to be solved (lower bound submatrix), augmenting an independent set of rows in order to increase incrementally the cardinality of the minimum solutions that cover it. Eventually either this incremental raise yields a lower bound that matches the current upper bound, and so we are done with this matrix, or we produce at least one better solution. The selection of a next row induces the recomputation of all the solutions of the lower bound submatrix augmented by the next row, as disjoint cubes of solutions. Each such cube together with the augmented matrix defines a new node of the computation tree explored by *raiser*.

A key technical contribution to implement negative thinking for UCP is the introduction of the data structure of *cubes of solutions*, inspired by multi-valued cubes. Applying the operator *Rec* to a cube of solutions one obtains a collection of cubes of solutions, thereby providing a natural clustering of the recomputed solutions. Clustering allows us to design a recursive algorithm based on branching in subsets of solutions and to raise independently the lower bound starting from different subsets of solutions.

The procedure *raiser* can be implemented on top of any existing B&B procedure. We did this for ESPRESSO and SCHERZO obtaining in both cases new search engines (respectively, AURA and AURA II) that are much more efficient.

Future work includes application to the binate covering problem. A more basic line of research is the exploration of data structures different from cubes of solutions, but still enjoying their properties of offering representations that are compact and easy to update.

Acknowledgments

We gratefully thank Dr. Olivier Coudert (Monterey Design Systems) who kindly provided a version of SCHERZO and was always available for technical discussions. We also gratefully thank Stan Liao (Synopsys) for his assistance in running BCU on some benchmarks.

References

[1] C.H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice Hall, 1982.
[2] R. Rudell and A. Sangiovanni-Vincentelli, "Multiple-valued minimization for PLA optimization," *IEEE Transactions on Computer-Aided Design*, vol. CAD-6, pp. 727–750, Sept. 1987.
[3] O. Coudert, "On solving binate covering problems," in *The Proceedings of the Design Automation Conference*, June 1996, pp. 197–202.
[4] T. Kam, T. Villa, R. K. Brayton, and A. Sangiovanni-Vincentelli, *Synthesis of FSMs: functional optimization*, Kluwer Academic Publishers, 1997.
[5] M.R. Garey and D.S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-completeness*, W. H. Freeman and Company, 1979.
[6] S. Liao and S. Devadas, "Solving covering problems using LPR-

based lower bounds," in *The Proceedings of the Design Automation Conference*, June 1997.

[7] O. Coudert, "Two-level logic minimization: an overview," *Integration*, vol. 17-2, pp. 97–140, Oct. 1994.

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

[9] E. I. Goldberg, L. P. Carloni, T. Villa, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Negative Thinking by Incremental Problem Solving: Application to Unate Covering," in *The Proceedings of the International Conference on Computer-Aided Design*. Nov. 1997, pp. 91–98, IEEE.

[10] L. P. Carloni, E. I. Goldberg, T. Villa, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Aura II: Combining Negative Thinking and Branch-and-Bound in Unate Covering Problems," in *VLSI: Systems on a Chip*, S. Devadas, R. Reis and L.M. Silveira, Ed., Lisboa - Portugal, Dec. 1999, Kluwer Academic Publishers.

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

Appendix

I. Appendix: Representation and Recomputation of the Solutions

A. Recomputation of the Solutions

Theorem A.1: For any irredundant solution S^* of $UCP(A' + A_p)$ there is an irredundant solution S of $UCP(A')$ such that S^* is an element of $Rec(A' + A_p, S)$.

Proof: Let S^* be an irredundant solution of $UCP(A'+A_p)$. Clearly S^* is a solution of $UCP(A')$. There are two cases:

1. S^* is irredundant for $UCP(A')$ too. In this case we are done, noticing that $S^* \in Rec(A' + A_p, S^*)$, given that $Rec(A' + A_p, S^*) = \{S^*\}$.

2. S^* is redundant for $UCP(A')$. We show first that in this case there is only one redundant column and this is a column covering A_p.

(a) We prove that all redundant columns must cover A_p. Indeed a column of S^* is irredundant if and only if it covers a row not covered by others columns. Any column j in S^* not covering A_p cannot be redundant for $UCP(A')$, since S^* is irredundant for $UCP(A' + A_p)$. Indeed, if j is redundant for $UCP(A')$ and does not cover A_p, then it remains redundant for $UCP(A' + A_p)$.

(b) So far we know that there is at least one redundant column and that it must cover A_p, as all redundant columns do. We prove that it cannot be the case that two (or more) columns cover A_p. Indeed, if two columns cover A_p and one of them is redundant for $UCP(A')$, then it remains redundant for $UCP(A' + A_p)$ (the column cannot become irredundant because there is no row in $A'+A_p$ covered only by it), which contradicts the condition that S^* is irredundant for $UCP(A' + A_p)$.

So S^* can be represented as $S \cup \{j\}$, where j is redundant for $UCP(A')$ and it is the only column from S^* covering A_p, and S is an irredundant solution of $UCP(A')$ not covering A_p. Moreover, by definition of the Rec operation any solution of $UCP(A'+A_p)$ represented as $S\cup\{j\}$, where S is an irredundant solution to $UCP(A')$ not covering A_p and $j \in O(A_p)$, is also in $Rec(A' + A_p, S)$. So we conclude that for any irredundant solution S^* of $UCP(A' + A_p)$ there is an irredundant solution S of $UCP(A')$ such that S^* is an element of $Rec(A' + A_p, S)$. ■

Theorem A.2: If the computation of the Rec operator is modified as follows:

$$\begin{aligned} Rec(A' + A_p, C) &= part1(C) \cup \{\, part2(C) \times \\ &\times \; [O(A_p) \setminus (D_1 \cup \cdots \cup D_d)] \,\} \end{aligned}$$

no irredundant solution of $A' + A_p$ is discarded.

Proof: Let $C = D_1 \times \cdots \times D_d$ be the cube of solutions and A_p the row to be added. Without loss of generality assume that A_p intersects the first r domains of C, $r \le d$.

By construction $part1(C) = C_1 \cup \cdots \cup C_r$, where $C_k = D'_1 \times \cdots \times D'_{k-1} \times D''_k \times D_{k+1} \times \cdots \times D_d$, $1 \le k \le r$, $D'_i = D_i \setminus O(A_p)$, $1 \le i \le k-1$, and $D''_k = D_k \cap O(A_p)$, $1 \le k \le r$. Moreover, $part2(C) = D'_1 \times \cdots \times D'_r \times D_{r+1} \times \cdots \times D_d$, where $D'_i = D_i \setminus O(A_p)$, $1 \le i \le r$.

If we prove that any solution from the cube $C^* = part2(C) \times (O(A_p) \cap D)$ is redundant, where $D = D_1 \cup \cdots \cup D_d$, we are allowed to replace the computation of $part2(C) \times O(A_p)$ with the computation of $part2(C) \times (O(A_p) \setminus D)$.

By distributivity of the Boolean operators $\cup$ and $\cap$, and the fact that A_p intersects only the first r domains of C, it is $D \cap O(A_p) = D''_1 \cup \cdots \cup D''_r$, and so cube C^* can be rewritten as follows:

$$\begin{aligned} C^* &= part2(C) \times (D \cap O(A_p)) \\ &= part2(C) \times (D''_1 \cup \cdots \cup D''_r) \\ &= part2(C) \times D''_1 \cup \cdots \cup part2(C) \times D''_r \end{aligned}$$

and therefore C^* can be represented as the union $C^*_1 \cup \cdots \cup C^*_r$ where $C^*_k = part2(C) \times D''_k$, $1 \le k \le r$.

Now define the cubes C'_k, $1 \le k \le r$, obtained from $part2(C)$ by replacing in turn D'_k with D''_k. Cubes C'_k and C_k - which have the same number of domains - by construction are such that cube C_k (obtained from $part1(C)$) contains cube C'_k (obtained from $part2(C)$), as shown by a component-wise comparison, using the fact that $D'_{k+1} = D_{k+1} \setminus O(A_p), \cdots, D'_r = D_r \setminus O(A_p)$:

$$\begin{aligned} C_k &= D'_1 \times \cdots \times D'_{k-1} \times D''_k \times D_{k+1} \times \cdots \\ &\quad \cdots \times D_r \times D_{r+1} \times \cdots \times D_d \\ C'_k &= D'_1 \times \cdots \times D'_{k-1} \times D''_k \times D'_{k+1} \times \cdots \\ &\quad \cdots \times D'_r \times D_{r+1} \times \cdots \times D_d. \end{aligned}$$

Consider the k-th component, for $1 \le k \le r$, of the representation of cube C^* as $C^* = C^*_1 \cup \cdots \cup C^*_r$:

$$\begin{aligned} C^*_k &= part2(C) \times D''_k \\ &= D'_1 \times \cdots \times D'_{k-1} \times D'_k \times D'_{k+1} \times \cdots \\ &\quad \cdots \times D'_r \times D_{r+1} \times \cdots \times D_d \times D''_k \end{aligned}$$

and permute the domains D'_k (from $part2(C)$) and D''_k

$$\begin{aligned} C^*_k &= D'_1 \times \cdots \times D'_{k-1} \times D''_k \times D'_{k+1} \times \cdots \\ &\quad \cdots \times D'_r \times D_{r+1} \times \cdots \times D_d \times D'_k \\ &= C'_k \times D'_k. \end{aligned}$$

Therefore any solution S from C_k^* consists of a set of columns $S' \in C_k'$ and a column $j \in D_k'$. Since C_k contains C_k' (as shown earlier) and by construction C_k is made of solutions of A' which cover also A_p, then S' covers both A' and A_p and so column j is redundant in the solution $S = S' \cup \{j\}$. So any solution from C_k^* is redundant for $1 \leq k \leq r$. ■

B. Avoiding Generation of Repeated Solutions

Given $UCP(A)$, suppose that $C = D_1 \times D_2 \times \cdots \times D_d$ is the cube of solutions of $UCP(A')$, where A' is a subset of rows of A. Then add row A_p, which, say, intersects only the domain D_1. As argued in Section III, the solutions of $A' + A_p$ are found by

$$Rec(A' + A_p, C) \quad = \quad C_1 \quad \cup \quad C_2 \times O^*(A_p),$$

where

$$\begin{aligned} C_1 &= D_1' \times D_2 \times \cdots \times D_d, \\ C_2 &= D_1'' \times D_2 \times \cdots \times D_d, \\ D_1' &= D_1 \cap O(A_p), \\ D_1'' &= D_1 \setminus O(A_p), \\ O^*(A_p) &= O(A_p) \setminus D_1. \end{aligned}$$

Now let $S = (j_1, j_2, \cdots, j_d)$ be a solution from C_1 and $S' = (j_1', j_2, \cdots, j_d, j_{d+1})$ be a solution from $C_2 \times O^*(A_p)$, which differs from S_1 only by replacing j_1 with j_1' and by adding j_{d+1} from $O^*(A_p)$. Suppose that there is a solution S'' of $UCP(A)$ containing the partial solution $S \cup S'$. Then the same solution S'' may be constructed both from the branch of cube C_1 and the branch of cube $C_2 \times O^*(A_p)$. In general this means that a solution may be generated more than once.

The reason is that, even though when forming D_1'' we remove from D_1 the columns covering A_p, still it is possible to extend solutions from C_1 by adding columns from $D_1 \setminus O(A_p)$ and $O^*(A_p)$ and to extend solutions from $C_2 \times O^*(A_p)$ by adding columns from $D_1 \cap O(A_p)$, so that we may obtain from both branches the same partial solution from $D_1 \cap O(A_p) \times D_1 \setminus O(A_p) \times D_2 \times \cdots \times D_d \times O^*(A_p)$.

To eliminate this possibility it is sufficient to avoid the consideration of solutions containing columns from $D_1 \cap O(A_p)$ in the branch of cube $C_2 \times O^*(A_p)$. Indeed, if we do so, a solution containing the partial solution $S \cup S'$ can be found only in the branch of cube C_1, because in the branch of C_2 solutions containing columns from $D_1 \cap O(A_p)$ are not considered, whereas $S \cup S'$ contains such a column, i.e., column j_1. In general, if A_p intersects the first r domains of C, in the branch of cube $C_k, 1 \leq k \leq r+1$, where C_k contains $k-1$ domains $D_i \setminus O(A_p), i = 1, \cdots, k-1$, we should avoid the generation of solutions containing columns from $(D_1 \cup D_2 \cup \cdots \cup D_{k-1}) \cap O(A_p)$.

			ESPRESSO		AURA			time
matrix	$R \times C(S)$	Sol.	nodes	time	nodes/A-nodes	time	r	speedup
ex5	831 × 2428 (2.04%)	37	-	time	155/169245	1315.2	3	∞
lin.rom	1030 × 1076 (0.9%)	120	370	29.1	61/240	7.7	3	3.78
max1024	1090 × 1264 (0.52%)	245	-	time	12402/3850628	36240.0	3	∞
mlp4	530 × 594 (0.99%)	109	2122	22.6	34/206	1.3	3	17.39
prom2	1924 × 2611 (0.31%)	278	-	time	1478/1097624	24071.4	3	∞
ex4inp	91 × 240 (46.03%)	5	5279	16.81	9/14	0.27	3	62.26
maincont	105 × 67 (34.51%)	7	504	0.69	11/12	0.06	3	11.50
saucier	171 × 6207 (47.17%)	6	-	mem	10/76	222.47	3	∞
m100_100_10_10	100 × 100 (10%)	12	-	time	5043/201091	129.3	3	∞
m100_100_30_30	100 × 100 (30%)	5	116307	792.8	35/1108	2.5	3	317.12
m100_100_50_50	100 × 100 (50%)	4	66147	316.4	7/1030	1.6	3	197.75
m100_100_70_70	100 × 100 (70%)	3	5083	171.0	3/1	1	3	171.00
m100_100_90_90	100 × 100 (90%)	2	175	21.2	5/112	0.7	3	30.29
m100_50_10_10	100 × 50 (20%)	8	12466	59.6	94/2529	2.9	3	20.55
m100_50_20_20	100 × 50 (40%)	5	16905	49	31/951	1.7	3	28.82
m100_50_30_30	100 × 50 (60%)	3	947	9.5	5/26	0.3	3	31.67
m100_50_40_40	100 × 50 (80%)	2	73	4.3	3/1	0.3	3	14.33
m50_100_10_10	50 × 99 (10.1%)	8	843	3.3	17/166	0.1	3	33.00
m50_100_30_30	50 × 100 (30%)	4	12047	37.8	11/203	0.2	3	189.00
m50_100_50_50	50 × 100 (50%)	3	2569	13.9	5/32	0.1	3	139.00
m50_100_70_70	50 × 100 (70%)	2	135	3.5	3/1	0.1	3	35.00
m50_100_90_90	50 × 100 (90%)	2	135	2.6	3/1	0.1	3	26.00

TABLE I

EXPERIMENTAL RESULTS (ESPRESSO VS. AURA).

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

TABLE II

RESULTS ON *Espresso* BENCHMARKS (SCHERZO VS. AURA II).

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

TABLE III

RESULTS ON RANDOM BENCHMARKS (SCHERZO VS. AURA II).

Evguenii I. Goldberg Evgueni I. Goldberg received the M.S. degree in physics from the Belorussian State University in 1983 and the Ph.D. degree in computer science from the Institute of Engineering Cybernetics of the Belorussian Academy of Sciences in 1995. From 1983 to 1995 he worked at the laboratory of logic design of the Institute of Engineering Cybernetics as a Researcher. From 1996 to 1997 he was a visiting scholar at the University of California at Berkeley. Currently, he works at the Cadence Berkeley Labs as a Research Scientist. His areas of interests include logic design, test, and verification.

Luca P. Carloni received the Laurea degree in Electrical Engineering *summa cum laude* from the University of Bologna, Italy in July 1995, and the M.S. degree in Electrical Engineering and Computer Sciences from the University of California at Berkeley in December 1997. He is currently a Ph.D. candidate in the Department of Electrical Engineering and Computer Sciences of the University of California at Berkeley. His research interests are in the area of computer-aided design of electronic systems and include embedded system design, high level synthesis, logic synthesis, and combinatorial optimization.

Tiziano Villa studied mathematics at the universities of Milano, Pisa and Cambdrige, U.K. and electrical engineering and computer science at the University of California, Berkeley, where he completed a Ph.D. in EECS in 1995. He worked in the integrated circuits division of the CSELT Labs, Torino Italy, as a computer-aided design specialist, and then for many years he was a research assistant at the Electronics Research laboratory, University of California, Berkeley. In 1997 he joined the Parades Labs, Rome, Italy. His research interests include logic synthesis, formal verification, combinatorial optimization, automata theory and hybrid systems. His contributions are mainly in the area of combinational and sequential logic synthesis. In May 1991 he was awarded the Tong Leong Lim Pre-doctoral Prize at the EECS Department of the University of California, Berkeley. He co-authored the books "*Synthesis of FSMs: functional optimization*", Kluwer, 1997 and "*Synthesis of FSMs: logic optimization*", Kluwer 1997.

Robert K. Brayton received the BSEE degree from Iowa State University in 1956 and the Ph.D. degree in mathematics from MIT in 1961. From 1961 to 1987 he was a member of the Mathematical Sciences Department of the IBM T. J. Watson Research Center. In 1987 he joined the EECS Department at Berkeley, where he is a Professor and director of the SRC Center of Excellence for Design Sciences. He has authored over 300 technical papers, and seven books, "Computer Aided Design: Sensitivity and Optimization", "Logic Minimization Algorithms for VLSI Synthesis", "Integrating Functional and Temporal Domains in Logic Design", "Timed Boolean functions: A Unified Formalism for Exact Timing analysis", "Logic Synthesis for Field-Programmable Gate Arrays" "Synthesis of Finite State Machines: Functional Optimization" "Synthesis of Finite State Machines: Logic Optimization" Dr. Brayton holds the Edgar L. and Harold H. Buttner Endowed Chair in Electrical Engineering in the EECS Department at Berkeley. He is a member of the National Academy of Engineering, and a Fellow of the IEEE and the AAAS. He received the 1991 IEEE CAS technical achievement award, the 1971 Guilleman-Cauer award, and the 1987 Darlington award. He was the editor of the Journal on Formal Methods in Systems Design from 1992-1996. Past contributions have been in analysis of nonlinear networks, and electrical simulation and optimization of circuits. Current research involves combinational and sequential logic synthesis for area/performance/testability, asynchronous synthesis, and formal design verification.

Alberto L. Sangiovanni-Vincentelli holds the Edgar L. and Harold H. Buttner Chair of Electrical Engineering and Computer Sciences at the University of California at Berkeley where he has been on the Faculty since 1976. He obtained an electrical engineering and computer science degree ("Dottore in Ingegneria") summa cum laude from the Politecnico di Milano, Milano, Italy in 1971. In 1980-1981, he spent a year as a Visiting Scientist at the Mathematical Sciences Department of the IBM T.J. Watson Research Center. In 1987, he was Visiting Professor at MIT. He was a co-founder of Cadence and Synopsys, the two leading companies in the area of Electronic Design Automation. He was a Director of ViewLogic and Pie Design System and Chair of the Technical Advisory Board of Synopsys. He is the Chief Technology Advisor of Cadence Design System. He is a member of the Board of Directors of Cadence, Sonics Inc., and Accent. He is the founder of the Cadence Berkeley Laboratories and of the Cadence European laboratories. Dr. Sangiovanni-Vincentelli was the founder of the Kawasaki Berkeley Concept Research Center, where he holds the title of Chairman of the Board. He has consulted for a number of US companies including IBM, Intel, ATT, GTE, GE, Harris, Nynex, Teknekron, DEC, HP, Japanese companies including Kawasaki Steel, Fujitsu, Sony and Hitachi, and European companies including SGS-Thomson Microelectronics, Alcatel, Daimler-Benz, Magneti-Marelli, BMW, Bull. He is the Scientific Director of the Project on Advanced Research on Architectures and Design of Electronic Systems (PARADES), a European Group of Economic Interest. He is on the Advisory Board of the Lester Center of the Haas School of Business and of the Center for Western European Studies and a member of the Berkeley Roundtable of the International Economy (BRIE). In 1981 he received the Distinguished Teaching Award of the University of California. He received the worldwide 1995 Graduate Teaching Award of the IEEE (a Technical Field award for "inspirational teaching of graduate students"). He has received numerous awards including the Guillemin-Cauer Award (1982-1983) and the Darlington Award (1987-1988). He is an author of over 480 papers and ten books in the area of design methodologies, large-scale systems, embedded controllers, hybrid systems and tools. Dr. Sangiovanni-Vincentelli is a Member of the National Academy of Engineering. Dr. Sangiovanni-Vincentelli was the Technical Program Chairperson of the International Conference on CAD and his General Chair. He was the Executive Vice-President of the IEEE Circuits and Systems Society.