Negative Thinking by Incremental Problem Solving: Application to Unate Covering

Evguenii I. Goldberg[†,‡] Luca P. Carloni[‡] Tiziano Villa[§] Robert K. Brayton[‡]

Alberto L. Sangiovanni-Vincentelli[‡]

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

[†] Academy of Sciences of Belarus, Minsk

[§] PARADES, Via di S.Pantaleo, 66, 00186 Roma

Abstract

We introduce a new technique to solve exactly a discrete optimization problem, based on the paradigm of "negative" thinking. The motivation is that when searching the space of solutions, 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, but it does not yield any improvement of the cost function. So it is quite natural for an algorithm to be "skeptical" about the chance to improve the current best solution.

For illustration we have applied our approach to the unate covering problem. We designed a procedure, $raiser$, implementing a negative thinking search, which is incorporated into a common branch-and-bound procedure. $Raiser$ is invoked at a node of the search tree which is deep enough to justify negative thinking.

Raiser tries to detect a hard core of the matrix corresponding to the node by augmenting an independent set of rows in order to increase incrementally the cost of the minimum solutions covering the matrix. Eventually either $raiser$ prunes the subtree rooted at the node (having found a lower bound equal or greater than the current best solution) or returns a new solution that becomes the current best one.

Experiments show that our program, AURA, outperforms both ESPRESSO and our enhancement of ESPRESSO using Coudert's limit lower bound [3]. It is always faster and in the most difficult examples either has a running time better by up to two orders of magnitude, or the other programs fail to finish due to timeout or spaceout. The package SCHERZO is faster on some examples and loses on others, due to a less powerful pruning strategy of the search space, partially mitigated by a more effective computation of the maximal independent set.

1 Introduction

A common approach to find an exact solution to problems in combinatorial optimization is branch-and-bound (BAB), which improves over exhaustive enumeration, because it avoids the exploration of some regions of the solution space, when it can certify by means of lower bounds that they do not contain a solution better than the current best one.

To ground the exposition in a concrete domain, in this paper we consider BAB applied to the solution of the Unate Covering Problem (UCP), that is of great interest in logic synthesis and operations research [4]. For the sake of simplicity we consider the case of UCP where all columns have the same cost. Such version of UCP is defined as follows. *Given a Boolean matrix A (all entries are 0 or 1), find a minimum size subset of columns of A such that every row of A is covered by at least one column of the subset.* A row i is covered by a column j if $A_{ij} = 1$. We will denote an instance of UCP with matrix A as $UCP(A)$.

An exact solution of UCP is typically obtained by a branch-and-bound recursive algorithm, which has been implemented in successful computer programs [7, 6]. 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, call it *mincov*, 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 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; we denote by $path(A_N)$ the set of columns included in the partial solution.

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 $UCP(A_N)$. So the size of any solution of $UCP(A)$ including the columns in $path(A_N)$ is greater or equal to $L(A_N) + |path(A_N)|$. So 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 branch-and-bound 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 disprove that there is a better solution in the tree rooted at A_N.

To compare the role of "negative" and "positive" ways of search, notice that at the n-th level of the computation tree we can have up to 2^n nodes, i.e., subproblems. It is an experimental fact that usually in the first leaf a solution very close to the 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 2^n subproblems. 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 "negative" search is justified. In turn this is related to how much the solution space is "diversified", i.e., different solutions have different costs. Notice that BAB uses "negative thinking" in optimization problems by finding lower bounds, and in decision problems by checking the consistency of the partial solution with the current subproblem.

To exploit both "positive" and "negative" search, BAB is modified as follows. We 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", each subproblem is solved in the "negative thinking" mode, called NT-mode. In optimization problems 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 is to switch to the NT-mode.

Let P be a subproblem to be solved in NT-mode and suppose that if the cost of P is greater than a given $ubound$ then solving P cannot give a better solution (w.l.o.g., assume to solve a minimization problem). The aim of the algorithm in the NT-mode is to prove that there is no solution of P with cost less than $ubound$.

We propose a new way to implement "negative thinking": incremental problem solving (IPS). When solving a problem A incrementally, we start with a subproblem A' of A, such that the solutions of A' can be represented compactly. Then we modify gradually A' by making it more complex to come closer to the full problem A and we recompute the set of solutions of the modified problem. When applying "negative thinking", we try to find first the most difficult "obstacles" in the sequence from A' to A with the goal to prove that no solution of A' can overcome the obstacles and be extended to a solution of A.

More precisely, let P' be a subproblem of P such that its set of solutions $Sol(P')$ can be represented in a compact form. Each solution of P' from $Sol(P')$ can be considered as a seed from which one may grow some solutions of P. In the NT-mode, the algorithm tries to show that no solution of P with $cost(P) < ubound$ can grow from any solution $S \in Sol(P')$. A naive approach is to form a sequence of problems $P_1, \cdots, P_n$, where $P_1 = P'$ and $P_n = P$. At each step one recomputes $Sol(P_i)$ starting from $Sol(P_{i-1})$ and discards all solutions in $Sol(P_i)$ with a cost greater than $ubound$. If, after removing the solutions costing more or as $ubound$, $Sol(Pi) = \emptyset$, for some P_i, $i \leq n$, then there is no solution of P with cost less than $ubound$. A direct implementation of this approach has two drawbacks:

1. The size of the representation of $Sol(P_i)$ may grow exponentially.
2. There are different ways of approaching P from P'. Each specific seed solution $S \in Sol(P')$ is extended more quickly to a solution costing more or as $ubound$ by a specific sequence of augmentations, different from those appropriate for another solution $\hat{S} \in Sol(P')$.

As a remedy we propose the paradigm of clusterization of solutions. We group in a cluster the solutions that are similar, in the sense of having the same witnesses of the fact that they cannot produce solutions of P costing less than $ubound$.

In this paper we present an incremental UCP solver called *raiser*. Although we demonstrate our technique on UCP it can be applied to any discrete optimization problems with a monotone cost function, i.e., for which a minimum solution of a subproblem has a smaller cost than that of the initial problem.

The paper is organized as follows. Section 2 shows how an incremental solver is incorporated into the standard branch-and-bound procedure for UCP. The idea of incremental improvement of the lower bound is sketched in Section 3, while Section 4 describes how the solutions of UCP are represented and recomputed. The raising procedure is explained in detail in Section 5 and experimental results are discussed in Section 6. Conclusions are given in Section 7.

2 Incorporating an Incremental Solver into Branch-and-Bound

The flow of a UCP solver based on branch-and-bound is shown in Fig. 1. The parts of text in bold font refer to the incremental solver and will be explained below. For details the reader is referred to [4]. Given a matrix A, existing UCP solvers employ column branching to decompose the problem and use a maximal set of independent (non-intersecting) rows ($MSIR$) to compute a lower bound of $UCP(A)$ (since no column covers two rows from $MSIR$) [1].

```
branch_and_bound(A, Sol, n) {
    /* A = matrix of UCP, Sol = current (partial) solution */
    /* n = "range" of raiser, best = best current solution */
    if (A = ∅)
        return(Sol) /* new best solution */
    /* Column and row dominance */
    simplify(A)
    /* Lower bound evaluation */
    MSIR = find_msir(A)
    if ((lower_bound(A) + cost(Sol)) ≥ cost(best))
        return(∅)
    /* Is the current node within the range of raiser ? */
    if ((|MSIR| + cost(Sol) + n) ≥ cost(best)) {
    /* n' exact amount to raise */
        n' = cost(best) − (|MSIR| + cost(Sol))
        return(raiser(A, MSIR, n'))
    }
    /* select a branching column */
    j = select_column(A)
    /* Decomposition: A1(A2) for including (not including) j
    in solution */
    Sol1 = Sol ∪ {j}
    Sol2 = Sol
    for (i = 1; i ≤ 2; i++)
        {New = branch_and_bound(Ai, Soli, n)
        if (cost(New) < cost(best)) {
            best = New
            if (cost(best) ≤ (cost(Sol) + |MSIR|)
                return(best)
        }
    }
    return(best)
}
```

Figure 1: Branch-and-Bound enhanced by incremental solver

A procedure n-raiser, performing "negative thinking", is invoked when $MSIR$ is a lower bound not sufficient to prune the subtree rooted at the current node, but increasing the lower bound by n would allow such pruning. The n-raiser starts from the subproblem $UCP(MSIR)$ whose solution space is very regular and then tries to extend it gradually to A. The n-raiser either returns a minimum cost solution of $UCP(A)$, if the lower bound cannot be raised by n, or returns the empty solution.

[1]A lower bounding technique based on linear programming relaxation, as commonly done in ILP, has been tested successfully for solving covering problems and reported in [5].

The parameter n is specified a-priori and is the same for all invocations of *raiser* in the column branching mode. The value of n is usually a small number in the range from 2 to 4 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 one can make use of the fact that $UCP(MSIR)$ has a regular solution space.

Note that improving the lower bound even by a small amount may lead to considerable runtime reductions. For example, in [1] it was reported a new technique for pruning the search tree called limit lower bound. Sometimes the technique allows to reduce the search tree size by ten times. It can be shown that the limit lower bound technique prunes no more branches of the search tree than *1-raiser*.

The detailed description of the raiser is given in Sections 3, 4 and 5.

3 Incremental Improvement of the Lower Bound

Given an optimization problem such that for any subproblem the cost of a minimum solution of the problem is greater than or equal to that of the subproblem, the size of a minimum solution of the subproblem gives a **lower bound** on the the size of a minimum solution of the problem (called **cost monotonicity assumption**). This fact is of practical interest if it is not difficult to find a minimum solution of the subproblem.

Denote by $min(UCP(A))$ the size of a minimum solution of $UCP(A)$ and let A' be a submatrix of matrix A, consisting of some rows of A, i.e., $Col(A) = Col(A')$ and $Row(A') \subseteq Row(A)$. Any $UCP(A')$ where A' is a submatrix of A satisfies the cost monotonicity assumption, since $min(UCP(A')) \le min(UCP(A))$. We call **lower bound submatrix** a submatrix A' whose minimum solution is used for evaluating a lower bound for $UCP(A)$. If A' is a $MSIR$ then $min(UCP(A')) = |Row(A')|$. We are now going to describe the idea underlying the method for an incremental improvement of the lower bound.

Denote by $A' + A_r$ the submatrix of A obtained by adding to A' a row $A_r \in Row(A) \setminus Row(A')$. Let S be a solution of $UCP(A)$. A column $j \in S$ is called **redundant** if $S \setminus \{j\}$ is also a solution of $UCP(A)$. If a solution of $UCP(A)$ does not contain redundant columns then it is said to be **irredundant**. Denote by $Sol(A', m)$ the set of solutions of $UCP(A')$ which includes all the irredundant solutions consisting of m or fewer columns. So $Sol(A', m)$ contains all the irredundant solutions of size from $min(UCP(A'))$ to m columns. So if $m = min(UCP(A'))$ then $Sol(A', m)$ gives exactly the set of all minimum solutions of $UCP(A')$.

Suppose that for a lower bound submatrix A' of A we know a set of solutions $Sol(A', m)$. The lower bound given by A' is equal to $m = min(UCP(A'))$. Let us 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 having added a set of rows $A_{i_1}, .., A_{i_k}$ of A to A', we can reach a stage when $Sol(A' + A_{i_1} + .. + A_{i_k}, m) = \emptyset$, meaning that we improved the lower bound for $UCP(A)$ by 1 taking as a lower bound the submatrix $A' + A_{i_1} + .. + A_{i_k}$. If $Sol(A' + A_{i_1} + .. + A_{i_k}, r) = \emptyset, r \ge m$ we improved the lower bound by $r - m + 1$.

So an attractive idea is to start from a submatrix A' which is an $MSIR$ (since the solutions of an $MSIR$ can be represented compactly) and then to add rows to the $MSIR$ with the goal to improve the initial lower bound given by $|MSIR|$. The proposal relies on the intuition that, knowing $Sol(A', m)$, it is not difficult to recalculate $Sol(A' + A_p, m)$ and, adding one row at a time, eventually we may reach the desired lower bound improvement. In Section 4.1 we discuss how to recalculate solutions. This "naive" way of raising the lower bound may require too much memory. In Sections 4.2 and 5 we introduce a technique to avoid the problem: clustering solutions in cubes and branching by clusters.

To motivate the theory that will be developed, we show by an example how to raise the lower bound incrementally. Consider the following matrix A_N that cannot be reduced by dominance.

	0	1	2	3	4	5	6	7	8	9
0	1	1	.	.	.	.	.	.	.	.
1	.	.	1	1	.	.	.	.	.	.
2	.	.	.	.	1	1	.	.	.	.
3	.	.	.	.	.	.	1	1	.	.
4	1	.	.	.	.	.	.	.	1	.
5	.	1	.	.	.	.	.	.	1	.
6	.	.	1	.	.	.	.	.	.	1
7	.	.	.	1	.	.	.	.	.	1
8	.	.	.	.	.	1	1	.	.	.
9	.	.	.	.	1	.	.	1	.	.

Suppose that A_N is the submatrix corresponding to the node N of a column branching search tree, such that the cost of the best solution found is 7 and the partial solution Sol contains 1 column.

An $MSIR$ is made by the 4 rows A_0, A_1, A_2 and A_3. Since $cost(best) - cost(Sol) - |MSIR| = 2$, potentially the lower bound could be raised by 2. The set of irredundant solutions of $UCP(MSIR)$ is equal to $C_0 = \{0, 1\} \times \{2, 3\} \times \{4, 5\} \times \{6, 7\}$. Select row A_4 from the rest of the matrix. The solutions of the matrix made by the rows A_0, A_1, A_2, A_3 and A_4 are represented by the union of the following two sets: $C_1 = \{0\} \times \{2, 3\} \times \{4, 5\} \times \{6, 7\}$ and $C_2 = \{1\} \times \{2, 3\} \times \{4, 5\} \times \{6, 7\} \times \{8\}$.

Consider independently the sets C_1 and C_2 starting with the solutions from C_1. Select row A_5 that is not covered by any solution in C_1. So each solution from C_1 must be augmented by a column covering A_5, which transforms C_1 into $C_1^{'} = \{0\} \times \{2, 3\} \times \{4, 5\} \times \{6, 7\} \times \{1, 8\}$. Note that all solutions in $C_1^{'}$ consist of 5 columns. Now select row A_6. The solutions that have 5 columns, cover A_6, and can be obtained from the solutions in $C_1^{'}$ form the set $C_1^{''} = \{0\} \times \{2\} \times \{4, 5\} \times \{6, 7\} \times \{1, 8\}$. But no solution in $C_1^{''}$ covers row A_7 and therefore each solution from the set must be augmented by a column to cover also A_7. It means that no solution from C_1 can be "extended" to a set of ≤ 5 columns covering all rows from A_N.

Consider solutions from C_2. If we select row A_6, then all the solutions that have 5 columns, cover A_6 and can be obtained from the solutions in C_2 form the set $C_2^{'} = \{1\} \times \{2\} \times \{4, 5\} \times \{6, 7\} \times \{8\}$. But no solution in $C_2^{'}$ covers row A_7 and therefore each solution from the set must be augmented by a column to cover also A_7. It means that no solution from C_2 can be "extended" to a set of ≤ 5 columns covering all rows from A_N. So the lower bound is raised to 6 and the branch corresponding to A_N can be pruned.

4 Representation and Recomputation of the Solutions

In order to present the algorithm for raising the lower bound we must describe how the set of solutions of a matrix is represented and updated.

4.1 Recomputation of the Solutions

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

$Rec(A' + A_p, S)$ 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.

Theorem 4.1 *For any irredundant solution $S^* \in UCP(A' + A_p)$ there is an irredundant solution $S \in UCP(A')$ such that S^* is an element of $Rec(A' + A_p, S)$.*

The proof of the theorem is omitted for lack of space. There are examples showing that $Rec(A' + A_p, S)$ may contain also redundant solutions.

Corollary 4.1 *Let Sol be a set containing all irredundant solutions of $UCP(A')$. Let $Sol^* = \bigcup_{S \in Sol} Rec(A' + Ap, S)$, then Sol^* contains every irredundant solution $S^* \in UCP(A' + Ap)$.*

Proof. It is a direct consequence of Theorem 4.1. □

4.2 Cubes of Solutions

In line of principle, given the operator Rec, one could add one row at a time to A' and build the set of irredundant solutions of $UCP(A)$ from the set of irredundant solutions of $UCP(A')$. This "naive" approach must be discarded because of two disadvantages:

1. The size of the set of irredundant solutions may grow exponentially in the number of added rows.
2. Suppose that we want to raise the lower bound of $MSIR$ by small n and that S is a solution of $UCP(MSIR)$. It may happen that in order to raise S by n we need to add only a small set of rows from $Row(A) \setminus Row(MSIR)$. Denote the set $R(S)$. Let S' be another solution of $UCP(MSIR)$ and suppose that to raise it by n we need to add a small set of rows $R(S')$. The problem is that $R(S)$ and $R(S')$ are usually different. In other words, when we add rows to $MSIR$ we want to add a minimal number of rows which raise all solutions of $MSIR$ by n. But, since these small sets $R(S)$ are usually different for different solutions S from $UCP(MSIR)$, we actually need to add almost all rows.

To solve the previous issues we propose to clusterize solutions that can be raised by the same rows from $Row(A) \setminus Row(MSIR)$. This is achieved by the introduction of *cubes of solutions*, a data structure 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 clusterization of the recomputed solutions. This will support later the design of a raising algorithm based on branching in clusters of solutions, each cluster being one of the recomputed cubes of solutions.

Note however that cubes should not be considered as the only convenient way to clusterize solutions. We believe that studying clusterizations based on different data structures, e.g., binary decision diagrams, will yield interesting results.

As anticipated, we represent the solutions of $UCP(A)$ by sets with a structure of multi-valued cubes [8]. 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. So cube C denotes a set of sets consisting of d columns. In contrast to standard cubes used for the representation of multi-valued functions here cubes 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.

Let A' be a $MSIR$ of A. The set of all irredundant solutions (which are at the same time minimum) 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 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 $UCP(A')$. From the definition of the Rec operator it follows that

$$Rec(A' + A_p, C) = part1(C) \cup part2(C) \times O(A_p) \quad (1)$$

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.

There are three cases:

1. If $D_i \subseteq O(A_p)$ for some i, $1 \leq i \leq d$, then any solution from C covers the row A_p and so $Rec(A' + A_p, C) = C$.
2. If $O(A_p) \cap D_i = \emptyset$ for any $i, 1 \leq i \leq d$, 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. If 1. and 2. are not true, i.e., no D_i is a subset of $O(A_p)$ and $O(A_p)$ intersects at least one domain (without loss of generality, we assume that A_p intersects the first r domains, i.e., $D_1, \cdots, D_r$), then cube C can be partitioned into the following $r + 1$ pairwise not intersecting cubes:

$$\begin{aligned}
C_1 &= D_1 \cap O(A_p) \times D_2 \times \cdots \times D_d \\
C_2 &= D_1 \setminus O(A_p) \times D_2 \cap O(A_p) \times D_3 \times \cdots \times D_d \\
C_3 &= D_1 \setminus O(A_p) \times D_2 \setminus O(A_p) \times D_3 \cap O(A_p) \times \\
&\quad \times D_4 \times \cdots \times D_d \\
&\cdots \\
C_r &= D_1 \setminus O(A_p) \times \cdots \times D_{r-1} \setminus O(A_p) \times \\
&\quad \times D_r \cap O(A_p) \times D_{r+1} \times \cdots \times D_d \\
C_{r+1} &= D_1 \setminus O(A_p) \times \cdots \times D_{r-1} \setminus O(A_p) \times D_r \\
&\quad \setminus O(A_p) \times D_{r+1} \times \cdots \times D_d
\end{aligned} \quad (2)$$

It is not hard to check that the union $C_1 \cup \cdots \cup C_{r+1}$ gives the cube C and that for any pair $C_i, C_j, i \neq j, C_i \cap C_j = \emptyset$. 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

$$part1(C) = C_1 \cup \cdots \cup C_r, \quad part2(C) = C_{r+1}. \quad (3)$$

Equations 1–3 realize the Rec operator as defined in Section 4.1 and characterized by Theorem 4.1. Notice that here we force the Rec operator to generate non-intersecting cubes of solutions; this is not a consequence of the definition of Rec, but is an additional requirement introduced now to avoid considering the same partial solution in more than one branch.

We mentioned that in the computation of Rec some redundant solutions may be introduced. The following revised definition of Rec avoids the generation of obviously redundant solutions obtained from the application of formula (1). Namely, 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 $part1(C)$ is redundant since it contains more columns than S''.

Theorem 4.2 *If the computation of the* Rec *operator is modified as follows:*

$$Rec(A' + A_p, C) = part1(C) \cup part2(C) \times \times [O(A_p) \setminus (D_1 \cup \cdots \cup D_d)] \quad (4)$$

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

The proof of the theorem is omitted for lack of space.

5 The Raising Procedure

5.1 Overview of the Raising Algorithm

As anticipated in Section 2, we propose an n-*raiser* procedure which is called in the column branching mode as described in Fig. 1. Let A be the covering matrix corresponding to the node where n-raiser is invoked and A' be an $MSIR$ of A. We start with the set of irredundant solutions of $UCP(A')$, 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 $MSIR$. Then choose a "good" row of A from those not in A', say row A_p. According to Equations (1–5), $Rec(MSIR(A) + A_p, C)$ 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 Equations (1–5).

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 $A - MSIR(A)$ (i.e., matrix A without the rows of $MSIR(A)$). 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. So the first matrix gives a "lower bound submatrix" for the node.

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 submatrix of the pair.

The flow of *n-raiser* is shown in Fig. 2. 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 and any solution from $Cube$ consists of fewer columns than the current best one, since *n-raiser* is invoked if $|MSIR| + n + cost(Sol) = cost(best)$ where Sol is the partial solution found in the column branching mode before invoking the raiser. Then a solution from $Cube$ is selected as the current best and the range of raiser is reduced to $n - (cost(old\_best) - cost(best))$ since the gap between the current best solution and $MSIR$ is reduced.

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 $|S| \leq |MSIR| + n$.

If neither pruning condition holds the procedure *raise_or_trim* is invoked to address the following two cases, which let us modify the cube of solutions without branching:

1. If a row A_p exists such that no solution from $Cube$ covers A_p, then there is no splitting of the cube, since Rec yields only one cube $C \times O(A_p)$. Row A_p is removed from A'' and added to A'.

2. If there exists a row A_p intersecting only one domain D_i of $Cube$ and the number of domains in $Cube$ is equal to $|MSIR| + n - 1$ then only two cubes are generated after splitting. The first cube is from $part1(Cube)$ and is obtained by reducing domain D_i to $D_i \cap O(A_p)$. The second cube has one more domain and can be discarded since the total number of domains in the cube is $|MSIR| + n$. Row A_p is removed from A'' and added to A'.

The previous conditions are checked in $raise\_or\_trim$ by iterating through the rows of A'' until both conditions are false for any row from A''.

After all these special cases have been addressed, a new row A_p is selected by $select\_row$. The row A_p is removed from A and drives the splitting of $Cube$. The strategy to select the best row in order to split the current $Cube$, before calling recursively $raiser$, looks for the row of A which intersects the minimum number of domains of $Cube$. The reason is to reduce the number of branches from the node, i.e., the number of domains intersecting the row to be added plus 1. 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 $Cube$ 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 $Cube$ that do not cover A_p, which we want to maximize when selecting a new row. If $D'_{i_k} = \emptyset$, for some k, this means that A_p is covered by any solution from $Cube$. Such a row is simply removed from A'' and added to A'.

5.2 Correctness of n-raiser

The correctness of the n-raiser procedure, applied to matrix A with lower bound $|MSIR(A)|$, can be argued using the notions of *subsolution* or *partial solution* and of *complete set of solutions*, introduced as follows.

A set S' of columns of A is a **subsolution** or **partial solution** of $UCP(A)$ if it is a solution of a subproblem A', but is not a solution of $UCP(A)$.

Let C be the cube of subsolutions corresponding to $MSIR(A)$, then C has the property that for any solution S of $UCP(A)$ there is a subsolution from C which is contained in S. Indeed, since S covers all the rows of A, including those contained in $MSIR(A)$, then S contains $|MSIR(A)|$ columns covering the submatrix $MSIR(A)$ that form a subsolution from C. A set of subsolutions is **complete** if for any solution S of $UCP(A)$ there is a subsolution from the set which is contained in S. So the set of subsolutions contained in the cube C is complete.

Let S' be a solution of subproblem $UCP(A')$. Denote by $Gen(S')$ the set of irredundant solutions of $UCP(A)$ that contain S'. Similarly, if C is a set of partial solutions, denote by $Gen(C)$ the set of irredundant solutions of $UCP(A)$, each of which contains a solution from C.

Lemma 5.1 *Let* S' *be a solution of* $UCP(A')$ *and* A_p *be a row from* $Row(A) \setminus Row(A')$*. Then* $Gen(S') \subseteq Gen(Rec(A' + A_p, S'))$ *where* Rec *is the recalculation operation defined in Section 4.1.*

Proof. Let S be a solution of $UCP(A)$ containing S', i.e., $S \in Gen(S')$. If S' covers row A_p then $Rec(A' + A_p, S')$ is equal to $\{S'\}$ and so $Gen(Rec(A' + A_p, S'))$ contains S. If S' does not cover A_p, then $Rec(A' + A_p, S')$ contains every solution $S' \cup \{j\}$, $j \in O(A_p)$. Moreover, S contains S' and, since it covers A_p, it obviously contains a column $j \in O(A_p)$. So again

```
/* n-raiser returns an empty solution if lower bound of
UCP(A) can be raised to |MSIR| + n. If not, it returns
a current minimum solution of UCP(A) */

raiser(A', A'', Cube, n) {
    /* A' = MSIR, A'' = A - A', Cube = solutions of
    UCP(MSIR) */
    /* Cost of solutions from Cube exceeds lower bound by n */
    if (number_of_domains(Cube) > |MSIR| + n)
        return(∅)
    /* No rows to add ? */
    if (A'' = ∅) {
        /* Extract new best solution */
        best = extract(Cube)
        /* Recalculate range of raiser */
        n' = n - (cost(Old_best) - cost(best))
        n = n'
        return(best)
    }

    /* Process rows that do not split Cube */
    raise_or_trim(A', A'', Cube)
    /* Select a row to add */
    A_j = select_row(Cube, A'')
    A' = A' + A_j; A'' = A' - A_j
    /* split Cube. Note that Cube_{r+1} has one more domain */
    Cube = Cube_1 ∪ ··· ∪ Cube_r ∪ Cube_{r+1}
    /* Call raiser recursively */
    for (i = 1; i ≤ (r + 1); i++) {
        New_sol = raiser(A', A'', Cube_i, n)
        if (cost(New_sol) < cost(best))
            best = New_sol
    }
    return(best)
}
```

Figure 2: *n-raiser* algorithm

$Gen(Rec(A' + A_p, S'))$ contains S. □

From Lemma 5.1 it follows that the Rec operation preserves the completness of a set of subsolutions.

Theorem 5.1 *The n-raiser procedure finds correctly a larger lower bound or a smaller upper bound.*

Proof. n-raiser starts with the set of solutions of $UCP(MSIR)$, which is a complete set of partial solutions of $UCP(A)$. Since the Rec operation preserves completness, the set of all "boundary" cubes, i.e., cubes corresponding to either leaf nodes of the search tree or to the nodes not yet split, is a complete set of partial solutions. When we apply an n-raiser to A we actually try to find a complete set of partial solutions 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.

Suppose that there is no complete set of partial solutions consisting of at least $|MSIR(A)|+n$ columns. It means that n-raiser finds a leaf node with a cube containing solutions of $|MSIR(A)| + n'$ columns where $n' < n$. In that case we update the n-raiser into an n'-raiser and continue the search. If the n'-raiser succeeds we return a solution of $|MSIR(A)| + n'$ columns which is minimal.

If the n'-raiser fails then there is a solution of $UCP(A)$ consisting of $|MSIR(A)| + n''$ columns, where $n'' < n'$. Then we update the n'-raiser into an n''-raiser and continue the search. □

6 Experimental Results

We have implemented a program AURA to solve UCP and we have compared it with the routine *mincov* available in ESPRESSO, with MINCOV_LLB, that is our implementation of some features of SCHERZO and with the results of the real SCHERZO implemented by O. Coudert. The program SCHERZO is the most effective solver of UCP currently reported. Its main features have been described in the literature [3, 2, 1]; they include a better heuristic selection of the $MSIR$, logarithmic lower bound, left hand side lower bound, limit lower bound, and partition-based pruning. Of these features we have implemented in MINCOV_LLB, to the best of our understanding of the original description, the following two: better heuristic selection of the $MSIR$ and limit lower bound. The limit lower bound is a major novelty of SCHERZO, which accounts for strong savings in the number of nodes of the computation tree compared to the original *mincov* of ESPRESSO.

The benchmarks belong to three different classes: in Table 1 there are difficult cases from the collection of ESPRESSO (we start from the matrix obtained by ESPRESSO after removing the essential primes), in Table 2 there are random generated matrices with varying row/column ratios and densities, in Table 3 there are matrices encoding constraints satisfaction problems from [9]. The experiments have been performed with a 2GB 300Mhz Alpha with timeout set to 3 days of cputime.

The tables report two types of data for comparison: the number of nodes of the column branching computation tree and the running time. About the number of nodes we clarify that

1. AURA 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 follows a dual strategy, i.e., it builds the column branching computation tree, but when at a node the difference between the upper bound and the lower bound is less or equal to the raising parameter r (or $maxRaiser$), AURA 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.

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

3. The raising parameter is an input to AURA. Currently we have experimented with some values and we report in the tables the value used in a specific run. The higher is the raising parameter, the fewer column branching nodes compared to cube branching nodes there will be. With a value high enough, there will be a single column node and the rest will be all row nodes.

We compared also with the real SCHERZO, whose author was kind enough to run for us the examples. There is a large gap in many cases between the results of SCHERZO and those of MINCOV_LLB, which is our implementation of a subset of SCHERZO, A major reason may be that our reimplementation of the better heuristic selection of the $MSIR$, even though it follows the hint given by Coudert, in practice does not mimic well enough the one in SCHERZO; moreover, as already said, SCHERZO features additional improvements that we did not implement. It is important to underline that:

1. both AURA and MINCOV_LLB exploit the same re-implementation of Coudert's better heuristic selection of the $MSIR$;

2. AURA could be improved noticeably by reproducing more successfully the better heuristic selection of the $MSIR$ or any other feature of SCHERZO. In other words, AURA demonstrates a dual search technique, which may benefit from other improvements to standard branch and bound.

3. in overall SCHERZO has been implemented more efficiently, as magnified also by the circumstance that it is comparatively faster on a slower machine.

The experiments show that AURA outperforms ESPRESSO and MINCOV_LLB. It is always faster and in the most difficult examples either it has a running time advantage up to two orders of magnitude or the other programs fail due to timeout (3 days) or spaceout (2G). Instead SCHERZO is a very tough competitor, which is faster on the examples from Table 1, but has a less effective pruning strategy in those of Tables 2 and 3, partially compensated by a better $MSIR$. The example *saucier.t* is an extreme case where the virtues of AURA prevail. Recently O. Coudert kindly provided us with a copy of SCHERZO, to let us analyze in depth the comparative features of the two programs. We will report on the study as soon as done. We expect to transfer to AURA the better computation of the $MSIR$ apparently implemented in SCHERZO.

We do not have a systematic comparison with the results by BCU, a recent ILP-based covering solver [5]. 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 [5] 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 completed the example in less than 3 minutes.

7 Conclusions

We have introduced a new technique to solve exactly a discrete optimization problem, based on the paradigm of "negative" thinking. The motivation is that when searching the space of solutions often a good solution is reached quickly and then it is improved only a few times before the optimum is found; so most of the solution space is explored to certify optimality, but it does not yield any improvement in the cost function. This suggests that more powerful lower bounding would speed up the search dramatically, as shown by the introduction of the limit lower bound [3]. Our approach is more radical because when we are dealing with a subspace of solutions unlikely to improve the upper bound, we switch the search strategy to a different one geared to raise the lower bound. To design a search strategy which realizes negative thinking we introduced *cubes of solutions*, a data structure inspired by multi-valued cubes. Applying the operator Rec to a cube of solutions one obtains a collection of cubes of solution, thereby providing a natural clustering of the recomputed solutions. As argued in the paper, clustering is required to design a recursive algorithm based on branching in subsets of solutions and allows the lower bound to be raised independently starting from different subsets of solutions.

For illustration we applied our technique to the unate covering problem, usually solved exactly by a branch-and-bound procedure, where one lower bounds by means of an independent set of rows, and branches on columns. We have 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 $maxRaiser$, that we are free to set. 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 raising yields a lower bound that matches the current upper bound and so we are done, or we produce a better solution. $Raiser$ defines a computation tree whose nodes have associated a lower bound submatrix and a cube of solutions. 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; operationally $raiser$ calls itself recursively passing as parameters each such disjoint cube of solutions and the augmented lower bound submatrix. It would be interesting to explore a mixed approach where one accumulates some cubes of solutions at the same node and fewer recursive calls are made, trading off time vs. memory.

The reported experiments show that our program AURA, outperforms ESPRESSO and MINCOV_LLB, which is the algorithm in ESPRESSO enhanced by our implementation of Coudert's limit lower bound. The package SCHERZO is faster than AURA on the examples from Table 1, but it has a less effective pruning strategy in those of Tables 2 and 3, partially compensated by a better $MSIR$.

Future work includes a more careful study of some algorithmic design issues, like the selection of the next row, trading-off number of nodes vs. number of cubes stored in a node, and setting automatically and adaptively the raiser parameter.

A more basic line of research is the exploration of data structures different from cubes since the latter are just the simplest way of representing sets of partial solutions. We believe that studying various ways of representing implicitly sets of solutions is a promising direction of investigation to rescue branch-and-bound from its current limits. Another important direction of future research is to apply the negative thinking approach to other problems.

References

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

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

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

[4] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli. *Synthesis of FSMs: functional optimization*. Kluwer Academic Publishers, 1996.

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

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

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

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

[9] Tiziano Villa. *Encoding Problems in Logic Synthesis*. PhD thesis, University of California, Berkeley, Electronics Research Laboratory, May 1995. Memorandum No. UCB/ERL M95/41.

matrix	R × C (Sparsity)	Sol.	ESPRESSO		SCHERZO		MINCOV_LLB		AURA		
			nodes	time	nodes	time	nodes	time	nodes/A-nodes	time	r
exps	680 × 696 (1.2%)	76	13	0.0	na	na	13	0.0	13/0	0.0	3
fout	177 × 431 (2.4%)	38	161	1.3	na	na	49	0.7	18/44	0.2	2
max512	559 × 515 (1.3%)	113	111	1.4	na	na	25	0.4	19/25	0.4	3
addm4	832 × 1073 (0.6%)	165	121	3.6	na	na	29	1.1	17/11	0.6	2
mlp4	530 × 594 (0.99%)	109	2122	22.6	24	0.1	153	4.3	34/206	1.3	3
pdc	6904 × 19021 (0.34%)	94	195	62.7	44	6.1	88	58	41/132	52.9	3
lin.rom	1030 × 1076 (0.9%)	120	370	29.1	238	4.7	106	10.1	61/240	7.7	3
ex5	831 × 2428 (2%)	37	-	time	616091	2450.5	597644	214300	155/169245	1315.2	4
prom2	1924 × 2611 (0.31%)	278	-	time	25993	5149.2	-	time	1478/1097624	24071.4	3
max1024	1090 × 1264 (0.52%)	245	-	time	531618	9583.6	-	time	12402/3850628	36240	3

Table 1: Results from *Espresso Benchmarks*

matrix	R × C (Sparsity)	Sol.	ESPRESSO		SCHERZO		MINCOV_LLB		AURA		
			nodes	time	nodes	time	nodes	time	nodes/A-nodes	time	r
tc.90	50 × 100 (90%)	2	135	2.6	2	0.0	3	0.3	3/1	0.1	3
tc.70	50 × 100 (70%)	2	135	3.5	2	0.0	3	0.2	3/1	0.1	3
tc.50	50 × 100 (50%)	3	2569	13.9	107	0.6	107	2.3	5/32	0.1	3
tc.30	50 × 100 (30%)	4	12047	37.8	65	0.3	1061	7.1	11/203	0.2	3
tc.10	50 × 99 (10%)	8	843	3.3	90	0.1	131	0.7	17/166	0.1	3
tr.10	100 × 50 (20%)	8	12466	59.6	2077	4.1	2232	21.1	94/2529	2.9	3
tr.20	100 × 50 (40%)	5	16905	49	1823	3.9	2193	19.2	31/951	1.7	3
tr.30	100 × 50 (60%)	3	947	9.5	63	0.9	61	3.4	5/26	0.3	3
tr.40	100 × 50 (80%)	2	73	4.3	2	0.0	3	0.6	3/1	0.3	3
ts.90	100 × 100 (90%)	2	175	21.2	2	0.0	3	2.6	3/1	1	3
ts.70	100 × 100 (70%)	3	5083	47.0	167	5.3	163	15.8	5/112	0.7	3
ts.50	100 × 100 (50%)	4	66147	316.4	4011	20.2	3137	67.3	7/1030	1.6	3
ts.30	100 × 100 (30%)	5	116307	792.8	1752	8.5	8997	139.6	35/1108	2.5	3
ts.10	100 × 100 (10%)	12	-	time	95573	187.3	175255	1255.1	5043/201091	129.3	3

Table 2: Results from *Random Generated Matrices*

matrix	R × C (Sparsity)	Sol.	ESPRESSO		SCHERZO		MINCOV_LLB		AURA		
			nodes	time	nodes	time	nodes	time	nodes/A-nodes	time	r
bbara.t	45 × 26 (41%)	7	61	0.02	0	0.0	7	0	7/2	0	3
dk512x.t	91 × 59 (45%)	6	213	0.24	55	0.0	57	0.15	9/24	0.04	3
ex4inp.t	91 × 240 (46%)	5	5279	16.81	17	0.3	19	0.66	9/14	0.27	3
ex5inp.t	36 × 34 (48%)	4	64	0.05	4	0.0	6	0.01	6/2	0.01	3
ex6inp.t	28 × 96 (48%)	4	639	0.54	35	0.0	103	0.28	7/23	0.03	3
maincont.t	105 × 67 (35%)	7	504	0.69	68	0.0	101	0.4	11/12	0.06	3
opus.t	45 × 63 (45%)	5	121	0.1	7	0.0	5	0.01	5/2	0.01	3
ricks.t	78 × 363 (47%)	5	20	0.37	10	0.2	12	0.36	8/43	0.33	3
saucier.t	171 × 6207 (47%)	6	-	mem	186927	5441.0	-	mem	10/76	222.47	3

Table 3: Results from *Encoding Problem Matrices*