NEGATIVE THINKING IN SEARCH PROBLEMS

by

Luca P. Carloni

Memorandum No. UCB/ERL M97/89

12 December 1997

NEGATIVE THINKING IN SEARCH PROBLEMS

by

Luca P. Carloni

Memorandum No. UCB/ERL M97/89

12 December 1997

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Negative Thinking in Search Problems

Luca P. Carloni

Abstract

We introduce a new technique to solve exactly a discrete optimization problem, based on the paradigm of "negative" thinking. 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 in the cost function. So it is quite natural for an algorithm to be "skeptical" about the chance to improve the current best solution. This suggests that more powerful lower bounding would speed up the search dramatically, as shown by the good results obtained by Olivier Coudert with its "limit lower bound" technique [1]. Our approach is more radical than Coudert's because, when we deal with a subspace of solutions, if appropriate, we switch the search strategy to a different one based on negative thinking by incremental problem solving.

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.

We developed a program, AURA, based on this paradigm. Experiments show that AURA outperforms both ESPRESSO and our enhancement of ESPRESSO using Coudert's limit lower bound. 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 developed by Olivier Coudert 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.

Acknowledgements

I would like to thank my advisor Professor Alberto L. Sangiovanni-Vincentelli for its encouragement and support during these first years at Berkeley. Alberto has always given me the time and freedom to choose my research area and I am excited by the perspective of working with him to achieve my Ph.D. degree.

I am very grateful to Professor Robert K. Brayton, who has been a constant point of reference for my research, to Tiziano Villa, my "maestro" at Berkeley, and to Evguenii I. Goldberg, who came from Belarus to share his many bright ideas with us.

The present work, born from an idea of Evguenii, is a joint effort with all these people.

Contents

List of Figures

List of Tables

Most of us may not believe in the story of a Devil to whom one can sell one's soul, but those who must know something about the soul (considering that as clergymen, historians, and artists they draw a good income from it) all testify that the soul has been destroyed by mathematics and that mathematics is the source of an evil intelligence that while making man the lord of the earth has also made him the slave of his machines.

R. Musil.

Chapter 1

Positive Thinking and Negative Thinking

1.1 Branch-and-Bound and the Unate Covering Problem

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, we consider BAB applied to the solution of the Unate Covering Problem (UCP), that is of great interest in logic synthesis and operations research. UCP can be stated as follows.

Definition 1.1.1 **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 $x^T c = \sum_{j=1}^{n} x_j c_j$, over all $x \in \{0, 1\}^n$,*

Subject to:

$$A\,x \geq (1, 1, \cdots, 1)^T. \tag{1.1}$$

The constraint $A\,x \geq (1, 1, \cdots, 1)^T$, ensures that the nonzero elements of x determine a column set $S = \{j \mid x_j = 1\}$, which "covers" all rows of A, that is,

$$\forall i\,,\ \ \exists j \in S \ \text{ such that } \ A_{i,j} = 1.$$

Thus the minimum unate covering problem is to find a column set of minimum cost, which satisfies the constraint Equation 1.1. We shall discuss mainly the special case of this problem for which $c_j = 1$, $\forall j$. Exceptions to this assumption will be specifically noted in the sequel. We will denote an instance of UCP with matrix A as $\mathbf{UCP(A)}$.

A complete survey of the covering problem from the perspective of the logic synthesis community can be found in the fifth chapter of the book *"Synthesis of Finite State Machines: Functional Optimization"* by T. Kam *et. al.* [2]. An exact solution of the covering problem is obtained by a branch-and-bound recursive algorithm, which has been implemented in successful computer programs [3, 4]. 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 the A by removing some rows and columns. On the path some columns are included in the partial solution; we denote by $\mathbf{path(A_N)}$ 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 a 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.

1.2 Incremental Problem Solving

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 we are solving 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 P incrementally, we start with a subproblem P' of P, such that the solutions of P' can be represented compactly. Then we modify gradually P' by making it more complex to come closer to the full problem P 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 P' to P with the goal to prove that no solution of P' can overcome the obstacles and be extended to a solution of P.

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 than $ubound$, $Sol(Pi) = \emptyset$, 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 than $ubound$ 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 work we present an incremental UCP solver called *raiser*. Although we demonstrate our technique on UCP it can be applied to any discrete optimization problem 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 ideas discussed in this dissertation were presented for the first time at the *International Conference on Computer-Aided Design (ICCAD)* on November 1997 [5].

The dissertation is organized as follows. In Chapter 2 we first review briefly how UCP is solved traditionally by branch-and-bound and then we show how an incremental solver is incorporated into the standard branch-and-bound procedure for UCP. Chapter 3 describes how the solutions of UCP are represented and recomputed. The raising procedure is explained in detail in Chapter 4 and its relation to previously known lower bounding techniques is explored in Chapter 5. Experimental results are discussed in Chapter 6. Applications of incremental problem solving to other optimization and decision problems are outlined in Chapter 7. Conclusions are given in Chapter 8.

Chapter 2

Incremental Problem Solving

2.1 A Branch-and-Bound Algorithm for Minimum Cost Unate Covering

In this section we present with more detail the branch-and-bound recursive algorithm *mincov* to solve exactly UCP. The inputs of the *mincov* algorithm as outlined in Fig. 2.1 are:

- a covering matrix A;
- a partial solution of the current path, denoted *path* (initially empty);
- a row of non-negative integers *weight*, 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*. Instead an empty solution is returned if a solution cannot be found which beats *ubound* [1]. Infeasibility means that no satisfying assignment of the product of clauses exists. When *mincov* is called on A with an empty partial solution *path* and initial *lbound* and *ubound*, it returns a best global solution.

[1]in the case of an instance of BCP (see Section 7.1.1) an empty dolution is returned if a solution cannot be found which beats *ubound* or an infeasibility is detected.

The algorithm calls first a procedure *reduce* that applies to A essential column detection and dominance reductions. These reduction operations delete from A some rows, columns and entries. What is left after reduction is called a cyclic core. The final goal is to get an empty cyclic core. The value of the lower bound is updated using a maximal independent set computation. If no bounding is possible and the reductions do not suffice to solve completely the problem, a partition of the reduced problem into disjoint subproblems is attempted and each of them is solved recursively. When everything fails, binary recursion is performed by choosing a branching column. Solutions to the subproblems obtained by including the chosen column in the covering set or by excluding it from the covering set are computed recursively and the best solution is kept (the second recursion is skipped if the solution to the first one matches the updated lower bound).
The procedure *mincov* returns when:

- The cost of a partial solution, found by adding essential columns to *select*, is more than *ubound* or infeasibility is detected when applying the domination rules (line 1). An empty solution is returned.

- The best current solution is found by applying Gimpel's reduction technique (line 2). Since *gimpel_reduce* calls recursively *mincov*, an empty solution could be returned too.

- The updated lower bound, determined by adding to *lbound* the cost of the essential primes and of the maximal independent set, is not less than *ubound* (line 5). An empty solution is returned.

- The previous case does not hold and there is no cyclic core. The best current solution is found by updating *select* with the new essential and unacceptable columns (line 6).

- The best current solution is found by partitioning the problem (line 7). The procedure *mincov* is called recursively on two smaller covering matrices determined by *block_partition* (line 8 and 10). An empty solution can be returned by either recursive call. If the first call to *mincov* returns an empty solution, the second one is not invoked (line 9). If neither call returns an empty solution, each contributes its returned value to the current solution.

- A branching column is chosen and *mincov* is called recursively with the branching column in the covering set (line 12). If the recursive call of *mincov* returns a non-empty solution that matches the current lower bound (*lbound_new*), that solution is returned as the best current

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

Figure 2.1 A branch-and-bound algorithm for covering problems.

solution (line 14). If the cost of the best current solution is less than *ubound*, *ubound* is updated, i.e., the best current solution is also the best global solution (line 13).

- As in the previous case, except that *mincov* is called recursively with the branching column not in the covering set (line 15). The best among the solution found in the previous case and the one computed here is the best current solution.

Notice the following facts about the procedure *mincov*:

- The parameter *lbound* is updated once (line 4). The reason is that after the computation of the essential columns (line 1) and of the independent set (line 3), the cost of the previous partial solution summed to the cost of the essential columns and of the independent set is potentially a sharper lower bound on any complete solution obtained from this node of the recursion tree. The updated value *lbound_new* is used in the rest of the routine. The lower bound is a monotonically increasing quantity along each path of the computation tree.

- The parameter *ubound* is updated once (line 13). At that point a new complete solution has just been returned by the recursive call to *mincov* (line 12) and an updated value of *ubound* must be recomputed for the following recursive call of *mincov* (line 15). The reason is that when a new complete solution is obtained, the current *ubound* is not any more valid and therefore it must be updated before it is used again. To be updated, *ubound* is compared against the cost of the newly found solution, and the minimum of the two is the new *ubound*. The upper bound is a monotonically decreasing quantity throughout the entire computation.

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

2.2 Incorporating an Incremental Solver into Branch-and-Bound

The flow of a UCP solver based on branch-and-bound is shown in Fig. 2.2. The parts of text in bold font refer to the incremental solver and will be explained below. For details the reader is referred to [2]. 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 $MSIR$).

Procedure *raiser*, performing "negative thinking", is invoked with a parameter n when $MSIR$ is a lower bound not sufficient to prune the subtree rooted at the current node, but increasing

```
$branch\_and\_bound(A, Sol, n)$ {
    /* $A$ = matrix of UCP, $Sol$ = current (partial) solution */
    /* n = "range" of raiser, best = best current solution */
    if ($A = \emptyset$)
        return($Sol$) /* new best solution */
    /* Column and row dominance */
    $simplify(A)$
    /* Lower bound evaluation */
    $MSIR = find\_msir(A)$
    if (($lower\_bound(A) + cost(Sol)) \geq cost(best)$)
        return($\emptyset$)
    /* Is the current node within the range of raiser? */
    if ($(|MSIR| + cost(Sol) + n) \geq cost(best)$) {
    /* $n'$ exact amount to raise */
        $n' = cost(best) - (|MSIR| + cost(Sol))$
        return($raiser(MSIR, n', A)$)
    }
    /* select a branching column */
    $j = select\_column(A)$
    /* Decomposition: $A_1(A_2)$ for including (not including) $j$ in solution */
    $Sol_1 = Sol \cup \{j\}$
    $Sol_2 = Sol$
    for ($i = 1; i \leq 2; i++$)
        {$New = branch\_and\_bound(A_i, Sol_i, n)$
        if ($cost(New) < cost(best)$) {
            $best = New$
            if ($cost(best) \leq (cost(Sol) + |MSIR|)$)
                return($best$)
        }
    }
    return($best$)
}
```

Figure 2.2 Branch-and-Bound enhanced by incremental solver

the lower bound by n would allow such pruning. *Raiser* starts from the subproblem $UCP(MSIR)$ whose solution space is very regular and then tries to extend it gradually to A; *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 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 [6] a new technique for pruning the search tree called limit lower bound was reported. Sometimes this technique allows one 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 the procedure *raiser* invocated with $n = 1$.

The idea of incremental improvement of the lower bound is discussed in the following Section, while Chapter 3 describes how the solutions of UCP are represented and recomputed and Chapter 4 gives a detailed description of the raising algorithm.

2.3 Incremental Improvement of the Lower Bound

Given an optimization problem P such that for any subproblem P' the cost of a minimum solution of P is greater than or equal to that of P', the size of a minimum solution of P' gives a **lower bound** on the size of a minimum solution of P. This fact is called **cost monotonicity assumption** and it 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 $min(UCP(A')) \leq min(UCP(A))$. We shall call **lower bound submatrix** a submatrix A' whose minimum solution is used for evaluating a lower bound for $UCP(A)$. A **maximal set of independent (non-intersecting) rows** ($MSIR$) of A is usually chosen as submatrix A', denoted also as $A' = MSIR(A)$. If A' is a $MSIR$ then $min(UCP(A')) = |Row(A')|$ because each row in $MSIR$ is covered by a different column.

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 $\mathbf{Sol(A', m)}$ the set of solutions of $UCP(A')$ which includes all the irredundant solutions consisting of m or fewer 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 $Sol(A' + A_{i_1} + .. + A_{i_k}, r) = \emptyset, r \geq m$ 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 fact 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 3.1 we will discuss how to recalculate solutions. However, this "naive" way of raising the lower bound may require too much memory. In Section 3.3 we will introduce a technique to avoid the problem which is based on clustering the solutions in cubes and branching by clusters. Finally, Section 4.2 contains an example which shows how to raise the lower bound incrementally.

The previous discussion motivates the following modification of the algorithm illustrated in Fig. 2.1. This modification corresponds to the parts of text in bold font in Fig. 2.2 and is based on the new procedure *raiser*, which is invoked with an integer parameter n. When a node N is reached, compute an $MSIR$ for the matrix A_N corresponding to the node. If

$$|MSIR| + |path(A_N)| + n \geq |Best|$$

where $Best$ is the best current solution, then procedure *raiser* is applied to $UCP(A_N)$, otherwise branching on columns continues. The outcome of *raiser* may be either that the lower bound $|MSIR|$

can be improved by the quantity

$$n = |Best| - |MSIR| - |path(A_N)|$$

and the recursion in the node stops, or that the lower bound cannot be improved by n to become equal to $|MSIR| + n$. In the latter case 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 [6] it was reported that the limit lower bound allows the pruning of some or many branches of the search tree. The effect of this modification is to reduce the runtimes for some examples 10 times and even more. The limit lower bound prunes no more branches of the search tree than *raiser* with $n = 1$.

The next task is to design an efficient procedure to implement *raiser*. A "naive" implementation where one stores the set of solutions $Sol(A', |MSIR(A)| + n)$, where A' is a lower bound submatrix for $UCP(A)$, may require too much memory. In other words, if the lower bound can be raised to $|MSIR| + n$, eventually $Sol(A', |MSIR(A)| + n)$ will be empty, but 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 the whole set $Sol(A, |MSIR(A)| + n)$, i.e., all irredundant solutions of $UCP(A)$ with $|MSIR| + n$ or fewer columns. In the next chapter we present another way to design *raiser*, so that the previous memory problem is avoided by means of a new scheme of branching on rows.

Chapter 3

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.

3.1 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 $\mathbf{O}(\mathbf{A}_\mathbf{p})$ 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 rule 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 3.1.1 *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* $Rec(A' + A_p, S)$.

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 $S^* \in Rec(A' + A_p, S^*)$.

2. S^* is redundant for $UCP(A')$. First of all, we show that in this case there is only one redundant column and this is a column covering 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 $UCP(A' + A_p)$.

   On the other hand, two (or more) columns cannot 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 Rec, 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 $S^* \in UCP(A' + A_p)$ there is an irredundant solution $S \in UCP(A')$ such that S^* is an element of $Rec(A' + A_p, S)$.

$\square$

Notice that it is possible that $Rec(A' + A_p, S)$ may contain also redundant solutions. Consider the following situation

$$\begin{array}{c|ccccc}
 & c_1 & c_2 & c_3 & c_4 & c_5 \\
\hline\hline
A_p & 0 & 1 & 0 & 1 & 0 \\
\hline
 & 1 & 1 & 0 & 0 & 0 \\
A' & 0 & 0 & 1 & 0 & 0 \\
 & 0 & 0 & 0 & 0 & 1
\end{array}$$

A' has the following two irredundant solutions

$$Sol = c_1, c_3, c_5; \quad c_2, c_3, c_5$$

Then we compute $Rec(A' + A_p, S)$ as

$$Rec(A' + A_p, Sol) = c_1, c_3, c_5, c_2; \quad c_1, c_3, c_5, c_4; \quad c_2, c_3, c_5$$

where the first two solutions come from c_1, c_3, c_5 and the last one from c_2, c_3, c_5. The solution c_1, c_3, c_5, c_2 is redundant.

Corollary 3.1.1 *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 3.1.1. □

3.2 Cubes of Solutions

In 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 3 and that S is a solution of $UCP(MSIR)$. It may happen that in order to raise S by 3 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 3 we need to add a small set of rows $R(S')$. The problem is that $R(S)$ and $R(S')$ are usually different. This implies that when we add rows to $MSIR$ we want to add a minimal number of rows which raise all solutions of $MSIR$ by 3. But, since the 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 group solutions in clusters 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 solution, thereby providing a natural clustering of the recomputed solutions. In Chapter 4 we will use this idea to design a raising algorithm based on branching in cluster 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 cluster solutions. We believe that studying clusters 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 [7]. 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 common 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

$$Rec(A' + A_p, C) \; = \; part1(C) \cup part2(C) \times O(A_p) \tag{3.1}$$

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 may 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:

$$\begin{aligned}
C_1 &= D_1 \cap O(A_p) \times D_2 \times \cdots \times D_d \\
C_2 &= D_1 \setminus O(A_p) \times D_2 \cap O(A_p) \times D_3 \times \cdots \times D_d \\
C_3 &= D_1 \setminus O(A_p) \times D_2 \setminus O(A_p) \times D_3 \cap O(A_p) \times D_4 \times \cdots \times D_d \qquad (3.2) \\
& \ldots \\
C_r &= D_1 \setminus O(A_p) \times \cdots \times D_{r-1} \setminus O(A_p) \times D_r \cap O(A_p) \times D_{r+1} \times \cdots \times D_d \\
C_{r+1} &= D_1 \setminus O(A_p) \times \cdots \times D_{r-1} \setminus O(A_p) \times D_r \setminus O(A_p) \times D_{r+1} \times \cdots \times D_d
\end{aligned}$$

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

$$part1(C) = C_1 \cup \cdots \cup C_r, \;\; part2(C) = C_{r+1}. \tag{3.3}$$

Equations 3.1–3.3 realize the *Rec* operator as defined in Section 3.1 and characterized by Theorem 3.1.1. 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 3.1. 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 3.2.1 *If the computation of the Rec operator is modified as follows:*

$$Rec(A' + A_p, C) \;=\; part1(C) \;\cup\; part2(C) \times [O(A_p) \setminus (D_1 \cup \cdots \cup D_d)] \tag{3.4}$$

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, $r \leq d$.

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

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

Since, by distributivity of the Boolean operators $\cup$ and $\cap$, $D \cap O(A_p) = D''_1 \cup \cdots \cup D''_r$, cube C can be rewritten as follows:

$$C^* \;=\; part2(C) \times (D \cap O(A_p))$$

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

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

Now define the cubes C'_k, $1 \leq k \leq r$, 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 - are constructed so 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 $D'_{k+1} = D_{k+1} \setminus O(A_p), \cdots, D'_r = D_r \setminus O(A_p)$:

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

Consider the k-th component of cube C^*, for $1 \leq k \leq r$,

$$\begin{aligned} C^*_k &= part2(C) \times D''_k, \\ &= D'_1 \times \cdots \times D'_r \times D_{r+1} \times \cdots \times D_n \times D''_k \\ &= D'_1 \times \cdots \times D'_k \cdots \times D'_r \times D_{r+1} \times \cdots \times D_n \times D''_k \end{aligned}$$

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

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

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 $S = S' + j$. So any solution from C^*_k is redundant for $1 \leq k \leq r$. □

3.3 Avoiding Repeated Generation of 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 3.2, the solutions of $A' + A_p$ are found by

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

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 a 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 $O(A_p) \cap D_1$ 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 $O(A_p) \cap D_1$ are not considered, whereas $S \cup S'$ contains such a column, i.e., column j_1.

In summary, 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 generating solutions containing columns from $(D_1 \cup D_2 \cup \cdots \cup D_{k-1}) \cap O(A_p)$. The following lemma guarantees that no irredundant solution is missed by this restriction.

Lemma 3.3.1 *Let C be a cube of solutions of $UPC(A')$ and A_p be a row from $Row(A) \setminus Row(A')$. Let S be a solution of $UCP(A)$ from $Gen(C)$, where $Gen(C)$ denotes all the solutions of $UCP(A)$ which contain a partial solution from C. Suppose w.l.o.g. that A_p intersects the first r domains of C. Then S can be generated in one of the $r+1$ branches corresponding to the cubes C_k, $1 \leq k \leq r+1$, even if in the branch of each cube C_k, $1 \leq k \leq r+1$ we do not generate any solution containing columns from $(D_1 \cup \cdots \cup D_{k-1}) \cap O(A_p)$.*

Proof: Let S be a solution of $UCP(A)$ contained in $Gen(C)$ (as a matter of fact there may be many partial solutions from C covered by S). There are two cases:

1. There is a partial solution from C contained in S which covers A_p. Since $part1(C)$ contains all partial solutions from C covering A_p, the partial solution from C contained in S is in some of the cubes $C_1, \cdots, C_r$. Let C_k be the first of the cubes of $part1(C)$ containing the partial solution from C contained in S. Then the solution S is found in the branch of cube C_k. By hypothesis the solutions containing columns from $(D_1 \cup \cdots \cup D_{k-1}) \cap O(A_p)$ are excluded. But no column from S is contained in the set $(D_1 \cup \cdots \cup D_{k-1}) \cap O(A_p)$. Indeed, since S contains a partial solution from C, then $D_i \cap S \neq \emptyset, 1 \leq i \leq r$. E.g., for $r = 1$, if $D_1 \cap O(A_p) \cap S \neq \emptyset$, then C_1 contains a partial solution from C contained in S. If not, i.e., if $D_1 \cap O(A_p) \cap S = \emptyset$, then $(D_1 \setminus O(A_p)) \cap S \neq \emptyset$ (given $D_1 \cap S \neq \emptyset$, if $D_1 \cap O(A_p)$ does not contain a column from S, then there is a column from S contained in $D_1 \setminus O(A_p)$).

   In general, if for the first $k - 1 < r$ domains $D_1, \cdots, D_{k-1}$ intersecting A_p, it is true that $D_i \cap O(A_p), 1 \leq i \leq k-1$ does not contain a column from S, then there is a column from S contained in $D_i \setminus O(A_p), 1 \leq i \leq k-1$. If, for example, $D_k \cap O(A_p)$ contains a column from S, then the cube $C_k = D_1 \setminus O(A_p) \times \cdots \times D_{k-1} \setminus O(A_p) \times D_k \cap O(A_p) \times D_{k+1} \times \cdots \times D_d$ contains a partial solution from C contained in S and among the columns that we neglect (i.e., those in $(D_1 \cup \cdots \cup D_{k-1}) \cap O(A_p)$) in the branch of C_k there are no columns of S (because $D_i \cap O(A_p) \cap S = \emptyset, 1 \leq i \leq k-1$). So solution S can be found in this branch.

2. No partial solution from C contained in S covers A_p. Then partial solutions from C contained in S are in C_{r+1}. In the branch corresponding to C_{r+1} all solutions containing columns from $(D_1 \cup \cdots \cup D_r) \cap O(A_p)$ are excluded. But from the previous argument $D_i \cap O(A_p) \cap S = \emptyset, 1 \leq i \leq r$. So, again the solution S can be found in this branch.

□

Chapter 4

The Raising Procedure

Fig. 4.1 shows how the branch-and-bound algorithm of Fig. 2.1 is modified to incorporate the technique of incremental raise of the lower bound as discussed in Section 2.2. After the computation of the lower bound, if the gap $difference$ between the upper and lower bound is small, i.e., less than a global parameter $maxRaiser$, a new procedure $raiser$ is invoked with parameter $n = difference$. The parameter $maxRaiser$ currently is decided a-priori, but ideally it should be adapted dynamically. Intuitively if the gap is small, we conjecture that a search in this subtree will not improve the best solution and so we trigger the procedure $raiser$ that may either confirm the conjecture and prove that no better solution can be found here or disprove the conjecture and improve one or more times the best solution, updating the current one.

4.1 Overview of the Raising Algorithm

As anticipated in Section 2.3, we propose a *raiser* procedure, based on cube (row) branching [1]. Consider a covering matrix A, for which $A' = MSIR(A)$. 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 (3.1–3.4), $Rec(MSIR(A) + A_p, C)$ 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 (3.1–3.4).

[1]In the sequel we will use the expression n-*raiser* to denote an invocation of the *raiser* procedure with a given parameter n (e.g. we will use 1-*raiser* if n-*raiser* is invoked with $n = 1$)

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

Figure 4.1 AuraMincov: The Algorithm of Fig. 2.1 enhanced by incremental raising.

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.

When applying an *n-raiser*, we may prune the branches corresponding to cubes of more than $|MSIR(A)| + n$ domains. If at a node a row A_p is chosen such that no solution from the cube C of the node covers A_p, then there is no splitting of the cube, since *Rec* yields only one cube $C \times [O(A_p) \setminus (D_1 \cup \cdots \cup D_d)]$. The first matrix of the pair corresponding to a node gives a "lower bound submatrix" for the node. At each node the following reduction rule can be applied to the second matrix of the pair: if a row of the second matrix is covered by every solution of the cube C corresponding to the node, then the row can be removed from the matrix since, if we add it to the lower bound submatrix of the pair, then the recomputed cube will be equal to C.

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

1. There is a node such that there are no rows left in the second matrix of the pair and the corresponding cube has k domains, where $k < |MSIR| + n$. This means that the lower bound $|MSIR|$ cannot be improved by n. Any solution from the cube can be taken as the best current solution of $UCP(A)$.

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

4.1.1 Correctness of procedure *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 4.1.1 *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 3.1.*

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 $Gen(Rec(A'+A_p, S'))$ contains S. □

From Lemma 4.1.1 it follows that the Rec operation preserves the completeness of a set of subsolutions.

Theorem 4.1.1 *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. □

4.2 An Example of 1-raiser

As an example, apply an *1-raiser* to the following matrix A:

$$\begin{array}{c|ccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ \hline 1 & 0 & 0 & 0 & 1 & 1 & 0 & 1 \\ 2 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 3 & 0 & 1 & 1 & 1 & 0 & 1 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 5 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 6 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \end{array}$$

Suppose that the set of rows $A' = \{A_4, A_5, A_6\}$ is chosen as $MSIR(A)$. The set of irredundant solutions of $UCB(A')$ is represented by the cube $C = \{1,2\} \times \{3,5\} \times \{6,7\}$. The aim of applying an *1-raiser* to A is to improve the lower bound from 3 to 4. The root node of the search tree is specified by the cube C and the pair of matrices A', A'' where $Row(A'') = Row(A) \setminus Row(A')$.

Choose row A_3 from A'' to be added to A'. Since row A_3 intersects all three rows of A', according to (3.1–3.3) the set of all irredundant solutions of no more than 4 columns of $A' + A_3$ is given by the following expression:

$$\begin{aligned} C_1 &= \{2\} \times \{3,5\} \times \{6,7\}, \\ C_2 &= \{1\} \times \{3\} \times \{6,7\}, \\ C_3 &= \{1\} \times \{5\} \times \{6\}, \\ C_4 &= \{1\} \times \{5\} \times \{7\} \end{aligned}$$

$$\begin{aligned} part1(C) &= C_1 \cup C_2 \cup C_3, \quad part2(C) = C_4, \\ D &= D_1 \cup \ldots \cup D_d = \{1,2,3,5,6,7\} \\ Sol(C, A' + A_3) &= part1(C) \cup C_4 \times (O(A_3) \setminus D) \end{aligned}$$

where $O(A_3) = \{2,3,4,6\}$ and $D = D_1 \cup \ldots \cup D_d = \{1,2,3,5,6,7\}$, so that [2]

$$Sol(C, A' + A_3) \quad = \quad part1(C) \cup C_4 \times \{4\}.$$

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 and 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 $UCP(A' + A_3)$.

So the root node has four children nodes, each specified by one of the four cubes C_i and by the pair of matrices $A' + A_3$, $A'' - A_3$. Let us follow the branch corresponding to $C_1 = \{2\} \times \{3,5\} \times \{6,7\}$. Suppose that row 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, C_1 splits in: $C_{1_1} = part1(C_1) = \{2\} \times \{3\} \times \{6,7\}$, $C_{1_2} = part2(C_1) = \{2\} \times \{5\} \times \{6,7\}$.

So the node corresponding to C_1 has two branches whose pair of matrices are $A' + A_3 + A_2$ and $A'' - A_3 - A_2$. Let us follow the branch corresponding to the cube C_{1_1}. 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_{1_1}, we have the following splitting of C_{1_1}: $part1(C_{1_1}) = \{2\} \times \{3\} \times \{7\}$, $part2(C_{1_1}) = \{2\} \times \{3\} \times \{6\}$.

The branch corresponding to the cube $part1(C_{1_1})$ leads to the node at which the first matrix of the pair is equal to A and so the second is empty. This means that the cube $part1(C_{1_1})$ contains solutions of A of 3 columns (in this case only one solution) and so the lower bound cannot be raised to 4.

4.3 Detailed Description of the Raising Algorithm

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

[2]Notice that we used Equation 3.4. Instead applying Equation 3.1, 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: $\{1\} \times \{5\} \times \{7\} \times \{2\}$, $\{1\} \times \{5\} \times \{7\} \times \{3\}$, $\{1\} \times \{5\} \times \{7\} \times \{6\}$. It is a fact that they are all redundant; their irredundant counterparts are respectively: $\{5\} \times \{7\} \times \{2\}$, $\{1\} \times \{7\} \times \{3\}$, $\{1\} \times \{5\} \times \{6\}$, and they already appear in $part1(C)$.

```
$raiser(SolCube, n, A, lbound, bestSolution, ubound)$ {
    /* returns 1 if solutions in $SolCube$ raise lower bound of $A$ by $n$ */
    $stillToRaise = lbound + n - number\_domains(SolCube)$
    if ($stillToRaise \leq 0$) return 1
    /* If $A = \emptyset$ then $path$ + solutions of $A$ in $SolCube$ beats upper bound */
    if ($A = \emptyset$)   return $found\_solution(SolCube, n, bestSolution, ubound)$
    /* consider rows of $A$ not covered by any solution from $SolCube$ */
    $\mathcal{BSONIR} = find\_best\_set\_of\_non\_intersecting\_rows(A, SolCube)$
    foreach row $r_i \in \mathcal{BSONIR}$ { /* add a new domain for the columns covering $r_i \in A$ */
        $SolCube = add\_domain(SolCube, A, r_i)$
        $stillToRaise = stillToRaise - 1$
        if ($stillToRaise \leq 0$) return 1
    }
    $A = A \setminus \mathcal{BSONIR}$   /* Remove covered rows from $A$ and check again if $A$ is empty */
    if ($A = \emptyset$)   return $found\_solution(SolCube, n, bestSolution, ubound)$
    if ($stillToRaise = 1$) {
        /* Cover (with $SolCube$) and remove from $A$ the 1-intersecting rows */
        /* If 2 rows intersect 2 different cols in the same domain, prune the branch */
        if ($add\_set\_of\_1intersecting\_rows(A, SolCube) = 1$) return 1
        if ($A = \emptyset$)   return $found\_solution(SolCube, n, bestSolution, ubound)$
    }
    /* select next "best" row to be covered with $SolCube$ and remove it from $A$ */
    $r_i = select\_best\_uncovered\_row(A, SolCube)$
    $A = A \setminus \{r_i\}$
    /* splitting: $Part_1 = \{SolCube_1, \cdots, SolCube_k\}; Part_2 = \{SolCube_{k+1}\}$ */
    $split\_cubes(SolCube, A, r_i, Part_1, Part_2)$
    /* add to $SolCube_{k+1} \in Part_2$ new domain of the columns covering $r_i$ */
    $SolCube_{k+1} = add\_domain(SolCube_{k+1}, A, r_i)$
    /* branching on cubes of $Part_1$ and $Part_2$ */
    $returnValue = 1$
    while ($Part_1 \cup Part_2 \neq \emptyset$) {
        /* select first cubes from $Part_1$, then cube from $Part_2$ */
        $SolCube_j = get\_next\_cube(Part_1 \cup Part_2)$
        /* if a better global solution has been found set $returnValue$ to 0 */
        if ($raiser(SolCube_j, n, A, lbound, bestSolution, ubound) = 0$)
            $returnValue = 0$
    }
    return $returnValue$
}
```

Figure 4.2 The *raiser* algorithm.

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

Figure 4.3 Algorithm to handle terminal case $A = \emptyset$.

parameters are needed:

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

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

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

- $lbound$ is an input parameter for $raiser$ equal to $|MSIR|$. Notice that $lbound$ differs from the original lower bound [3] by a quantity equal to $|path|$, for consistency with the previous definition of n.

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

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

- *bestSolution* is the <u>output</u> of the procedure and contains the new best solution found by *raiser* if the lower bound could not be raised by n, otherwise is meaningless.

Fig. 4.2 shows the flow of *raiser*, the procedure that attempts to raise the lower bound of A. Notice that it requires a routine *split_cubes* which, for a selection of a row r_i covered by k of the d domains of *SolCube*, partitions *SolCube* in $k+1$ disjoint cubes, each of d domains; so $Part_1$ has k cubes of solutions from *SolCube* covering r_i, whereas $Part_2$ 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* [4], which measures the gap between the upper bound and the current lower bound, is less or equal to zero. If so, we know that the solutions in *SolCube* raise the lower bound of A by at least n, so that no solutions of A can beat the current upper bound. The second terminal case occurs when, after some recursive calls, A has become empty, and so any solution in the union of the solutions of A in *SolCube* together with the columns in the current *path* is the new best solution. Fig. 4.3 shows the housekeeping operations to update the variables *bestSolution*, *ubound* and n.

After performing these preliminary checks, the computation reaches the call of routine *find_best_set_of_non_intersecting_rows*, a routine which returns a set of rows of A denoted by the acronym $\mathcal{BSONIR}$. The code of this routine, which is reported in Figure 4.4, implements a fast heuristic to find a good subset of rows of A which do not intersect any domain of *SolCube* and which do not intersect each other. Ideally, we would like to get the best $\mathcal{BSONIR}$ which is a sort of "maximum set of independent rows" related to *SolCube*, but this would require the solution of another NP-complete problem. Therefore we are satisfied insert sequentially rows into $\mathcal{BSONIR}$ on the basis of the following criterion: we pick the largest row non intersecting neither a *solCube* domain or those row which have been just inserted into $\mathcal{BSONIR}$.

Once we have completed the previous selection, each row r_i in $\mathcal{BSONIR}$ is not covered by any solution encoded in *SolCube* and, therefore, we must add a new domain to *SolCube* made by the columns which cover r_i. While we are adding these new domains, we keep decreasing the variable *stillToRaise* and checking if its value becomes equal to zero. Finally, we can remove the

[4]By definition

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

```
$find\_best\_set\_of\_non\_intersecting\_rows(A, SolCube)$ {
    /* Heuristic returning the best set of rows non intersecting $solCube$ domains. */
    /* Ideally we would like the $MSIR$ among rows non intersecting $solCube$ domains. */
    $emptyInterRows = \emptyset$
    $bestRow = \emptyset$
    foreach row $r \in A$ {
        /* $\mathcal{D}$ is the set of $SolCube$ domains intersected by $r$ */
        $\mathcal{D} = compute\_set\_of\_intersected\_domains(SolCube, r)$
        if ($\mathcal{D} = \emptyset$) {
            $emptyInterRows = emptyInterRows \cup r$
            if $bestRow < r$
                $bestRow = r$
        }
    }
    /* If every row intersects $solCube$ domains then return the empty set */
    if ($emptyInterRows = \emptyset$)
        return $\emptyset$
    else {
        /* Let's build $\mathcal{BSONIR}$ starting from $bestRow$ */
        do {
            $\mathcal{BSONIR} = bestRow$
            $emptyInterRows = emptyInterRows \setminus bestRow$
            $previousBestRow = bestRow$
            $bestRow = \emptyset$
            /* Find the new $bestRow$ within $emptyInterRows$*/
            foreach row $r \in emptyInterRows$ {
                if $r \cap previousBestRow$
                    $emptyInterRows = emptyInterRows \setminus r$
                else if $bestRow < r$
                    $bestRow = r$
            }
        } while ($emptyInterRows \neq \emptyset$)
    }
    return $\mathcal{BSONIR}$
}
```

Figure 4.4 Algorithm to find the best set of rows non-intersecting *solCube*.

set $\mathcal{BSONIR}$ from A because the rows have been covered by the new added domains.

Notice that during the first call of *raiser* the set $\mathcal{BSONIR}$ is empty because $SolCube$ encodes the $MSIR$ and, by definition, every row not in the $MSIR$ must intersect at least one row in the $MSIR$. However, during the following recursive calls of *raiser* the original domains of $SolCube$ may change, namely decrease in cardinality due to the actions taken in the routines *split_cubes* and $add\_set\_of\_1intersecting\_rows(A, SolCube)$. Hence, at some node of the recursion tree, it may very well happen that a row of A is not covered anymore by any domain of $SolCube$.

After having removed the rows belonging to $\mathcal{BSONIR}$, another optimization step can be applied successively before splitting $SolCube$. If at this point *stillToRaise* is equal to 1, it means that we have already raised the lower bound by $n-1$. Therefore, if we are forced to add one more domain to $SolCube$, then we can prune the current branch. Hence, a simple condition which leads immediately to pruning is the following: consider two rows r_1 and r_2 of A which intersect $SolCube$ only in one domain $d = \{c^1, c^2, \cdots, c^l\}$, and suppose that r_1 intersects only the column c^i, while r_2 intersects only the column c^j. This fact allows us to prune the current branch because to cover one of the rows we can choose either one of the two distinct columns of the domain. Without loss of generality, say that we cover r_1 with c^i, then to cover r_2 we must use a column which does not belong to any domain of $SolCube$ and so we are forced to add one more domain to $SolCube$, thereby raising the lower bound by n.

Figure 4.5 illustrates the procedure $add\_set\_of\_1intersecting\_rows(A, SolCube)$ which exploits the previous situation and, in practice, is invoked often because the condition $stillToRaise = 1$ happens very commonly in hard problems. Basically, the routine is based on two nested cycles. The external cycle is repeated until the internal cycle does not modify $SolCube$ anymore. The internal cycle computes, for each row r of A, the set D of the domains of $SolCube$ intersected by r. If the cardinality of D is equal to 1, e.g., $D = \{d\}$, we remove from d all the columns which are not intersected by r and then we remove r from A, since r has been covered.

Notice that $add\_set\_of\_1intersecting\_rows$ is called just after we removed from A the set of non-intersecting rows $\mathcal{BSONIR}$ and therefore all the remaining rows of A intersect at least one domain of $SolCube$. However, after cycling inside this routine and removing some columns (which makes "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

```
$add\_set\_of\_1intersecting\_rows(A, SolCube)$ {
    /* 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 $\in A$ {
            /* $\mathcal{D}$ is the set of $SolCube$ domains intersected by $r$ */
            $\mathcal{D} = compute\_set\_of\_intersected\_domains(SolCube, r)$
            if ($|\mathcal{D}| = 1$) {
                $reducingDomains = TRUE$
                /* 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 $A$ */
                $A = A \setminus \{r\}$
            }
            else if ($|\mathcal{D}| = 0$) {
                /* After removing some columns, a row may not be */
                /* covered anymore, so current branch must be pruned. */
            }
            /* else ($|\mathcal{D}| > 1$): do nothing */
            /* because r is not a 1-intersecting row */
        }
    } while ($reducingDomains$)
    return 0
}
```

Figure 4.5 Algorithm to handle the 1-intersecting rows.

but, at least a certain number of rows have been removed from A and the corresponding intersected domains of $SolCube$ have been made "leaner". After calling $add\_set\_of\_1intersecting\_rows$ and removing 1-intersecting rows, it is possible that A has become empty. If so, $raiser$ calls $found\_solution$ to update the variables $bestSolution$, $ubound$ and n.

After all these special cases have been addressed, we must select a new row r_i to be covered with $SolCube$. The row r_i is removed from A and drives the splitting of $SolCube$. The selection of r_i is performed by $select\_best\_uncovered\_row$, shown in Fig. 4.6. The strategy to select the best row in order to split the current $SolCube$, before calling recursively $raiser$, looks for the row of A which intersects the minimum number of domains of $SolCube$. The reason is to reduce the number of branches from the node, i.e., the number of domains intersecting the row to be added plus 1. Notice that at this stage each row of A intersects at least 2 domains of $SolCube$. In case of ties between different rows, the row having the highest weight is chosen. The weight of a row A_p is defined as:

$$\prod_{k=1}^{m} \frac{|D'_{i_k}|}{|D_{i_k}|}$$

where m is the number of domains of $SolCube$ intersecting A_p, D_{i_k} is a domain intersected by A_p and $D'_{i_k} = D_{i_k} \setminus O(A_p)$. So the weight of A_p is just the fraction of solutions from $SolCube$ that do not cover A_p, 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 $SolCube$. Such a row is simply removed from A'' and added to A'.

The splitting of $SolCube$ is done as explained in Section 3.2. 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 once or more times, $raiser$ returns 0 after having updated the current best solution and upper bound.

```
$select\_best\_uncovered\_row(A, SolCube)$ {
    /* Return the row which intersects fewer domains of $SolCube$. */
    /* When it is called each row of $A$ intersects at least one domain */
    $bestIntersectedRowNum = \infty$
    $bestWeight = 0$
    foreach row $r \in A$ {
        $intersectedRowNum = 0$
        $weight = 1$
        foreach domain $D \in SolCube$ {
            if $(r \cap D)$ {
                $intersectedRowNum = intersectedRowNum + 1$
                $D_2 = rowMinus(D, r)$
                $w = | D_2 | / | D |$
                $weight = weight * w$
            }
        }
        if $(intersectedRowNum < bestIntersectedRowNum)$ {
            $bestIntersectedRowNum = intersectedRowNum$
            $bestWeight = weight$
            $bestRow = r$
        } else if $(intersectedRowNum = bestIntersectedRowNum)$ {
            /* Tiebreaker: pick the row with the highest weight */
            if $(weight > bestWeight)$ {
            $bestIntersectedRowNum = intersectedRowNum$
            $bestWeight = weight$
            $bestRow = r$
        }
    }
    return $bestRow$
}
```

Figure 4.6 Algorithm to select the best row to be covered.

Chapter 5

Of Lower Bounds there is No End

5.1 Maximal Independent Set Lower Bound

The cardinality of a maximum set of pairwise disjoint rows (i.e., there are no 1s in the same column) is a lower bound on the cardinality of the solution to the covering problem, because a different element must be selected for each of the independent rows in order to cover them. If the size of current solution plus the size of the independent set is greater or equal to the best solution seen so far, the search along this branch can be terminated because no solution better than the current one can possibly be found. Since finding a maximum independent set is an NP-complete problem, in practice a heuristic is used that provides a weaker lower bound. Notice that even the lower bound provided by solving exactly the maximum independent set problem is not sharp. In [8] an example of size $O(n^2)$ is given, whose minimal solution has cost $O(n)$, but whose lower bound by independent set is 1. In practice a lower bound by independent set is poor when the covering matrix is dense.

5.2 Limit Lower Bound

In [1] new rules to prune the search space were introduced. One such rule, called **limit lower bound**, has been shown of great effectiveness in practice. Given a covering problem A that corresponds to a node of the computation tree N, define the following notation: let $A.min$ be the cost of a minimum solution, $A.lower$ the value of a lower bound on $A.min$, $A.path$ the cost of the partial solution from the root to node N, and $A.upper$ the cost of the best solution found so far. Then the following holds.

```
$mincov(A, path, weight, lbound, ubound)$ {
    /* Apply row dominance, column dominance, select essentials and, if it is possible, Gimpel's reduction */    (1)(2)
    if (not $reduce(A, path, weight, ubound)$) return $empty\_solution$
    if ($gimpel\_reduce(A, path, weight, lbound, ubound, best)$) return $best$
    $MSIR = maximal\_independent\_set(A, weight)$    (3)
    $lbound\_new = max(cost(path) + cost(MSIR), lbound)$    (4)
    /* Test if it is possible to apply Limit Lower Bound */    (4a)
    $emptyIntersection$ = true
    while (($A \neq \emptyset$) and ($lbound\_new + 1 \geq ubound$) and ($emptyIntersection$)) {
        /* Remove from $A$ columns having no intersection with $MSIR$ */    (4b)
        $emptyIntersection$ = false
        foreach column $c \in A$
            if (not $check\_intersection(MSIR, c)$) { $A = A \setminus \{c\}$   $emptyIntersection$ = true }
        if (not $reduce(A, path, weight, ubound)$) return $empty\_solution$
        $MSIR = maximal\_independent\_set(A, weight)$
        $lbound\_new = max(cost(path) + cost(MSIR), lbound)$
    }
    /* Bounding based on no better solution possible */    (5)
    if ($lbound\_new \geq ubound$)   $best = empty\_solution$
    else if ($A$ is empty)  { $best = solution\_dup(path)$ }   /* New best solution at current level */    (6)
    } else if ($block\_partition(A, A_1, A_2)$ gives non-trivial bi-partitions) {    (7)
        $path1 = empty\_solution$
        $best1 = mincov(A_1, path1, weight, 0, ubound - cost(path))$    (8)
        if ($best1 = empty\_solution$) $best = empty\_solution$  /* Add best solution to the selected set */    (9)
        else { $path = path \cup best1$;  $best = mincov(A_2, path, weight, lbound\_new, ubound)$}    (10)
    } else { /* Branch on cyclic core and recur */    (11)
        $branch = select\_column(A, weight, MSIR)$
        $path1 = solution\_dup(path) \cup branch$
        let $A_{branch}$ be the reduced table assuming $branch$ in solution    (12)
        $best1 = mincov(A_{branch}, path1, weight, lbound\_new, ubound)$
        /* Update the upper bound if we found a better solution */    (13)
        if ($best1 \neq empty\_solution$) /* It implies ($ubound > cost(best1)$) */
            $ubound = cost(best1)$
        /* Do not branch if lower bound matched */    (14)
        if ($best1 \neq empty\_solution$) and ($cost(best1) = lbound\_new$) return $best1$
        let $A_{\overline{branch}}$ be the reduced table assuming $branch$ not in solution    (15)
        $best2 = mincov(A_{\overline{branch}}, path, weight, lbound\_new, ubound)$
        $best = best\_solution(best1, best2)$
    }
    return $best$
}
```

Figure 5.1 The Algorithm of Fig. 2.1 enhanced by the "limit lower bound" technique.

Theorem 5.2.1 **(Limit lower bound)**. *Given a binate covering problem A, let I be an independent set of the rows, i.e., a set of unate rows intersecting no common column. Let $A.lower$ be a lower bound from the independent set I, i.e., the sum of a minimum cost column for each row in I. Consider the set B of the columns b that do not intersect rows in I and such that $b \in B$ only if*

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

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

A proof can be found in [2]. In practice in the common case that all columns have cost 1 if included in a solution, one needs only to check whether

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

If so, all the columns that do not intersect rows in the independent set I can be removed [1]. Experimental results in [1] on exact two-level minimization show strong gains by this new pruning technique, resulting in reductions of the search space up to three orders of magnitude.

Fig. 5.1 shows the branch-and-bound algorithm of Fig. 2.1 enhanced by the limit lower bound. When the condition $lbound\_new + 1 \geq A.upper$ is true, the columns of A which do not intersect the $MSIR$ are deleted . Then the matrix is reduced again and the $MSIR$ is recomputed. This sequence of actions is iterated as long as $lbound\_new + 1 \geq A.upper$ holds and until nothing changes.

5.3 Lower Bound by Incremental Raising

We develop an example that shows how to raise the lower bound incrementally by means of our technique, developed in Chapter 4. Consider the following matrix A_N that cannot be reduced

[1]Together with the rows that they intersect in a 0, in instances of binate covering (see Chapter 7).

by dominance.

$$
\begin{array}{c|cccccccccc}
 & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
2 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
3 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
4 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
5 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
6 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
7 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
8 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\
9 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0
\end{array}
$$

Suppose that A_N is the submatrix corresponding to the node N of a column branching search tree, such that $ubound = 6$ and $|path(A_N)| = 0$.

An $MSIR$ is made by the 4 rows A_0, A_1, A_2 and A_3. Since $ubound - |path(A_N)| - |MSIR| = 2$, the limit lower bound does not apply. Instead we can apply a 2-*raiser*. Initially the cube of solutions is $C_0 = \{0,1\} \times \{2,3\} \times \{4,5\} \times \{6,7\}$. Select row A_4 from the rest of the matrix. Applying the operator *Rec*, the cube C_0 splits into two cubes: $C_1 = \{0\} \times \{2,3\} \times \{4,5\} \times \{6,7\}$ and $C_2 = \{1\} \times \{2,3\} \times \{4,5\} \times \{6,7\} \times \{8\}$.

Consider the branch corresponding to C_1. Select row A_5 that is not covered by any solution in C_1. So a domain must be added to C_1, which becomes $C_1' = \{0\} \times \{2,3\} \times \{4,5\} \times \{6,7\} \times \{1,8\}$ Now select row A_6, which intersects only the second domain of C_1'. As a result C_1' becomes $C_1'' = \{0\} \times \{2\} \times \{4,5\} \times \{6,7\} \times \{1,8\}$. But no solution in C_1'' covers row A_7 and therefore one should add one more domain, and so the lower bound is raised to 6 and we can prune the search.

Consider the branch corresponding to C_2. If we select row A_6, which intersects only the second domain of C_2, then C_2 becomes $C_2' = \{1\} \times \{2\} \times \{4,5\} \times \{6,7\} \times \{8\}$. But no solution in C_2' covers row A_7 and therefore one should add one more domain, and so the lower bound is raised to 6 and we can prune the search.

Summarizing, by using a 2-*raiser* the search requires 1 node of the column branching search tree and 3 nodes of the row branching search tree. The same example with the limit lower bound requires 5 nodes of the column branching search tree. Finally 9 nodes are required with a standard implementation that relies only on the $MSIR$ to find a lower bound. It is important to notice that a node of the row branching search tree is much less expensive than a node of the column branching search tree.

For ease of comparison, Fig. 5.2 shows the column branching search tree of the matrix A_N, constructed by calling the original *mincov* of ESPRESSO. We explain how the parameters change at each node. We refer to the numbers of the nodes in the picture; notice that to the upper right of each node there is a pair of numbers, being respectively *lbound* (left) and *ubound* (right). The reader is advised to follow the run on the algorithm presented in Fig. 2.1. The procedure *mincov* has been called on matrix $A_{N_1} = A_N$ with $ubound = 6$ to simulate the assumption that A_N is a submatrix at the node N of a column branching tree, whose root starts with a matrix A, of which the current best solution has cardinality 6 [2].

Node 1 Parameters of *mincov*: $lbound = 0$, $ubound = 6$, $path = \emptyset$,

$$A_{N_1} = \left[\begin{array}{r|cccccccccc} & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \hline 0: & 1 & 1 & . & . & . & . & . & . & . & . \\ 1: & . & . & 1 & 1 & . & . & . & . & . & . \\ 2: & . & . & . & . & 1 & 1 & . & . & . & . \\ 3: & . & . & . & . & . & . & 1 & 1 & . & . \\ 4: & 1 & . & . & . & . & . & . & . & 1 & . \\ 5: & . & 1 & . & . & . & . & . & . & 1 & . \\ 6: & . & . & 1 & . & . & . & . & . & . & 1 \\ 7: & . & . & . & 1 & . & . & . & . & . & 1 \\ 8: & . & . & . & . & . & 1 & 1 & . & . & . \\ 9: & . & . & . & . & 1 & . & . & 1 & . & . \end{array}\right]$$

No reduction of A_1 is possible. $MSIR = \{0, 1, 2, 3\}$. After recomputation of the maximal independent set we have $lbound = 4$, $ubound = 6$, $path = \emptyset$. Matrix A_{N_1} is decomposed into two submatrices A_{N_2} and A_{N_5}.

Node 2 Parameters of *mincov*: $lbound = 0$, $ubound = 6$, $path = \emptyset$,

$$A_{N_2} = \left[\begin{array}{r|ccc} & 0 & 1 & 8 \\ \hline 0: & 1 & 1 & . \\ 4: & 1 & . & 1 \\ 5: & . & 1 & 1 \end{array}\right]$$

[2]This assumption has been made in order to build a simple example which brings out the different behavior of the algorithms being compared.

No reduction of A_2 is possible. $MSIR = \{0\}$. After recomputation of the maximal independent set we have *lbound* $= 1$, *ubound* $= 6$, *path* $= \emptyset$. Branching on column 0.

Node 3 Parameters of *mincov*: *lbound* $= 1$, *ubound* $= 6$, *path* $= \{0\}$.

$$A_{N_3} = \left[\begin{array}{r|cc} & 1 & 8 \\ \hline 5: & 1 & 1 \end{array}\right]$$

A_3 is empty after column dominance and selection of essential column 1. After recomputation of the maximal independent set we have *lbound* $= |path| + |MSIR| = 1 + 1 = 2$, *ubound* $=$ 6, *path* $= \{0, 1\}$. Returns the solution *best* $= \{0, 1\}$. Back to node 2, *ubound* $= |best| = 2$.

Node 4 Parameters of *mincov*: *lbound* $= 1$, *ubound* $= 2$, *path* $= \emptyset$.

$$A_{N_4} = \left[\begin{array}{r|cc} & 1 & 8 \\ \hline 0: & 1 & . \\ 4: & . & 1 \\ 5: & 1 & 1 \end{array}\right]$$

During reduction, after row dominance and selection of essential columns 1 and 8, this node is pruned because $|path| = 2 \geq ubound = 2$.

Node 5 Parameters of *mincov*: *lbound* $= 4$, *ubound* $= 6$, *path* $= best_{Node1} = \{0, 1\}$.

$$A_{N_5} = \left[\begin{array}{r|ccccccc} & 2 & 3 & 4 & 5 & 6 & 7 & 9 \\ \hline 1: & 1 & 1 & . & . & . & . & . \\ 2: & . & . & 1 & 1 & . & . & . \\ 3: & . & . & . & . & 1 & 1 & . \\ 6: & 1 & . & . & . & . & . & 1 \\ 7: & . & 1 & . & . & . & . & 1 \\ 8: & . & . & . & 1 & 1 & . & . \\ 9: & . & . & 1 & . & . & 1 & . \end{array}\right]$$

No reduction of A_5 is possible. $MSIR = \{1, 2, 3\}$. After recomputation of the maximal independent set we have *lbound* $= |path| + |MSIR| = 2 + 3 = 5$, *ubound* $= 6$, *path* $=$ $\{0, 1\}$. Matrix A_{N_5} is decomposed into two submatrices A_{N_6} and A_{N_9}.

Node 6 Parameters of *mincov*: $lbound = 0$, $ubound = ubound_{Node5} - |path_{Node5}| = 6 - 2 = 4$, $path = \emptyset$.

$$A_{N_6} = \left[\begin{array}{r|ccc} & 2 & 3 & 9 \\ \hline 1: & 1 & 1 & . \\ 6: & 1 & . & 1 \\ 7: & . & 1 & 1 \end{array}\right]$$

No reduction of A_6 is possible. $MSIR = \{1\}$. After recomputation of the maximal independent set we have $lbound = 1$, $ubound = 4$, $path = \emptyset$. Branching on column 2.

Node 7 Parameters of *mincov*: $lbound = 1$, $ubound = 4$, $path = \{2\}$.

$$A_{N_7} = \left[\begin{array}{r|cc} & 3 & 9 \\ \hline 7: & 1 & 1 \end{array}\right]$$

A_7 is empty after column dominance and selection of essential column 3. After recomputation of the maximal independent set we have $lbound = |path| + |MSIR| = 1+1 = 2$, $ubound = 4$, $path = \{2, 3\}$. Returns the solution $best = \{2, 3\}$. Back to node 6, $ubound = |best| = 2$.

Node 8 Parameters of *mincov*: $lbound = 1$, $ubound = 2$, $path = \emptyset$.

$$A_{N_8} = \left[\begin{array}{r|cc} & 3 & 9 \\ \hline 1: & 1 & . \\ 6: & . & 1 \\ 7: & 1 & 1 \end{array}\right]$$

During reduction, after row dominance and selection of essential columns 3 and 9, this node is pruned because $|path| = 2 \geq ubound = 2$.

Node 9 Parameters of *mincov*: $lbound = 5$, $ubound = 6$, $path = path_{Node5} \cup best1_{Node5} = \{0, 1\} \cup \{2, 3\} = \{0, 1, 2, 3\}$.

$$A_{N_9} = \left[\begin{array}{r|cccc} & 4 & 5 & 6 & 7 \\ \hline 2: & 1 & 1 & . & . \\ 3: & . & . & 1 & 1 \\ 8: & . & 1 & 1 & . \\ 9: & 1 & . & . & 1 \end{array}\right]$$

No reduction of A_9 is possible. $MSIR = \{2, 3\}$. So the lower bound becomes $|path| + |MSIR| = 4 + 2 = 6$ and this node is pruned at line (5) of Fig. 2.1 because $lbound = 6 \geq ubound = 6$.

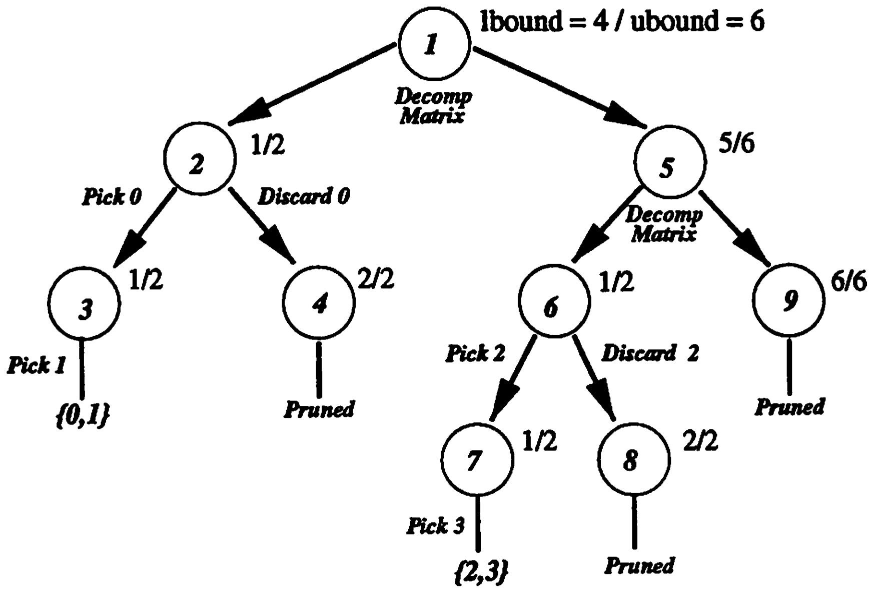

Figure 5.2 Search tree of A_N in Section 5.3 by *mincov* of ESPRESSO.

The procedure mincov enhanced by the limit lower bound prunes the previous search tree at Node 5. More precisely, it discovers that $lbound + 1 = 5 + 1 = 6 \geq ubound = 6$ and so it removes from the matrix

$$A_{N_5} = \left[\begin{array}{c|ccccccc} & 2 & 3 & 4 & 5 & 6 & 7 & 9 \\ \hline 1: & 1 & 1 & . & . & . & . & . \\ 2: & . & . & 1 & 1 & . & . & . \\ 3: & . & . & . & . & 1 & 1 & . \\ 6: & 1 & . & . & . & . & . & 1 \\ 7: & . & 1 & . & . & . & . & 1 \\ 8: & . & . & . & 1 & 1 & . & . \\ 9: & . & . & 1 & . & . & 1 & . \end{array}\right]$$

column 9 which does not intersect any row of the $MSIR = \{1,2,3\}$. The result is the matrix

$$A_{N_5'} = \left[\begin{array}{c|cccccc} & 2 & 3 & 4 & 5 & 6 & 7 \\ \hline 1: & 1 & 1 & . & . & . & . \\ 2: & . & . & 1 & 1 & . & . \\ 3: & . & . & . & . & 1 & 1 \\ 6: & 1 & . & . & . & . & . \\ 7: & . & 1 & . & . & . & . \\ 8: & . & . & . & 1 & 1 & . \\ 9: & . & . & 1 & . & . & 1 \end{array}\right]$$

whose $MSIR$ is now $\{6,7,8,9\}$. This raises the lower bound to $lbound = |path| + |MSIR| = 2+4 = 6$, enabling to prune the node because $lbound = 6 \geq ubound = 6$ and so no better solution is possible.

Chapter 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 previously reported. Its main features described in the literature [1, 8, 6] 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 classes: in Table 6.1 there are difficult cases from the collection of ESPRESSO (we start from the matrix obtained by ESPRESSO after removing the essential primes), in Table 6.2 there are random generated matrices with varying row/column ratios and densities, in Table 6.3 there are matrices encoding constraints satisfaction problems from [9]. For each of these matrices, we report in Table 6.4 their size and their sparsity. The experiments were 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 it 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 for comparison results 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. 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 6.1, but has a less effective pruning strategy in those of Tables 6.2 and 6.3, 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 [10]. 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 [10] 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.

matrix	Sol.	ESPRESSO		SCHERZO		MINCOV_LLB		AURA		
		nodes	time	nodes	time	nodes	time	nodes/A-nodes	time	r
exps	76	13	0.0	na	na	13	0.0	13/0	0.0	3
fout	38	161	1.3	na	na	49	0.7	18/44	0.2	2
max512	113	111	1.4	na	na	25	0.4	19/25	0.4	3
addm4	165	121	3.6	na	na	29	1.1	17/11	0.6	2
mlp4	109	2122	22.6	24	0.1	153	4.3	34/206	1.3	3
pdc	94	195	62.7	44	6.1	88	58	41/132	52.9	3
lin.rom	120	370	29.1	238	4.7	106	10.1	61/240	7.7	3
ex5	37	-	time	616091	2450.5	597644	214300	155/169245	1315.2	4
prom2	278	-	time	25993	5149.2	-	time	1478/1097624	24071.4	3
max1024	245	-	time	531618	9583.6	-	time	12402/3850628	36240	3

Table 6.1 Results from *Espresso Benchmarks*

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

Table 6.2 Results from *Random Generated Matrices*

matrix	Sol.	ESPRESSO		SCHERZO		MINCOV_ILB		AURA		
		nodes	time	nodes	time	nodes	time	nodes/A-nodes	time	r
bbara	7	61	0.02	0	0.0	7	0	7/2	0	3
dk512x	6	213	0.24	55	0.0	57	0.15	9/24	0.04	3
ex4inp	5	5279	16.81	17	0.3	19	0.66	9/14	0.27	3
ex5inp	4	64	0.05	4	0.0	6	0.01	6/2	0.01	3
ex6inp	4	639	0.54	35	0.0	103	0.28	7/23	0.03	3
maincont	7	504	0.69	68	0.0	101	0.4	11/12	0.06	3
opus	5	121	0.1	7	0.0	5	0.01	5/2	0.01	3
ricks	5	20	0.37	10	0.2	12	0.36	8/43	0.33	3
saucier	6	-	mem	186927	5441.0	-	mem	10/76	222.47	3

Table 6.3 Results from *Encoding Problem Matrices*

matrix	R × C (Sparsity)	Sol.
exps	680 × 696 (1.2%)	76
fout	177 × 431 (2.4%)	38
max512	559 × 515 (1.3%)	113
addm4	832 × 1073 (0.6%)	165
mlp4	530 × 594 (0.99%)	109
pdc	6904 × 19021 (0.34%)	94
lin.rom	1030 × 1076 (0.9%)	120
ex5	831 × 2428 (2%)	37
prom2	1924 × 2611 (0.31%)	278
max1024	1090 × 1264 (0.52%)	245
tc.90	50 × 100 (90%)	2
tc.70	50 × 100 (70%)	2
tc.50	50 × 100 (50%)	3
tc.30	50 × 100 (30%)	4
tc.10	50 × 99 (10%)	8
tr.10	100 × 50 (20%)	8
tr.20	100 × 50 (40%)	5
tr.30	100 × 50 (60%)	3
tr.40	100 × 50 (80%)	2
ts.90	100 × 100 (90%)	2
ts.70	100 × 100 (70%)	3
ts.50	100 × 100 (50%)	4
ts.30	100 × 100 (30%)	5
ts.10	100 × 100 (10%)	12
bbara	45 × 26 (41%)	7
dk512x	91 × 59 (45%)	6
ex4inp	91 × 240 (46%)	5
ex5inp	36 × 34 (48%)	4
ex6inp	28 × 96 (48%)	4
maincont	105 × 67 (35%)	7
opus	45 × 63 (45%)	5
ricks	78 × 363 (47%)	5
saucier	171 × 6207 (47%)	6

Table 6.4 Characteristics of the *Benchmarks*

Chapter 7

Future Work

7.1 Extension to Binate Covering

7.1.1 Definition of Binate Covering Problem and Related Work

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

Suppose that a set $S = \{s_1, \ldots, s_n\}$ is given. The cost of s_i is c_i where $c_i \geq 0$. By associating a binary variable x_i to s_i, which is 1 if s_i is selected and 0 otherwise, the binate covering problem (BCP) can be defined as finding $S' \subseteq S$ that minimizes

$$\sum_{i=1}^{n} c_i x_i, \tag{7.1}$$

subject to the constraint

$$A(x_1, x_2, \ldots, x_n) = 1, \tag{7.2}$$

where A is a boolean function, sometimes called the constraint function. The constraint function specifies a set of subsets of S that can be a solution. No structural hypothesis is made on A. Binate refers to the fact that A is in general a binate function (a function is binate if it has at least a binate variable). BCP is the problem of finding an onset minterm of A that minimizes the cost function (i.e., a solution of minimum cost of the boolean equation $A(x_1, x_2, \ldots, x_n) = 1$).

If A is given in product-of-sums form, finding a satisfying assignment is exactly the problem SAT, the prototypical NP-complete problem ([11]). In this case it also possible to write A as an array of cubes (that form a matrix with coefficients from the set $\{0, 1, 2\}$). Each variable of A is a column and each sum (or clause) is a row and the problem can be interpreted as one of finding a subset C of columns of minimum cost, such that for every row r_i, either

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

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

In other words, each clause must be satisfied by setting to 1 a variable appearing in it in the positive phase or by setting to 0 a variable appearing in it in the negative phase. If A is given in product-of-sums form one can say that the assignment of a variable to 0 or 1 covers some rows that are satisfied by that choice. The product-of-sums A is called covering matrix or covering table. In a unate covering problem, the coefficients of A are restricted to the values 1 and 2 and only the first condition must hold.

As an example of binate covering formulation of a well-known logic synthesis problem consider the problem of finding the minimum number of prime compatibles that are a minimum closed cover of a given FSM. A binate covering problem can be set up, where each column of the table is a prime compatible and each row is one of the covering or closure clauses of the problem [12]. There are as many covering clauses as states of the original machine and each of them states that a state is covered by any of the prime compatibles in which it is contained. There are as many closure clauses as prime compatibles and each of them states that if a given prime compatible is chosen, then for each implied class in the corresponding class set one of the prime compatibles containing it must be chosen too.

In the matrix representation, entry (i, j) is 1 or 0 according to the phase of the literal corresponding to prime j in clause i; if such a literal is absent the entry is 2.

Various techniques have been proposed to solve binate covering problems. A class of them [13, 14] are branch-and-bound techniques that build explicitly the table of the constraints expressed as product-of-sum expressions and explore in the worst-case all possible solutions, but avoid the generation of some of the suboptimal solutions by a clever use of reduction steps.

A second approach [15] formulates the problem with Binary Decision Diagrams (BDD's) and reduces finding a minimum cost assignment to a shortest path computation. In that case the number of variables of the BDD is the number of columns of the binate table.

A mixed technique has been proposed in [16] by Jeong and Somenzi in [16]. It is a branch-and-bound algorithm, where the clauses are represented as a conjunction of BDD's. The usage of BDD's leads to an effective method to compute a lower bound on the cost of the solution.

Notice that unate covering is a special case of binate covering. Therefore techniques for the latter solve also the former. In the other direction, exact state minimization, a problem naturally formulated as a binate covering problem, can be reduced to a unate covering problem, after the generation of irredundant prime closed sets [17]. But there is a catch here: the cost function is not anymore additive, so that the reduction techniques so convenient to solve covering problems, are not anymore applicable as they are.

7.1.2 Computation of MSIR

For applying the paradigm of "negative" thinking to the binate covering problem (BCP) we need to start from the computation of $MSIR$ as we do for the unate covering problem (UCP). In the BCP case we compute a $MSIR$ of the rows having only 1s and dashes (as it is done in standard implementations of the binate solver) and then we augment it maximally by adding non-intersecting rows. The latter rows are useless for lower bound estimation, but should be added to the lower bound submatrix, when forming an initial matrix for the *raiser* procedure, because further addition of rows and recalculation of the solution space may discover sooner costly cubes of solutions to be bounded away.

For instance, suppose that an augmented $MSIR$ consists of three rows $\{x_1, x_2\}$, $\{x_3, x_4\}$ and $\{\overline{x}_5, x_6\}$. From the lower bound point of view the last row is useless, since it does not contribute to the lower bound estimate (equal to 2) due to the costless assignment $x_5 = 0$. Suppose to add row $\{\overline{x}_5, x_7\}$ to the previous $MSIR$. After recomputation there are two cubes of solutions. The first cube $\{x_1, x_2\} \times \{x_3, x_4\} \times \{\overline{x}_5\}$ corresponds to the assignment $x_5 = 0$ and contains solutions of cost 2. The second cube $\{x_1, x_2\} \times \{x_3, x_4\} \times \{x_5 x_6\} \times \{x_7\}$ corresponds to the assignment $x_5 = 1$ and contains solutions of cost 5.

7.1.3 Cubes of Solutions

We revise the definition of solution cube to accommodate the fact that a solution may require positive and negative literals. A solution cube is a set of solutions represented by

$$C = D_1 \times D_2 \times \cdots \times D_d$$

where D_i is a set of partial solutions consisting of assignments to some variables from the set $var(D_i)$, which is the support of D_i. The sets $var(D_i), i = 1, \cdots, n$ are disjoint. We define the minimal cost of the solutions contained in C as $cost(D_1) + \cdots + cost(D_d)$, where $cost(D_i)$ is the minimal cost of the solutions contained in D_i.

An example of a cube of solutions is $C = D_1 \times D_2$, where $D_1 = \{\overline{x}_1 x_2, x_1, x_2 x_3\}$, $var(D_1) = \{x_1, x_2, x_3\}$ $D_2 = \{x_4 x_5, x_5 x_6\}$, $var(D_2) = \{x_4, x_5, x_6\}$. For instance, $\overline{x}_1 x_2$ denotes the partial solution with $x_1 = 0$ and $x_2 = 1$. The cube C contains 6 ($6 = 3 \times 2$) solutions. Since $cost(D_1) = 1$ ($\overline{x}_1 x_2$ and x_1 have both cost 1) and $cost(D_2) = 2$ ($x_4 x_5$ and $x_5 x_6$ have both cost 2), then $cost(C) = 1 + 2 = 3$.

Notice that in the unate covering formulation there are no products of literals in a domain D, which then consists only of a collection of single literals.

7.1.4 Recomputation of Solutions

The recalculation $Rec(C, A' + A_p)$ of C can be described by formulas structurally similar to Equations (3.1-3.4), but with a different interpretation of the involved operations. Let the cube C of solutions of matrix A' be

$$C = D_1 \times D_2 \times \cdots \times D_d$$

Denote by $var(C)$ the set $var(D_1) \cup \cdots \cup var(D_d)$ and by $var(A_p)$ the set of variables occurring in row A_p.

Equation 3.4 is modified to

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

where $sol'(Ap)$ is the set of solutions covering A_p and consisting of variables $var(A_p) \setminus var(C)$. Moreover, in the definition of $part1(C)$ and $part2(C)$ the operators $\cap$ and $\setminus$ must be replaced as follows:

1. Instead of the operation $D_i \cap O(Ap)$ use the operation $D_i \hat{\cap} sol(Ap)$, where $sol(A_p)$ are solutions covering A_p consisting of variables in $var(D_i)$. Operation $\hat{\cap}$ returns all irredundant solutions either

   (a) included in D_i and covering A_p, or

   (b) extensions (by setting variables from $var(D_i) \cap var(A_p)$) which cover A_p of solutions in D_i not covering A_p.

2. Instead of the operation $D_i \setminus O(A_p)$ use the operation $D_i \tilde{\setminus} O(A_p)$ returning all irredundant solutions either
   (a) included in D_i, not covering A_p and not extendable to cover A_p (by setting variables from $var(D_i) \cap var(A_p)$), or
   (b) extensions (again, by setting variables from $var(D_i) \cap var(A_p)$) which do not cover A_p of solutions in D_i not covering A_p.

We introduce an example to clarify the previous extension to the binate case of the recomputation rule presented in Chapter 3 for the unate case. Let $C = D_1 \times D_2$ be a cube of solutions, with $D1 = \{x_1\overline{x}_2, x_3\}$ and $D2 = \{\overline{x}_4, \overline{x}_5\}$ and let $A_p = \overline{x}_1 + x_5 + x_7$ be the row to be added. After the recomputation of solutions, cube C yields three cubes C_1, C_2 and C_3:

$$\begin{aligned}
C_1 &= \{x_3\overline{x}_1\} \times \{\overline{x}_4, \overline{x}_5\} \\
C_2 &= \{x_1\overline{x}_2, x_3x_1\} \times \{\overline{x}_4x_5\} \\
C_3 &= \{x_1\overline{x}_2, x_3x_1\} \times \{\overline{x}_5, \overline{x}_4\overline{x}_5\} \\
part1(C) &= C_1 \cup C_2 \\
part2(C) &= C_3 \\
Rec(A' + A_p, C) &= part1(C) \cup part2(C) \times \{x_7\}.
\end{aligned}$$

Cube C_1 is equal to $D_1' \times D_2$, where D_1' is the set of the solutions from D_1 covering A_p or of the extensions (by setting variables from $var(D_1) \cap var(A_p)$) which cover A_p of solutions from D_1 not covering A_p. In fact, $x_1\overline{x}_2$ does not cover A_p and cannot be extended to cover A_p (assigning values to free variables from $var(D_1) \cap A_p$, i.e., variable x_3). Solution x_3 does not cover A_p, but it can be extended to cover A_p by assigning 0 to x_1, so that $D1' = \{x_3\overline{x}_1\}$.

Cube C_2 is equal to $D_1'' \times D_2'$, where D_1'' is the set of the solutions from D_1 not covering A_p and not extendable to cover A_p, or the extensions (by setting variables from $var(D_1) \cap var(A_p)$) which do not cover A_p of solutions from D_1 not covering A_p. So $D1'' = \{x_1\overline{x}_2, x_3x_1\}$. In fact, the first solution of D_1, $x_1\overline{x}_2$, does not cover A_p and is not extendable to cover A_p, because the extensions in the domain x_1, x_2, x_3 of $x_1\overline{x}_2$ are $x1\overline{x}_2x_3$ and $x1\overline{x}_2\overline{x}_3$ which do not cover A_p. The second solution of D_1, x_3, does not cover A_p and its extension x_3x_1 in the domain x_1, x_2, x_3 does not cover A_p either, while the extension $x_3\overline{x}_1$ does. So x_3x_1 is in D_1'.

D_2' consists of the solutions from D_2 covering A_p or extendable to cover A_p by assigning a value to variables from $var(D_2) \cap var(A_p)$. In fact, D_2 contains $\overline{x}_4$ and $\overline{x}_5$, of which $\overline{x}_5$ cannot be extended in the domain x_4, x_5 to cover A_p, while $\overline{x}_4$ can be extended to $\overline{x}_4x_5$ to cover A_p.

Cube $C_3 = D_1'' \times D_2''$ contains the solutions from D_2 not covering A_p and not extendable to cover A_p, or the extensions (by setting variables from $var(D_2) \cap var(A_p)$) which do not cover A_p of solutions from D_2 not covering A_p. D_2 contains $\overline{x}_4$ and $\overline{x}_5$. Solution $\overline{x}_5$ does not cover A_p nor any of its extensions in the domain x_4 and x_5 does, while $\overline{x}_4$ has an extension $\overline{x}_4\overline{x}_5$ which does not cover A_p and an extension $\overline{x}_4x_5$ which covers A_p. Since in D_2'' the conjunct $\overline{x}_4\overline{x}_5$ is subsumed by $\overline{x}_5$, we can remove the former, so that $var(D_2'')$ ends up as equal to $\{x_5\}$. Therefore in general after the recalculation the support of the domains of the cubes may shrink.

The additional domain $\{x_7\}$ multiplied by $part2(C)$ describes the solutions covering A_p by setting variables from $var(A_p) \setminus var(C)$.

The following rule to obtain the domains D_i' and D_i'', given the domain D_i and a row A_p, can be given.

Recomputation rule. Suppose w.l.o.g. that $A_p = d(x_1) + \cdots + d(x_p)$, where $d(x_k)$ is either x_k or $\overline{x}_k$, and that $var(D_i) \cap var(A_p)$ consists of variables $x_1, \cdots, x_r, r \leq p$. To get D_i' multiply each solution from D_i by each literal $d(x_k), k = 1, \cdots, r$. To get D_i'' multiply each solution from D_i by the product term $\overline{d(x_1)} \cdot \cdots \cdot \overline{d(x_r)}$. If a solution from D_i implies $d(x_k)$ then that solution is added to D_i'. If the result of multiplying a solution by $d(x_k)$ is empty then that solution is added to D_i''. After obtaining D_i' and D_i'', remove conjuncts subsumed by other conjuncts.

Theorem 7.1.1 *The recomputation of $Rec(A' + A_p, C)$ with the previous recomputation rule yields a collection of non-overlapping solution cubes whose union is exactly the set of all the irredundant solutions of $A' + A_p$.*

7.2 Other Applications of Incremental Problem Solving

To underline its versatility, we show how incremental problem solving can be applied to the following problems: graph coloring (GC), traveling salesman problem (TSP) and satisfiability (SAT). We do not provide ready-to-use algorithms, but only demonstrate the applicability of IPS to different problems.

Notice that when solving an optimization problem as a starting point for IPS we can always employ a "lower bound" subproblem from a traditional BAB formulation. Such lower bound subproblems are used because they are easy to solve, due to the simple and regular structure of their solution space which can be represented in a compact form.

Graph Coloring Let $G = (V, E)$ be a graph to be colored. Suppose that we need to prove that there is no n-coloring of G. A lower bound subproblem of $GC(G)$ is $GC(G')$, where G' is a complete subgraph G' of G of maximal size. Let $Col(V')$ be an assignment of colors to vertices from V'. The solution space $Sol(G')$ of $GC(G')$ is exactly the set $Perm(Col(V'))$, where the operator $Perm$ generates all $|V'|!$ permutations of $Col(V')$.

However, in G there may be several subgraphs of maximal size not intersecting each other. Denote by $G_1, \cdots, G_n$ all such complete subgraphs of maximal size, where $G_i = (V_i, E_i)$, $V_i \subset V$, $E_i \subset E$, $|V_i| = |V_j|$ and V_i does not intersect V_j, $i \neq j$, $i = 1, \cdots, n$. Obviously the choices of the G_is can be made in different ways.

The set of all minimal colorings of $G_1 \cup \cdots \cup G_n$ is exactly equal to $Perm(Col(V_1)) \times \cdots \times Perm(Col(V_n))$. So we can choose $GC(G')$, where $G' = G_1 \cup \cdots \cup G_n$, $V' = V_1 \cup \cdots \cup V_n$ and $E' = E_1 \cup \cdots \cup E_n$ as a starting problem. Then we approach $GC(G)$, by adding each time to G' a vertex v from $V \setminus V'$ and all edges $E(v)$ connecting v from $E \setminus E'$. To that effect, one can formulate rules to recalculate the solution space from $Sol(G')$ to $Sol(G'')$, where $G'' = (V'', E'')$, $V'' = V' \cup \{v\}$ and $E'' = E' \cup \{E(v)\}$. Once the solutions of the augmented graph are recomputed, the solutions with n or more colors are discarded.

Traveling Salesman Problem Let $C = \{c_1, \cdots, c_d\}$ be the set of cities and D be the distance matrix where D_{ij} specifies the distance between cities c_i and c_j. TSP is the problem to find a minimal distance tour going through all cities of C.

Suppose that we need to prove that $TSP(C, D)$ has no solution costing less than *ubound*. Denote by D' the matrix all whose elements are equal to m, where m is the minimal distance between two cities from C. $TSP(C, D')$ is a lower bound subproblem of $TSP(C, D)$. Denote by $I(C)$ an assignment of integers $1, \cdots, d$ to the cities from C specifying a tour. Then the set of minimal solutions $Sol(C, D')$ of $TSP(C, D')$ is exactly $Perm(I(C))$, because every tour has the same cost $d \cdot m$. So we can use $TSP(C, D')$ as a starting problem in the IPS paradigm.

Then we approach $TSP(C, D)$ from $TSP(C, D')$ by replacing each time an element D'_{ij} with the corresponding element D_{ij}, so that the two matrices D and D' become closer. One can formulate rules to recalculate the solution space of the modified cost matrix. After the recomputation, any solution costing more or as *ubound* is discarded.

Satisfiability We conclude with an example of IPS applied to a decision problem, SAT, i.e., satisfiability of a conjunctive normal form (CNF). Suppose that the input is a CNF $C = D_1 \cdot \cdots \cdot D_n$ of n implicates. Denote by $Lit(D_i)$ the set of literals occurring in D_i.

Let $Indep(C) = D_{i_1}, \cdots, D_{i_p}$ be a set of implicates from C of maximal size not intersecting each other, i.e., for D_i, D_j, $i \neq j$, $i_1 \leq i \leq i_p$, $i_1 \leq j \leq i_p$,, it is the case that $Lit(D_i)$ and $Lit(D_j)$ do not intersect. The set of solutions of $Sat(Indep(C))$ can be represented as $A_{i_1} \times \cdots \times A_{i_p}$, where A_{i_k} is a set of assignments satisfying implicate D_{i_k}. For example if $D_{i_k} = x_5 + \overline{x}_7$ the set of assignments satisfying D_{i_k} consists of two elements: $\{x_5 = 1, x_7 = 0\}$.

Then we approach $Sat(C)$ by adding to $Indep(C)$ implicates of C not contained in $Indep(C)$. One can formulate rules to recalculate the solution space after adding a new implicate. There will be solutions of the starting problem that cannot be extended to solutions of the augmented CNF, because of contradictory requirements on the assignments.

Chapter 8

Conclusions

We have presented 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 [1]. 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. A key technical contribution to design a search strategy which realizes negative thinking is 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 solution, thereby providing a natural clustering of the recomputed solutions. As argued in this dissertation, 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 lower bounds are obtained by means of an independent set of rows, and branches are 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 with this matrix, or we produce at least one 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 MIN-COV_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 6.1, but it has a less effective pruning strategy in the examples of Tables 6.2 and 6.3, 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. Also of great interest is the extension of our algorithm to the binate covering problem and to other exact search problems.

A more basic line of research is the exploration of data structures different from cubes of solutions, but still enjoying their nice properties, since the latter are just the simplest way of representing sets of partial solutions. We believe that studying various ways of implicitly representing sets of solutions is a promising direction of investigation to rescue branch-and-bound from its current limits.

References

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

[2] T. Kam, T. Villa, R. Brayton, and A. Sangiovanni-Vincentelli, *Synthesis of FSMs: Functional Optimization*. Kluwer Academic Publishers, 1996.

[3] R. Rudell, "Espresso," *Computer Program*, 1987.

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

[5] 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*, pp. 91–98, IEEE, Nov. 1997.

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

[7] 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.

[8] O. Coudert, "On solving binate covering problems," in *The Proceedings of the Design Automation Conference*, pp. 197–202, June 1996.

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

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

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

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

[13] R. Brayton and F. Somenzi, "An exact minimizer for Boolean relations," in *The Proceedings of the International Conference on Computer-Aided Design*, pp. 316–319, Nov. 1989.

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

[15] B. Lin and F. Somenzi, "Minimization of symbolic relations," in *The Proceedings of the International Conference on Computer-Aided Design*, pp. 88–91, Nov. 1990.

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

[17] S. D. Sarkar, A. Basu, and A. Choudhury, "Simplification of incompletely specified flow tables with the help of prime closed sets," *IEEE Transactions on Computers*, pp. 953–956, Oct. 1969.