McMaster University

Advanced Optimization Laboratory

Title:

Efficient Preprocessing for VLSI Optimization Problems

Authors:

W. L. Hare, M. J. J. Liu, and T. Terlaky

AdvOl-Report No. 2006/10

August 2006, Hamilton, Ontario, Canada

Efficient Preprocessing for VLSI Optimization Problems

W. L. Hare [*] M. J. J. Liu [†] T. Terlaky [‡] [§]

August 22, 2006

Abstract

Advances in technology for the manufacturing of integrated circuits have resulted in extremely large, and time consuming, problems on how to lay out components for optimal circuit performance. Although these problems can be written as a linear program, due to the very high number of variables involved in solving these problems, it is of upmost importance to preprocess these problems both efficiently and well before attempting their solution. Fortunately, VLSI problems have a remarkably well structured constraint matrix, which allows for some novel preprocessing techniques. In this report we examine efficient preprocessing for the linear programs resulting from VLSI circuit design. We provide analysis showing our preprocessing techniques are accurate and more efficient than traditional preprocessing techniques, as well as provide some numerical testing demonstrating the increased efficiency.

Keywords VLSI programming, Linear Programming, Preprocessing

1 Introduction

Recent advances in technology for manufacturing integrated circuits has allowed for the production of circuit with millions of components. This has lead to extremely large, and time consuming, problems on how to layout components for optimal circuit performance. In general the *very large scale integration circuit design problem (VLSI problem)* is modelled as a sequence of very large discrete optimization problems (see [She99] and references therein).

One method of modelling VLSI problems begins by writing the circuit as a graph, i.e. a collection of nodes and edges. Each component of the circuit is placed at one of the nodes of the graph. Each edge now represents a possible route between two nodes, i.e a path along which a wire can be laid. The problem is now dissected into a series of smaller subset problems, each of which is designed to preform a specific propose (joining a collection of nodes for example). For each subset a set of trees is created to solve the subproblem. The VLSI problem now becomes

[*] `whare@cecm.sfu.ca`, research supported by MITACS and NSERC Discovery Grant #5-48923

[†] `jessiel@optlab.mcmaster.ca`, research supported by NSERC Discovery Grant #5-48923

[‡] `terlaky@mcmaster.ca`, research supported by MITACS, NSERC Discovery Grant #5-48923, and the Canada Research Chair Program

[§] all authors: Dept of Comp & Soft, McMaster University, Hamilton, ON, L8N 3Z5, Canada

a matter of selecting exactly one tree from each subset in a manner which somehow optimizes performance [FE97] [AVY06]. This can be written as a linear program as follows:

Let the routing problem be subdivided t subproblems. For subproblem i, we have a collection of trees, $\mathbf{t}_1^i, \mathbf{t}_2^i, ...\mathbf{t}_{m_i}^i$ and a $\{0,1\}$ vector $x^i \in \{0,1\}^{m_i}$. In this way $x_j^i = 1$ if tree $\mathbf{t}_j^i$ is used and $x_j^i = 0$ if tree $\mathbf{t}_j^i$ is not used. Mathematically, this yields the constraints,

$$\sum_{j=1}^{m_i} x_j^i = 1, \quad x_j^i \in \{0,1\}. \tag{1}$$

Let $n = \sum_{i=1}^{t} m_i$ and T be the $t \times n$ matrix

$$T = \begin{bmatrix} \mathbf{1}_{m_1}^\top & 0 & 0 & \dots & 0 \\ 0 & \mathbf{1}_{m_2}^\top & 0 & \dots & 0 \\ \vdots & & & \ddots & \\ 0 & 0 & 0 & \dots & \mathbf{1}_{m_t}^\top \end{bmatrix},$$

where $\mathbf{1}_k$ represents the column vector of length k consisting of all 1s (we shall occasionally drop the subscript when the length of $\mathbf{1}$ is obvious). Then constraint (1) reduces to

$$Tx = \mathbf{1}_t, \quad x \in \{0,1\}^n.$$

Let the graph representing the circuit layout have p edges. We next create a $p \times n$ matrix P, such that each column represents a given tree and each row represents the edges used by that tree. For example if column 3 was $[0,1,1,0,1,0,0...0]^\top$ then the tree represented by x_3 would use edges 2, 3, and 5. The vector Px is then a list of the edge congestions for the given selection x. Finally we let r represent the maximal edge congestion, $r = \max Px$, and M be a (given) maximal edge capacity. Thus we have the following constraint system:

$$\left.\begin{array}{rcl} Tx & = & \mathbf{1}_t \\ r & \geq & \max Px \\ r & \leq & M \\ x & \in & \{0,1\}^n \end{array}\right\} \Leftrightarrow \left\{\begin{array}{rcl} Tx & = & \mathbf{1}_t \\ Px - r\mathbf{1}_p & \leq & 0 \\ r & \leq & M \\ x & \in & \{0,1\}^n. \end{array}\right.$$

Our objective might be to minimize the total wire length ($\sum \alpha_j^i x_j^i$, where α_j^i is the wire length of tree $\mathbf{t}_j^i$), minimize the total number of bends ($\sum \beta_j^i x_j^i$, where β_j^i is the number of bends in tree $\mathbf{t}_j^i$), minimize the maximal congestion (set $M = \infty$ and use r as the objective function), or any combination of the above ($\omega_1 \sum \alpha_j^i x_j^i + \omega_2 \sum \beta_j^i x_j^i + \omega_3 r$, where ω_1, ω_2 and ω_3 are weighting factors).

In order to solve this problem we often relax the integer constraint ($x \in \{0,1\}^n$) to create

the linear program

$$\min \left\{ \omega_1 \alpha^\top x + \omega_2 \beta^\top x + \omega_3 r \ : \ \begin{array}{c} \begin{bmatrix} T & 0 & 0 \\ P & -\mathbf{1}_p & I \end{bmatrix} \begin{bmatrix} x \\ r \\ s \end{bmatrix} = \begin{bmatrix} \mathbf{1}_t \\ 0 \end{bmatrix}, \\ x, r, s \geq 0, \quad x \leq 1, \quad r \leq M \end{array} \right\}. \tag{2}$$

where α is a vector listing the wire length for each tree, β a vector listing the number of bends for each tree, I is the identity matrix and s are slack variables.

Of course, the question of how to dissect the total VLSI problem into a collection of trees from which to select an optimal solution is not a trivial one. However, this is not the focus of this work, and instead we refer curious readers to [She99]. The main goal of this report is to describe how the structure of the subproblem can be exploited in order to create a highly efficient preprocessing algorithm.

Remark 1.1 *It should be noted that although the preprocessing techniques discussed in this work were inspired by the VLSI problem, the techniques developed within this paper are clearly applicable to preprocessing any problem where the constraint matrix has structures similar to that of the T or P matrices of the VLSI problem. Many combinatorial selection problem take a suitable form.*

The remainder of this paper is organized as follows. In Section 2 we survey the previous preprocessing techniques for linear problems. In Section 3 we develop our new techniques for preprocessing the VLSI problem. We split this section into several parts including: Subsection 3.1 which deals with the preprocessing of the tree selection matrix, Subsection 3.2 which deals with the preprocessing of the tree description matrix, and Subsection 3.4 which examines the complexity for the new preprocessing techniques. In Section 4 we provide some computational experience of the effect preprocessing has the solving time for the VLSI problem using interior point methods.

2 Classical Preprocessing

We will assume the the user is given a minimization problem

$$\min\{c^\top x : Ax = b, x \geq 0\}.$$

(In the case of the VLSI problem, $A = \begin{bmatrix} T & 0 & 0 \\ P & -\mathbf{1}_p & I \end{bmatrix}$.) The classical preprocessing procedure for such a problem requires a number of steps. Although many other preprocessing techniques exists, the most common techniques are listed below (see [Hua04] [Mar03] [JNS00] and references therein).

1. remove empty rows of A (confirm feasibility),
2. remove empty columns of A,

3. remove singleton rows of A and update the corresponding column,
4. remove duplicate rows of A (confirm feasibility),
5. remove duplicate columns of A,
6. remove redundant constraints,
7. tighten upper and lower bounds, and
8. improve the sparsity of A.

Since each of these techniques may change the overall structure of the matrix, they are often repeated multiple times. Of course during each step one must also be careful to update the vectors b and c appropriately, as well as keep track of any changes to the objective vector x.

The reasons behind and execution of steps 1 to 5 are fairly transparent, all that needs to be stated is that a row (or column) is considered a *duplicate* of another row (respectively column) if they form a linearly dependant pair; e.g., $\text{row}_i = \lambda \text{row}_j$ for some λ. Step 7 consists of determining the accuracy of any upper or lower bounds given for the problem and sharpening them if possible. Steps 6 and 8 could be enlightened by a brief discussion.

In step 6, remove redundant constraints, the goal is to find rows of A which create similar inequalities. In general this occurs when the rows are duplicate rows expect for a single *slack* variable, a slack variable being a singleton column. For example

$$\left.\begin{array}{rcl} x_1 + 2x_2 - x_3 + x_4 & = & 5 \\ -2x_1 - 4x_2 + 2x_3 + x_5 & = & 3 \\ x_i & \geq & 0 \end{array}\right\} \Leftrightarrow \left\{\begin{array}{rcl} x_1 + 2x_2 - x_3 & \leq & 5 \\ 2x_1 + 4x_2 - 2x_3 & \leq & 3 \\ x_i & \geq & 0. \end{array}\right.$$

As the second inequality reduces to $x_1 + 2x_2 - x_3 \leq 3/2$, the first inequality is redundant and can therefore be removed. If one knows exactly which variable are slack variables step 6 can easily be incorporated into step 4. Otherwise, step 6 can be quite difficult and time consuming.

In step 7, improve the sparsity of A, the goal is to decrease the number of nonzero entries in the matrix A. In general, this is done by locating two rows, which we shall denote *sb* and *sp*, such that the support of row *sb* forms a subset of the support of *sp* (recall the support of a vector is the location of its nonzero elements). The row *sb* can then be used to cancel some of the nonzero terms from *sp*. The key problem to reduce the sparsity of A is therefore how to choose the subset row *sb* and the superset row *sp*. Determining these rows can be an extremely time consuming process.

3 Preprocessing VLSI programs

Having discussed classical, generic, preprocessing techniques, we turn our attention to how these techniques can be refined in the question of preprocessing VLSI problems. The first step in preprocessing this problem is to extract from the matrix $A = \begin{bmatrix} T & 0 & 0 \\ P & -\mathbf{1}_p & I \end{bmatrix}$ the matrices T and P. This can be accomplished in several manners. For example, by searching the final row for the value -1 and using this to compute the sizes (and therefore locations) of the matrices T and P.

Lemma 3.1 *Let* $A = \begin{bmatrix} T & 0 & 0 \\ P & -\mathbf{1} & I \end{bmatrix}$ *be the* $M \times N$ *constraint matrix of a VLSI problem. Suppose* T *be* $t \times n$ *and* P *be* $p \times n$*; so* $-\mathbf{1}$ *is* $p \times 1$*, and* I *is* $p \times p$*. If* $A_{M,k} = -1$ *then*

$$t = M - N + k, \quad p = M - p \quad \text{and} \quad n = k - 1.$$

Proof: Clearly $M = t + p$ and $N = n + 1 + p$. Moreover, k is unique and equal to $n + 1$. Therefore $p = N - k$ and $t = M - p = M - N + k$. □

Obviously, there are many other methods for computing the sizes of T and P (locating the first 0 in the right hand side vector $[\mathbf{1}_t^\top \quad 0]^\top$ for example). We leave alternate approaches as exercises for the inquisitive reader. The advantage of the approach in Lemma 3.1 is that our preprocessing algorithm will not require the input of anything other than the constraint matrix A.

After extracting the matrices T and P from the VLSI problem, we can rewrite the problem in the form

$$\min\{r : Tx = \mathbf{1}_t,\ Px - r\mathbf{1}_p \leq 0,\ x, r \geq 0,\ r \leq M\}.$$

By doing this, it becomes clear that in our preprocessing we need only concern ourself with the processing of matrices T and P. This immediately reduces the amount of work required, as we are effectively able to ignore all of the columns of the matrix A which appear after the matrix P. In examining the preprocessing of T and P more thoroughly, we shall see that this reformulation allows for several other improvements on classical preprocessing techniques. We begin with the preprocessing of T.

3.1 Preprocessing T

Recall the matrix T takes the form

$$T = \begin{bmatrix} \mathbf{1}_{m_1}^\top & 0 & 0 & \dots & 0 \\ 0 & \mathbf{1}_{m_2}^\top & 0 & \dots & 0 \\ \vdots & & & \ddots & \\ 0 & 0 & 0 & \dots & \mathbf{1}_{m_t}^\top \end{bmatrix}. \tag{3}$$

Due to the structure of T it is clear that there are no empty rows or columns in T, nor are there any duplicate rows of T. Although T has many duplicate columns, we cannot remove them because it is unlikely they will remain duplicate columns once the matrix P is reattached. Since the are no slack variables associated with T, there cannot be any redundant constraints. Finally, the structure of T makes it clear that no sparsity improvement can be achieved. The preprocessing of T therefore reduces to determining which rows of i have exactly one value, i.e. when does $m_i = 1$. Algorithm 3.1 provides Pseudo code to accomplish this.

Examining the pseudo code for this technique it is clear that the singleton row search for T requires the examination of exactly $n - 1$ locations in T. Furthermore, by the arguments above, this provides all the preprocessing required for the matrix T.

Algorithm 3.1 : **[Preprocessing T]**

```
INPUT: T OF SIZE t × n AND FORMAT (3)
INITIALIZE i = 1, ct = 1;
FOR j = 2, 3, ...n
   IF T_{i,j} = 1
      SET ct = ct + 1
   ELSE (T_{i,j} = 0)
      IF ct = 1 (ROW i IS A SINGLETON)
         MARK ROW i IN T AND COLUMN j − 1 IN T AND P FOR REMOVAL
         STORE x_{j−1} = 1
      END IF
      SET i = i + 1 AND ct = 1
   END IF
END FOR
```

3.2 Preprocessing P

Before preprocessing P we assume that the algorithm in Algorithm 3.1 has already been preformed. Furthermore, we assume that the rows and columns marked for removal have been removed from T and P, and b has been updated appropriately. As such, any empty column in P is a result of a preexistent blank column in P. We now consider the system of equations generated by the bottom portion of A,

$$[P \quad -\mathbf{1}_p \quad I] \begin{bmatrix} x \\ r \\ s \end{bmatrix} = [b'] \quad \Leftrightarrow \quad r \geq Px - b',$$

where b' is the vector resulting from the preprocessing of T.

Remark 3.2 *It is worth taking a moment to discuss the structure of the vector b'. If the matrix T is reduced to a $t' \times n'$ matrix T', it is clear that the first t' rows of b' will still be 1. For the remaining rows, since a removed column of P corresponds to setting a variable to 1 (due to a singleton row in T) is it easy to see that component i of vector b' is equal to $-\sum_{j \in RC} P_{i,j}$, where RC is the list of removed columns.*

Recall that each column of the sparse $p \times n$ matrix P represent the edges used by the trees corresponding to that column. As such, unless there is a grievous error in matrix creation, it is impossible for P to have empty columns. Similarly if P contains duplicate columns, then there are two trees which are exactly the same. Although possible, this would again represent an error in problem generation. Therefore we ignore the examination of columns until after we have completed a duplicate constraint search.

Examining the preprocessing of T we note that whenever a row is removed, its corresponding column $(j-1)$ sets $x_{j-1} = 1$. When substituting $x_{j-1} = 1$ in the matrix P, the right hand side vector (originally all 0s) will take on negative integer values; $b' \leq 0$. Recalling that P is a 0-1 matrix and $x \geq 0$ allows us to apply the following lemma.

Lemma 3.3 *Let P be a nonnegative $p \times n$ matrix ($P_{i,j} \geq 0$ for all i, j) and let b' be a $p \times 1$ vector. If x is a nonnegative $n \times 1$ vector and $r\mathbf{1}_p \geq Px - b'$ then*

$$r \geq -\min b'.$$

Proof: Since P and x are nonnegative we have

$$r\mathbf{1}_p \;\geq\; Px - b' \;\geq\; P0 - b' \;=\; -b'.$$

Therefore $r \geq \max\{-b'\} = -\min b'$. □

When applied to the VLSI path congestion variable, Lemma 3.3 provides us with a rapid way to compute a lower bound for the path congestion: $-\min b'$. We now use that lower bounded to quickly determine which rows of P must be redundant. To do this we let $\bar{r} = -\min b'$, and recall that $x \leq 1$ (as $Tx = \mathbf{1}_t$, and $x \geq 0$). Considering the vector $P\mathbf{1}_n - b'$, it is clear that any row i for which $(P\mathbf{1}_n - b')_i \leq \bar{r}$ is a redundant constraint for the problem and can be removed. Algorithm 3.2 provides pseudo code to this effect.

Algorithm 3.2 : **[Redundant constraint removal for P]**

```
INPUT: 0-1 MATRIX P OF SIZE p × n AND VECTOR b' OF SIZE p × 1
COMPUTE THE VALUE r̄ = − min b'
COMPUTE VECTOR s̄ = P1_n − b'
FOR i = 1, 2, ...p
    IF s̄_i ≤ r̄
        REMOVE ROW i
    END IF
END FOR
ADD CONSTRAINT r ≥ r̄
```

Since the matrix P is a sparse 0-1 matrix, the second computation, $\bar{s} = P\mathbf{1}_n - b'$, can actually be preformed quite quickly by simply counting the number of nonzero constraints in each row of P, amounting to the examination of the total number of nonzero entries in P. The for loop then requires an additional p comparisons.

As mentioned, Lemma 3.3 gives us a rapid tool to compute a lower bound for the VLSI path congestion. Algorithm 3.2 then uses this lower bound to determine which rows of P are redundant. This knowledge is useful in three manners. First, if a fixed lower bound for the maximal value for the path congestion is given, this lower bound can immediately be compared to the lower bound developed here to determine if the partitioning of trees created is feasible. Second, if the path congestion is unrestricted, but a good lower bound for the optimal value for the path congestion in question is available, one could replace the computation of $\bar{r}$ with an input line. If $\bar{r}$ is selected well, this can further reduce the size of the problem. Finally, if the path congestion is unrestricted and no lower bound for path congestion is available, then the calculated $\bar{r}$ immediately provides one. In the five test cases considered in Section 4 of this work, this lower bound appears to be an excellent lower bound for the *relaxed* VLSI path congestion (see Table 1 below). Nonetheless, there is still considerable room for improvement

when compared to the best known *integer* VLSI path congestion (i.e. the VLSI problem with the objective set to $\min r$).

Table 1: Accuracy of $-\min b'$ as a lower bound

Test	$-\min b'$	correct solution to the relaxed problem	best known integer solution*
1	10	13.333	15
2	9	11.140	15
3	22	28.586	32
4	8	10.640	13
5	9	9.652	13

* from [AVY06]

3.3 Improving Matrix Sparsity

As a final, optional, step in preprocessing the VLSI problem, we consider improving the sparsity of the matrix A. We begin, as usual, by dividing A into two parts: $[T\ 0\ 0]$ and $[P\ \ -\mathbf{1}\ I]$. Suppose, for the sake of eventual contradiction, that the superset row *sp* was found in the first part, $[T\ 0\ 0]$. Then the right hand columns of row *sp* must all be 0. Therefore, the subset row *sb* cannot be in the matrix $[P\ \ -1\ I]$ as its right hand columns all have nonzero entries. Furthermore, the subset row cannot be in the matrix $[T\ 0\ 0]$, as each column of the T contains exactly one nonzero entry. Therefore we know that the superset row must be in the second part, $[P\ \ -\mathbf{1}\ I]$.

Given a superset row in $[P\ \ -\mathbf{1}\ I]$, the subset row cannot also lie in $[P\ \ -\mathbf{1}\ I]$. To see this note that the matrix I corresponds to the slack variables, and consider the equation $Px \leq r$. Since P is a 0-1 matrix the cancellation of nonzero elements in Px would require subtraction between two rows of P. However this is impossible as all the inequalities are the same direction. Thus row *sb* is in the first part, $[T\ 0\ 0]$.

Given this information, it is clear that to improve the sparsity of A one preforms an exhaustive comparison of the rows of $[T\ 0\ 0]$ to the rows of $[P\ \ -\mathbf{1}\ I]$. Although this produces a minor improvement to general sparsity improvement subroutine, it is quite time consuming.

3.4 Complexity Comparison

Before examining the complexity of the preprocessing techniques described here, we provide a pseudo code for our entire preprocessing method, Algorithm 3.4. Notice that simply stringing together Algorithms 3.1 and 3.2 does not complete the preprocessing of the problem. As the removal of redundant constraints may result in the creation of duplicate columns in the final matrix, further preprocessing to remove these columns is applied. Optional preprocessing to improve the sparsity of A may also be applied. Since both of these steps are dealt in the standard manner we do not include detailed pseudo code here.

Note the creation of b' in Algorithm 3.4 is accomplished in accordance to Remark 3.2.

Algorithm 3.4 : **[Preprocessing VLSI problems]**

INPUT: A OF SIZE $M \times N$ FOR A VLSI PROBLEM
DETERMINE T AND P
  FIND k SUCH THAT $A_{M,k} = -1$
  SET $t = M - N + k$, $p = M - p$ AND $n = k - 1$
  SET T AS THE $t \times n$ MATRIX IN THE UPPER LEFT PORTION OF A
  SET P AS THE $p \times n$ MATRIX IN THE BOTTOM LEFT PORTION OF A
PREPROCESS T
  PREFORM ALGORITHM 3.1 ON T
  REMOVE ROWS AND COLUMNS OF T TO CREATE T'
PREPROCESS P
  REMOVE COLUMNS OF P TO CREATE P' AND b'
  PREFORM ALGORITHM 3.2 ON P' AND b' TO CREATE P'' AND b''
FINAL PREPROCESSING
  REMOVE DUPLICATE COLUMNS OF $\begin{matrix} T' \\ P'' \end{matrix}$ TO CREATE P''' AND T''
  *IMPROVE THE SPARSITY OF $\begin{matrix} T'' \\ P''' \end{matrix}$ TO CREATE P'''' AND T''' (*OPTIONAL)
  OUTPUT $\widehat{A} = \begin{matrix} T'' & 0 & 0 \\ P''' & -\mathbf{1} & I \end{matrix}$ AND $\hat{b} = \begin{matrix} \mathbf{1} \\ b'' \end{matrix}$

From the arguments presented in Subsection 3.1 and 3.2 it is clear that Algorithm 3.4 provides a valid preprocessing of the VLSI problem. However, Algorithm 3.4 does not necessarily fully preprocess the problem. For example, Algorithm 3.4 only loosely approaches the problem of tightening upper and lower bounds. It is immediately clear that the path selection variables (x) are bounded below by 0 and above by 1. The creation of $\bar{r}$ in Subroutine 3.2 provide a quick lower bound for the path congestion variable, but the upper bound is unexamined. It is probable that further preprocessing tighter bounds can be obtained. Furthermore, it is also possible that row removals result in new singleton columns, column removals result in new singleton rows, or sparsity improvement results in the creation of new duplicate rows or columns. As such making a second pass using the classical preprocessing techniques discussed in Section 2 could further improve the structure of this problem. In our numerical testing (Section 4) we leave tightening bounds, and repeated passes of classical preprocessors to the individual optimization solvers.

In comparison with classical preprocessing techniques we have made several major time saving alterations. First, by splitting the constraint matrix A into its T and P components, we have effectively ignored the right hand side of the matrix. Second, we have dealt with the tree selection matrix T in a manner, which examines the absolute minimum number of matrix locations in order to preprocess T. Third, we have developed a rapid and elegant manner to determine the location of redundant constraints in the tree description matrix P. This reduced the usually quite difficult problem of locating redundant constraints to a simple sparse matrix multiplication followed by a simple vector comparison. In Section 4 we show numerically the effect these new preprocessing techniques have on preprocessing and solving the VLSI problem. But first we give a brief discussion on postprocessing techniques.

3.5 Postprocessing and Projecting onto Integer Solutions

After preprocessing and solving the reduced problem, we of course must postprocess in order to determine the solution to the unprocessed problem. Usually this consists of reinserting all removed values of the solution. However, in the case of the VLSI problem we are only concerned with the path selection component of the solution vector (i.e. the x in $[x \quad r \quad s]^\top$). Thus our postprocessing consists of reinserting any removed values from the x component of the solution, while we can safely ignore reinserting any removed values from the s component of the solution. This process is extremely rapid regardless of problem size (even for out largest tests this requires less than 10^{-3} seconds).

After solving (and postprocessing) the relaxed VLSI problem, one is still left with the question of how to obtain an integer solution. Many methods exist for "projecting" the relaxed solution onto an integer solution, we refer interested readers to [She99].

4 Numerical Results

We test our preprocessing techniques by solving five sample VLSI problems using four different solvers. By applying different settings to these solvers, we effectively solve the sample problems in seven different manners.

4.1 Test Problems and Effects of Preprocessing

Our five test problems come directly from industry and constitute an appropriate test set for our preprocessing techniques. These problems correspond to the final subproblems generated in solving `prim2, Ind2, Ind3, avq.small` and `avq.large` of [AVY06]. However, the reader should be aware that our testing does not represent solve times for these problems, only solve times for the linear programs generated by these problems. In order to avoid potential confusion, and for the sake of simplicity, we shall label the problems as 1, 2, 3, 4, and 5. In Table 2 we provide the size and density of the original problem ("none"), the size and density of the problem resulting from preprocessing without sparsity improvement ("basic"), and the size and density of the problem resulting from preprocessing with sparsity improvement ("full") as well as the time used to preform each preprocessing procedure. In all cases the postprocessing time was negligible (less than 10^{-3} seconds), so we do not report this. The "ratio to none" column gives the percentage ratio of the basic preprocessed problem to the unprocessed problem, the "ratio to basic" column gives the percentage ratio of the full preprocessed problem to the basic preprocessed problem. The number "nz" is the number of non-zero elements in A.

In examining Table 2, one can immediately see that the full preprocessing has very little effect over that of the basic preprocessing. Since the full preprocessing step is quite time consuming, we recommend against the procedure. However, it should be noted that both preprocessors were written in uncompiled MATLAB© code. Rewriting in a compiled language would decrease the preprocessing time significantly.

Table 2: Effect of Preprocessing on A

Test	None	Basic	Ratio to none	Full	Ratio to basic
1 (`prim2`)	0 secs 3447×6570 nz=45310 (0.20% dense)	0.4 secs 1669×4792 nz=35253 (0.44% dense)	$48.4\% \times 72.9\%$ 77.8%	16.1 secs 1669×4792 nz=31493 (0.39% dense)	89.3%
2 (`Ind2`)	0 secs 21054×52605 nz=666150 (0.06% dense)	9.3 secs 11945×43496 nz=602483 (0.12% dense)	$56.7\% \times 82.7\%$ 90.4%	2287.1 secs 13329×44880 nz=583545 (0.11% dense)	96.9%
3 (`Ind3`)	0 secs 23977×74961 nz=824611 (0.05% dense)	11.7 secs 12756×63740 nz=749719 (0.09% dense)	$53.2\% \times 85.0\%$ 90.9%	4444.4 secs 13481×64465 nz=727564 (0.09% dense)	97.0%
4 (`avq.small`)	0 secs 29609×81736 nz=1003464 (0.04% dense)	16.4 secs 18110×70237 nz=949557 (0.07% dense)	$61.2\% \times 85.9\%$ 94.6%	4844.2 secs 18985×71112 nz=937612 (0.07% dense)	98.7%
5 (`avq.large`)	0 secs 33630×78690 nz=909980 (0.03% dense)	18.5 secs 19698×64758 nz=839862 (0.07% dense)	$58.6\% \times 82.3\%$ 92.3%	4935.9 secs 20819×65879 nz=825769 (0.06% dense)	98.3%

4.2 Solvers and Run Time Results

To test the change in solving time required for the unprocessed and preprocessed problems we consider four different optimization codes: `SeDuMi`, `linprog`, `LipSol` and `CPlex`.

The `SeDuMi` solver is a MATLAB implemented interior point method for semi-definite programming. Information on `SeDuMi` can be found in [Stu99], and a free download is available at `sedumi.mcmaster.ca`. Our test results using `SeDuMi` are found in Table 5 of Appendix A.

The `linprog` linear program solver is a part of the MATLAB Optimization Toolbox. Details on `linprog` can be found on the MATLAB webpage: `www.mathworks.com/access/helpdesk/help/toolbox/optim/ug/linprog.html`, while our test results are found in Table 6 of Appendix A.

The `LipSol` solver is a Matlab-based package for solving linear programs by interior-Point methods. It makes use of several Fortran subroutines for computational efficiency. More information on `LipSol` can be found in [Zha99], while our test results are found in Table 7 of Appendix A.

The `CPlex` solver is a commercial product produced by ILOG. All tests were run using `CPlex` version 10. Further information can be found at `www.ilog.com`. The `CPlex` package comes equipped with several different optimization algorithms. The default algorithm for linear

programming is the dual simplex method, which is known to be efficient in practice. `CPlex` also comes equipped with an interior point solver, which can be accessed by setting `lpmethod = 'barrier'`. We examine this solver, labelling the tests with $\texttt{CPlex}_{\texttt{ipm}}$. Since the `CPlex` solver allows the user to turn off the `CPlex` internal preprocessing subroutines, we considered `CPlex` both using both the default mode (`presolve = 'on'`), and with internal preprocessing procedure turned off (`presolve = 'off'`). We label these $\texttt{CPlex}^{\texttt{on}}$ and $\texttt{CPlex}^{\texttt{off}}$ respectively ($\texttt{CPlex}^{\texttt{on}}_{\texttt{ipm}}$ and $\texttt{CPlex}^{\texttt{off}}_{\texttt{ipm}}$ for the interior point solver). Test results for `CPlex` can be found in Tables 8, 9, 10 and 11 of Appendix A.

All test were done on a 4x Pentium 3 700MHz cluster with 1GB RAM per node. As mentioned, tables detailing the run times for each of these solvers are found in Appendix A. Each table provides the times in seconds required for preprocessing ("pre- time"), running the solver ("solve time"), and total time used ("total time" = pre- time + solve time). In addition to this, we also calculate the ratio of the solve time used for a given problem and preprocessor to the solve time used for the same problem with no preprocessing ("solve time ratio"). Finally we calculate the ratio of the total time used for a given problem and preprocessor to the total time used for the same problem with no preprocessing ("total time ratio"). In the `CPlex` tables, Tables 8 and 9, these ratios are given with respect to the solve time and total time of the `CPlex` solver under its default settings (lines marked 'none' in Table 8), while in the `CPlex` interior point method tables, Tables 10 and 11, the ratios are given with respect to the solve time and total time of the $\texttt{CPlex}_{\texttt{ipm}}$ solver under its default settings (lines marked 'none' in Table 10).

4.3 Conclusions and Discussion

Appendix A contains a plethora of information which is somewhat difficult to sift through. In order to summarize these results into a usable form we include two tables here.

Table 3 gives, for each solver, the total solver time required to solve all five problems with no preprocessing ("no pre-solve time"), basic preprocessor ("basic pre-solve time") and using the full preprocessor ("full pre-solve time"). It also provides the total time required for basic and full preprocessing including solver and preprocessing time ("basic pre-total time" and "full pre-total time" respectively). Table 4 summarizes the best approach for each solver and provides the "Improvement", calculated via

$$\frac{\text{Default time} - \text{Best time}}{\text{Default time}} \times 100\%.$$

In both tables we omit the `linprog` results, as `linprog` returned an "out of memory" error on the majority of the test problems.

Examining Tables 3 and 4 (along with the tables in Appendix A) we are able to draw several conclusions.

First, it is clear that the basic preprocessing procedure described in this work have a profound impact on run time for this test set. Of course the overall improvement depends on the solver used. In the best case we see that `linprog` was able to solve test problem 2 only after the basic preprocessing was applied (see Table 6). Curiously, using the full preprocessing caused

Table 3: Running times required to solve all problems (seconds)

Solver	No presolve time	Basic presolve time	Basic pre-total time	Full presolve time	Full pre-total time
SeDuMi	103062	55541	55577	49493	66021
LipSol	247603	195407	195443	199676	216204
$\texttt{CPlex}^{\texttt{on}}$	22830	15522	15558	13694	30222
$\texttt{CPlex}^{\texttt{off}}$	48231	31146	31182	32234	48762
$\texttt{CPlex}^{\texttt{on}}_{\texttt{ipm}}$	5238	4432	4468	4237	20765
$\texttt{CPlex}^{\texttt{off}}_{\texttt{ipm}}$	5097	4389	4425	4224	20752

Table 4: Comparison of summed run times

Solver	Default solve time	Best solve time	Improvement	Best total run time	Improvement
SeDuMi	103062	full, 49493	52 %	basic, 55577	46 %
LipSol	247603	basic, 195407	21 %	basic, 195443	21 %
$\texttt{CPlex}^{\texttt{on}}$	22830	full, 13694	40 %	basic, 15558	32 %
$\texttt{CPlex}^{\texttt{off}}$	48231	basic, 31146	35 %	basic, 31182	35 %
$\texttt{CPlex}^{\texttt{on}}_{\texttt{ipm}}$	5238	full, 4237	19 %	basic, 4468	15 %
$\texttt{CPlex}^{\texttt{off}}_{\texttt{ipm}}$	5097	full, 4224	17 %	basic, 4425	13 %

`linprog` to revert to returning an out of memory error. Even in the worst case, the commercial software `CPlex`, we see overall improvements of 15 to 35 percent. In addition to this, the time required to run the basic preprocessing algorithm is quite negligible in comparison to the total run time require to solve the problem. As a result, in all cases the best total runtime was obtained by using the basic preprocessor before solving.

Second, although in general the sparsity improvement created by the full preprocessing algorithm yields a slightly faster solve time, when the time required to run this preprocessing is taken into account the total solve time is significantly higher. This might be corrected by coding the preprocessor in a compiled language. The exceptions to this are `LipSol` and $\texttt{CPlex}^{\texttt{off}}$ where the basic preprocessor resulted in solve times which were slightly faster (about 2 to 3 %) than the solve times after using the full preprocessor.

Another interesting note is that, although `CPlex`'s internal preprocessing routines are quite helpful while using `CPlex`'s default dual simplex method, they appear to have little effect while using `CPlex`'s interior point method. In fact, in our test set, the total run time for `CPlex`'s interior point method was slightly improved when `CPlex`'s preprocessing routines were turned off. One possible explanation for this is that the `CPlex`'s internal preprocessor includes subroutines which locate the equality and inequality constraints of the problem. Such subroutines would allow `CPlex` to run a more efficient version of the dual simplex algorithm, but not of the interior point method. Of course, as `CPlex` is commercial software we do not have access to the source

code and therefore cannot confirm (or deny) this hypothesis.

In terms of particular solvers, it is clear that `linprog` is a poor choice for this problem. Not only is its run time significantly higher than the remaining solvers, it failed to solve the majority of the test problems. The best solver appears to be `CPlex` running its interior point method. Combined with basic preprocessing $\texttt{CPlex}_{\texttt{ipm}}$ requires approximately 1.5 hours to complete the problem set and is over 3 times faster than using `CPlex`'s default settings. Considering the size of these test problems, it is surprising that the interior point algorithm is not `CPlex`'s default algorithm. The noncommercial solvers `SeDuMi` and `LipSol` take, respectively, approximately 15 hours and 55 hours to solve the complete problem set after basic preprocessing; a respectable performance considering their open source MATLAB based design and `SeDuMi`'s broader scope of applicability.

On a final note, during the computational testing done in this work, some experimentation with older versions of `CPlex` (versions 7 and 8) was also preformed. `CPlex` version 7 was unable to solve any but the smallest test problem considered in this paper, regardless of the preprocessing technique used. Tests for `CPlex` version 8 were run on a 900MHz Sun Blade 2000 workstation with 1GB RAM, so cannot be directly compared to our results for version 10. However on that machine, `CPlex` version 8 solved all five problems and had similar Solve Time Ratios as those found in Tables 8 and 10; the total run time on the (faster) Sun Blade workstation was approximately 8.5 hours after basic preprocessing. (`CPlex` version 9 was unavailable for testing.)

References

[AVY06] S. Areibi, A. Vannelli, and Z. Yang. An ILP based hierarchical global routing approach for VLSI ASIC design. Submitted to*: Opt. Letters*, 2006.

[FE97] A. Fernández and K. Efe. Efficient VLSI layouts for homogeneous product networks. *IEEE Trans. Comput.*, 46(10):1070–1082, 1997.

[Hua04] X. Huang. Preprocessing and postprocessing in linear optimization. M.Sc. Thesis, Department of Computing and Software, McMaster University, 2004.

[JNS00] E. L. Johnson, G. L. Nemhauser, and M. W. P. Savelsbergh. Progress in linear programming-based algorithms for integer programming: an exposition. *INFORMS J. Comput.*, 12(1):2–23, 2000.

[Mar03] I. Maros. *Computational techniques of the simplex method.* International Series in Operations Research & Management Science, 61. Kluwer Academic Publishers, Boston, MA, 2003.

[She99] N. Sherwani. *Algorithms for VLSI physical design automation.* Kluwer Academic, Boston, USA, 1999.

[Stu99] J. F. Sturm. Using SeDuMi 1.02, a MATLAB toolbox for optimization over symmetric cones. *Optim. Methods Softw.*, 11/12(1-4):625–653, 1999. Special Issue on Interior point methods; code available at: `sedumi.mcmaster.ca`.

[Zha99] Y. Zhang. User's guide to LIPSOL: linear-programming interior point solvers V0.4. *Optim. Methods Softw.*, 11/12(1-4):385–396, 1999. Special Issue on Interior point methods; code available at: `www.caam.rice.edu/~zhang/lipsol`.

A Appendix: Tables

Table 5: Run times using `SeDuMi` (seconds)

Test & Preprocess Used	Pre-Time (secs)	Solve Time (secs)	Total Time (secs)	Solve Time Ratio	Total Time Ratio
1 - none	0.0	11.7	11.7	100 %	100 %
1 - basic	0.4	6.1	6.5	52 %	56 %
1 - full	16.1	5.4	21.5	46 %	184 %
2 - none	0.0	16696.2	16696.2	100 %	100 %
2 - basic	9.2	5829.8	5819.0	35 %	35 %
2 - full	2287.1	6636.7	8923.8	40 %	53 %
3 - none	0.0	3197.4	3197.4	100 %	100 %
3 - basic	11.7	1101.4	1113.1	34 %	35 %
3 - full	4444.4	1159.5	5603.9	36 %	175 %
4 - none	0.0	28965.9	28965.9	100 %	100 %
4 - basic	16.4	15087.3	15103.7	52 %	52 %
4 - full	4844.2	14732.6	19576.8	51 %	68 %
5 - none	0.0	54191.7	54191.7	100 %	100 %
5 - basic	18.5	33516.6	33535.1	62 %	62 %
5 - full	4935.9	26958.9	31894.8	50 %	59 %

Table 6: Run times using `linprog` (seconds)

Test & Preprocess Used	Pre-Time (secs)	Solve Time (secs)	Total Time (secs)	Solve Time Ratio	Total Time Ratio
1 - none	0.0	462.6	462.6	100 %	100 %
1 - basic	0.4	155.2	155.6	34 %	34 %
1 - full	16.1	217.6	233.7	47 %	50 %
2 - none	NA	**Fail**[♭]	NA	NA	NA
2 - basic	9.2	45497.8	45507.0	NA	NA
2 - full	NA	**Fail**[♭]	NA	NA	NA
3 to 5 - none	NA	**Fail**[♭]	NA	NA	NA
3 to 5 - basic	NA	**Fail**[♭]	NA	NA	NA
3 to 5 - full	NA	**Fail**[♭]	NA	NA	NA

[♭] out of memory error

Table 7: Run times using `LipSol` (seconds)

Test & Preprocess Used	Pre-Time (secs)	Solve Time (secs)	Total Time (secs)	Solve Time Ratio	Total Time Ratio
1 - none	0.0	32.2	32.2	100 %	100 %
1 - basic	0.4	19.8	22.2	61 %	69 %
1 - full	16.1	16.6	32.7	52 %	102 %
2 - none	0.0	40686.8	40686.8	100 %	100 %
2 - basic	9.2	28350.9	28360.1	70 %	70 %
2 - full	2287.1	26350.1	28637.2	65 %	70 %
3 - none	0.0	6304.0	6304.0	100 %	100 %
3 - basic	11.7	3007.8	3018.8	48 %	48 %
3 - full	4444.4	3440.0	7884.4	55 %	125 %
4 - none	0.0	104933.3	104933.3	100 %	100 %
4 - basic	16.4	61618.6	61635.0	59 %	59 %
4 - full	4844.2	72329.5	77173.7	69 %	74 %
5 - none	0.0	95646.9	95646.9	100 %	100 %
5 - basic	18.5	102409.4	102427.9	107 %	107 %
5 - full	4935.9	97540.1	102467.0	102 %	107 %

Table 8: Run times using $\texttt{CPlex}^{\texttt{on}}$ (seconds)

Test & Preprocess Used	Pre-Time (secs)	Solve Time (secs)	Total Time (secs)	Solve Time Ratio	Total Time Ratio
1 - none	0.0	0.8	0.8	100 %	100 %
1 - basic	0.4	0.2	0.6	25 %	75 %
1 - full	16.1	0.2	16.3	25 %	2040%
2 - none	0.0	1678.3	1678.3	100 %	100 %
2 - basic	9.2	787.8	797.0	47 %	47 %
2 - full	2287.1	609.1	2896.2	36 %	173 %
3 - none	0.0	2581.6	2581.6	100 %	100 %
3 - basic	11.7	200.6	212.3	8 %	8 %
3 - full	4444.4	117.8	4562.2	5 %	177 %
4 - none	0.0	7169.3	7169.3	100 %	100 %
4 - basic	16.4	4869.8	4886.2	68 %	68 %
4 - full	4844.2	5475.1	10319.3	76 %	144 %
5 - none	0.0	11400.3	11400.3	100 %	100 %
5 - basic	18.5	9663.3	9681.8	85 %	85 %
5 - full	4935.9	7491.3	12427.2	66 %	109 %

Table 9: Run times using $\texttt{CPlex}^{\texttt{off}}$ (seconds)

Test & Preprocess Used	Pre-Time (secs)	Solve Time (secs)	Total Time (secs)	Solve Time Ratio	Total Time Ratio
1 - none	0.0	9.4	9.4	1175 %	1175 %
1 - basic	0.4	2.5	2.9	313 %	363 %
1 - full	16.1	0.3	16.4	38 %	2050 %
2 - none	0.0	4251.7	4251.7	253 %	253 %
2 - basic	9.2	2385.0	2394.2	142 %	143 %
2 - full	2287.1	1583.6	3870.7	94 %	231 %
3 - none	0.0	5663.8	5663.8	219 %	219 %
3 - basic	11.7	2216.4	2228.1	86 %	86 %
3 - full	4444.4	266.9	4711.3	10 %	182 %
4 - none	0.0	16490.9	16490.9	230 %	230 %
4 - basic	16.4	13822.4	13838.8	193 %	193 %
4 - full	4844.2	10321.7	15165.9	144 %	212 %
5 - none	0.0	21814.7	21814.7	191 %	191 %
5 - basic	18.5	12720.1	12738.6	112 %	112 %
5 - full	4935.9	20061.6	24997.5	176 %	219 %

Table 10: Run times using $\texttt{CPlex}^{\texttt{on}}_{\texttt{ipm}}$ (seconds)

Test & Preprocess Used	Pre-Time (secs)	Solve Time (secs)	Total Time (secs)	Solve Time Ratio	Total Time Ratio
1 - none	0.0	2.5	2.5	100 %	100 %
1 - basic	0.4	1.8	2.2	72 %	88 %
1 - full	16.1	1.6	17.7	64 %	708 %
2 - none	0.0	916.1	916.1	100 %	100 %
2 - basic	9.2	716.9	726.1	78 %	79 %
2 - full	2287.1	622.4	2909.5	68 %	318 %
3 - none	0.0	248.5	248.5	100 %	100 %
3 - basic	11.7	182.5	194.2	73 %	78 %
3 - full	4444.4	155.4	4599.8	63 %	1850 %
4 - none	0.0	1726.5	1726.5	100 %	100 %
4 - basic	16.4	1483.5	1499.9	86 %	87 %
4 - full	4844.2	1489.7	6333.9	86 %	367 %
5 - none	0.0	2344.1	2344.1	100 %	100 %
5 - basic	18.5	2047.3	2065.8	87 %	88 %
5 - full	4935.9	1968.2	6904.1	84 %	295 %

Table 11: Run times using $\texttt{CPlex}^{\texttt{off}}_{\texttt{ipm}}$ (seconds)

Test & Preprocess Used	Pre-Time (secs)	Solve Time (secs)	Total Time (secs)	Solve Time Ratio	Total Time Ratio
1 - none	0.0	2.6	2.6	104 %	104 %
1 - basic	0.4	1.5	1.9	60 %	76 %
1 - full	16.1	1.7	17.8	68 %	712 %
2 - none	0.0	917.5	917.5	100 %	100 %
2 - basic	9.2	735.0	744.2	80 %	81 %
2 - full	2287.1	648.9	2936.0	71 %	320 %
3 - none	0.0	260.1	260.1	105 %	105 %
3 - basic	11.7	153.2	164.9	62 %	66 %
3 - full	4444.4	153.1	4597.5	62 %	1850 %
4 - none	0.0	1782.0	1782.0	103 %	103 %
4 - basic	16.4	1509.0	1525.4	87 %	88 %
4 - full	4844.2	1491.7	6335.9	86 %	367 %
5 - none	0.0	2135.0	2135.0	91 %	91 %
5 - basic	18.5	1990.4	2008.9	85 %	86 %
5 - full	4935.9	1929.0	6864.9	82 %	293 %