Title:

Improving solution times in VLSI optimization problems

Authors:

W. L. Hare

AdvOl-Report No. 2006/11

August 2006, Hamilton, Ontario, Canada

Technical Report:
Improving solution times in VLSI optimization problems

W. L. Hare*

August 24, 2006

Keywords VLSI programming, Large Scale Linear Programming, Interior Point Methods,

Abstract

The problem of wire layout (or routing) in VLSI design can be written as a large scale linear program with upper bound constraints. Due to the size of these programs, optimization by use of classical methods can be extremely time consuming. This report surveys some of the recent techniques and ideas which have been emerging on how to improve the time required to solve these highly structured linear programs.

Techniques for improving the runtime for these linear programs generally fall into two categories: improving existing software and creating new algorithms. The first method is generally done by applying novel preprocessing techniques before using preexisting software or by parameter tuning on the software. The second approach is generally accomplished by exploiting the specific structure of the linear programs created in VLSI design. In this report we begin by discussing two preprocessing techniques which have shown some promise in VLSI design. We follow this with a discussion on parameter tuning in `CPlex` version 10. The parameter tuning strongly suggests that the best way to solve these linear programs is to embed the upper bound constraints into the constraint matrix and then solve via interior point methods. This naturally leads to the question of how to best embed the upper bound constraint into the constraint matrix. We explore several new methods to do this, and discuss how each of these methods affects the work required to calculate the Newton directions associated with interior point methods.

1 Introduction

Recent advances in technology for manufacturing integrated circuits has allowed for the production of circuits with millions of components. This has lead to extremely large and time consuming problems on how to layout and connect the components for optimal circuit performance. In general the *VLSI circuit design* problem is modelled as a sequence of discrete optimization problems (see [She99] and references therein). The interest of this work lies in one such problem: the routing problem.

One method of modelling the routing problem 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

*This report was researched and written while working at McMaster University under a MITACS post-doctorial fellowship and and NSERC Discovery Grant #5-48923.

can be laid. The total routing problem is now dissected into a series of smaller subproblems, each of which is designed to preform a specific propose (joining a collection of nodes for example). For each subproblem a series of trees is created to solve the subproblem. The routing problem now becomes a matter of selecting exactly one tree for each subproblem such that the total routing is optimized in some manner [She99] [AVY06].

More mathematically, let the routing problem be subdivided t subproblems. For subproblem i, we have a collection of trees, $\mathbf{t}_1^i, \mathbf{t}_2^i, \ldots \mathbf{t}_{m_i}^i$ 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

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

where $\mathbf{1}_k$ represent 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

$$T\mathbf{x_t} = \mathbf{1}_t, \quad \mathbf{x_t} \in \{0,1\}^n.$$

(The letter 'T' can be thought to stand for 'Tree selection', and $\mathbf{x_t}$ as the '**t**ree selection component' of the objective vector.)

Let the graph representing the circuit layout have p edges. We next create a $p \times n$ matrix P, such that each column represent a given tree and each row represents the edges used by that tree. For example if column 3 was $[0,1,1,0,1,0,0\ldots0]^\top$ then the tree represented by $(\mathbf{x_t})_3$ would use edges 2, 3, and 5. (As such, the letter 'P' can be thought to stand for 'Tree Partition'.) The vector $P\mathbf{x_t}$ is then a list of the edge congestions for the given selection $\mathbf{x_t}$. Finally we let $\mathbf{x_r}$ represent the maximal edge congestion, $\mathbf{x_r} = \max P\mathbf{x_t}$, and $M_\mathbf{r}$ be a (given) maximal edge capacity. Thus we have the following constraint system:

$$\left.\begin{array}{rcl} T\mathbf{x_t} & = & \mathbf{1}_t \\ \mathbf{x_r} & \geq & \max P\mathbf{x_t} \\ \mathbf{x_r} & \leq & M_\mathbf{r} \\ \mathbf{x_t} & \in & \{0,1\}^n \end{array}\right\} \Leftrightarrow \left\{\begin{array}{rcl} T\mathbf{x_t} & = & \mathbf{1}_t \\ P\mathbf{x_t} - \mathbf{x_r}\mathbf{1}_p & \leq & 0 \\ \mathbf{x_r} & \leq & M_\mathbf{r} \\ \mathbf{x_t} & \in & \{0,1\}^n. \end{array}\right.$$

Our objective might be to minimize the total wire length ($c_t^\top \mathbf{x_t}$, where c_t provides the wire lengths associated with each net represented by $\mathbf{x_t}$), minimize the total number of bends ($c_t^\top \mathbf{x_t}$, where c_t provides the number of bends, or *vias*, associated with each tree represented by $\mathbf{x_t}$), minimize the maximal congestion (set $M_\mathbf{r} = \infty$ and use $\mathbf{x_r}$ as the objective function), or any combination of the above ($c^\top \mathbf{x} = c_t^\top \mathbf{x_t} + c_r^\top \mathbf{x_r}$, where c_t and c_r are linear penalties associated with $\mathbf{x_t}$ and $\mathbf{x_r}$ respectively). In order to maintain a high degree of generality, we shall use $c^\top \mathbf{x}$.

To solve the resulting VLSI problem we often relax the integer constraint ($\mathbf{x_t} \in \{0,1\}^n$) to create the relaxed linear program

$$\min\left\{c^\top \mathbf{x} \ : A\mathbf{x} = b, \ \ 0 \leq \mathbf{x}, \ \ \mathbf{x_t} \leq \mathbf{1}_n, \ \ \mathbf{x_r} \leq M_\mathbf{r}\right\} \tag{2}$$

where

$$\mathbf{x} = \begin{bmatrix} \mathbf{x_t} \\ \mathbf{x_r} \\ \mathbf{x_s} \end{bmatrix}, \ A = \begin{bmatrix} T & 0 & 0 \\ P & -\mathbf{1}_p & I \end{bmatrix}, \ b = \begin{bmatrix} \mathbf{1}_n \\ 0 \end{bmatrix}, c = \begin{bmatrix} c_n \\ c_r \\ 0 \end{bmatrix}$$

($\mathbf{x_s}$ are introduced slack variables). The solution of this relaxed linear program can then be rounded to create a integer solution to the original problem.

Thus the relaxed program associated with VLSI design is a $(t+p)\times(n+1+p)$ linear program with $n+1$ upper bound constraints ($\mathbf{x_t}$ is length n, $\mathbf{x_r}$ is length 1, and $\mathbf{x_s}$ is length p). In application the number of trees to select from, n, is tens to hundreds of thousands, while the number of subproblems is of the order $t \approx n/4$. The size of the tree description matrix (P) can range greatly but is generally between $n/4$ and n: ($p \in [n/4, n]$). The result is extremely large linear programs which are difficult to solve by traditional methods.

In this report we examine various methods of improving the runtime required to solve these large scale linear programs. We begin, in Section 2, by discussing some of the recent work in how to preprocess these problems. In Section 3 we discuss how parameter tuning has lead to the opinion that the best method to solve the large scale linear programs of VLSI design is via interior point methods. We then provide, in Section 4, a brief description of how interior point methods are design and implemented. Of particular interest is Subsection 4.1 where we discuss how bounds are typically dealt with in interior point methods. In Section 5 we discuss two alternate methods of embedding the upper bound constraints for VLSI programs. The first, found in Subsection 5.1, is a simple approach based on noting which upper bounds are redundant to the problems solution. The second, found in Subsection 5.2, is a more complicated approach with relies on inserting the upper bounds into the constraint matrix in a manner which exploits the triangle-like structure of the constraint matrix. Both of these methods show a high theoretical promise. In Section 6 we discuss some techniques for solving large scale systems of linear equations, and their impact on the constraint embedding technique developed in Subsection 5.2. We conclude in Section 7.

Remark 1.1 (Open Questions) *Of particular note in this work is the large number of ideas for improving runtime in VLSI optimization problems which remain open for exploration. It is the hope of the author that this report will provide the background necessary to get new researchers started on these ideas and hopefully lead to many successful research projects. In order to help readers locate the open questions touched upon by this work, we shall end each section with a subsection outlining the open questions of the section.*

2 Preprocessing Techniques

Preprocessing techniques for linear programming have existed since at least the mid 1970s [BMW75] [GS81], but probably long before that. Simply put, the idea behind preprocessing is to take an arbitrary problem and reformulate it in a manner which reduces the total time required to solve the problem. In the case of linear programming, $\min\{c^\top \mathbf{x} : A\mathbf{x} = b, \mathbf{x} \geq 0\}$, this is typically approached through the following steps (see [JNS00] [Mar03] and references therein):

1. remove empty rows of A,
2. remove empty columns of A,
3. remove singleton rows of A and update the corresponding column,
4. remove duplicate rows of A,
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 $\mathbf{x}$.

In the case of VLSI programming, the specific structure of the matrix A allows for some improvements on classical methods. In [HLT06] it was shown that splitting the matrix A into its two natural row sections $[T\ 0\ 0]$ and $[P\ \ -\mathbf{1}\ I]$ allowed for a mathematical analysis which reduced the runtime of preprocessing. It was also shown that one can create a quick lower bound for the variable $\mathbf{x_r}$ and use that lower bound to significantly reduce the total size of A [HLT06, Table 2]. The techniques in [HLT06] successfully reduced VLSI problem sizes by approximately 30% and solution times by approximately 15% to 50% (depending on the solver used) [HLT06, Table 4].

Another preprocessing technique for VLSI design was recently preprocessed in [BKV02]. Based on the work of Otten, [Ott82], the work strongly suggests that by reordering the matrix A in a manner which forces nonzero elements closer to the diagonal improves overall runtime. Unfortunately the theory and algorithms in [BKV02] are based on the matrix A being a $\{0, 1\}$ matrix instead of the $\{0, 1, -1\}$ matrix which generally appears in VLSI problems (reducing A to a $\{0, 1\}$ matrix is only achievable if $\mathbf{x_r}$ does not appear in the objective function).

2.1 Open Questions

• Can the preprocess techniques of [HLT06] be further improved, either by better implementation of the current techniques or by implementation of new techniques? One likely direction to approach this would be developing bound tightening techniques designed specifically for VLSI programs.
• Can a more theoretical understanding of how [Ott82] works be achieved?
• Can the techniques in [Ott82] [BKV02] be generalized to $\{0, 1, -1\}$ matrices? If so, how, and does it still result in the same improvement in runtime?
• Can the techniques in [Ott82] [BKV02] be extended to an arbitrary sparse matrix?

3 Method Selection and Parameter Tuning

While preforming numerical testing on the preprocessing techniques developed in [HLT06] an interesting pattern occurred. It was found that, in the five test problems examined, solution times using `CPlex` version 10 on its default setting were approximately 4.2 times longer than using `CPlex` version 10 with the parameter `'lpmethod'` set to `'barrier'` (this changes `CPlex`'s solving technique from the dual-simplex method to a log-barrier interior point method). This, of course, leads one to believe that the dual-simplex method is not the most efficient method to solve the linear programs associated with VLSI design. However, this is not sufficient to conclude that interior point methods are definitively better than simplex methods for these problems. Exploration into primal-simplex methods, primal-dual-simplex methods has yet to be preformed. Furthermore, when the VLSI problems are small, the dual-simplex method has been seen to converge faster than the interior point method. However, in these cases even the interior point method convergence in under 10 seconds, so perhaps this is of little concern in practice.

Another concern in determining which linear program solving method might be best suited to solving VLSI problems, arises in the terms parameter tuning. Most, if not all, software packages allow the user to set a large collection of parameters used during optimization. In theory, any feasible parameter selection should eventually converge, but in practice certain parameter selections convergence faster than others. Unfortunately, in general the "best" parameter selection for one problem is unrelated to the "best" parameter selection for a different problem. However, if the problems share a similar structure (as is the case in VLSI design) there is often a good generic parameter selection for both problems.

This had lead some researchers to work on parameter tuning for various linear program solvers. In the past the idea of parameter tuning has largely been based on a combination of instinct and experimentation. However, a novel approach by Audet and Orban maybe of service in this regards. In [AO06], Audet and Orban discuss a method of automatically tuning the parameters for any optimization solver by using a black box optimization method to minimize the solver's average runtime.

3.1 Open Questions

• Is there a particular method best suited for solving the linear programs arising in VLSI design? More specifically:

  • How do the various simplex method rank? (see [Mar03] for simplex method algorithms)
  • How do the various interior point methods rank? (see [RTV06] for interior point algorithms)
  • In what cases do interior point methods outperform simplex methods?

More generally:

  • What is a good indicator that an interior point method will beat a simplex method for best runtime, for arbitrary linear programs?

• Can one determine generic optimal parameters for solving VLSI problems for a given solver? In particular, will the techniques in [AO06] work?

4 Interior Point Methods

Due to the large size of the linear programs resulting from VLSI design, interest in solving these problem via interior point methods has grown rapidly. Towards that end, let us quickly review the basics of interior point methods for linear programming. For further information we refer the reader to the book [RTV06].

Consider the primal-dual pair

$$
\begin{aligned}
&(\mathrm{P}) \qquad \min\{c^\top \mathbf{x} : A\mathbf{x} = b, \mathbf{x} \geq 0\} \\
&(\mathrm{D}) \qquad \max\{b^\top \mathbf{y} : A^\top \mathbf{y} + \mathbf{s} = c, \mathbf{s} \geq 0\}
\end{aligned}
$$

Supposing that both equation (P) and (D) are feasible, it is well known, in the area of linear programming, that $(\bar{\mathbf{x}}, \bar{\mathbf{y}}, \bar{\mathbf{s}})$ forms a solution set to (P)-(D) if and only if

$$A\bar{\mathbf{x}} = b, \quad A^\top \bar{\mathbf{y}} + \bar{\mathbf{s}} = c, \quad \bar{X}\bar{\mathbf{s}} = 0,$$

(as usual, variables denoted by capital letters represent the diagonal matrix generated by the vector represented by the lower case letter: $\bar{X} = \text{diag}\bar{\mathbf{x}}$). Interior point methods attempt to solve this system by finding accurate solutions to the *central path equations*:

$$A\mathbf{x} = b, \quad A^\top \mathbf{y} + \mathbf{s} = c, \quad X\mathbf{s} = \mu\mathbf{1},$$

for increasingly small μ. When $\mu = 0$ these equations give the precise primal-dual optimality conditions, while as μ approaches zero these equations yield increasingly accurate approximates to optimality [RTV06, §2].

4.1 Imbedding Bounds

If the linear program to be solved involves upper or lower bound constraints, then a first step to implementing an interior point method is to rewrite the problem in the form given in equation (P). Lower bounds are easily dealt with by shifting the variables:

$$\left\{ \begin{array}{rcl} A\mathbf{x} & = & b \\ \mathbf{x} & \leq & B_u \\ \mathbf{x} & \geq & B_l \\ \mathbf{x} & \geq & 0 \end{array} \right\} \Leftrightarrow \left\{ \begin{array}{rcl} A\hat{\mathbf{x}} & = & b - A(\max\{B_l, 0\} \\ \hat{\mathbf{x}} & \leq & B_u - \max\{B_l, 0\} \\ \hat{\mathbf{x}} & \geq & 0 \\ (\hat{\mathbf{x}} & = & \mathbf{x} - \max\{B_l, 0\}) \end{array} \right\}$$

Upper bounds are dealt with by inserting them into the constraint matrix A as follows:

$$\left\{ \begin{array}{rcl} A\mathbf{x} & = & b \\ \mathbf{x} & \leq & B_u \\ \mathbf{x} & \geq & 0 \end{array} \right\} \Leftrightarrow \left\{ \begin{array}{rcl} \begin{bmatrix} A & 0 \\ I & I \end{bmatrix} \begin{bmatrix} \mathbf{x} \\ \mathbf{x_s} \end{bmatrix} & = & \begin{bmatrix} b \\ B_u \end{bmatrix} \\ \mathbf{x}, \mathbf{x_s} & \geq & 0 \end{array} \right\}$$

where B_{u_i} is set to ∞ if no upper bound for $\mathbf{x}_i$ is given. Any row with $B_{u_i} = \infty$ can then be removed from the constraint matrix. Thus if the original matrix A is $M \times N$ with U upper bound constraints, then the final program contains $M + U$ variables and $N + U$ constraints.

Remark 4.1 *In the case of VLSI design $M = t + p$, $N = n + 1 + p$, and $U = n + 1$, so the new matrix is $(t + p + n + 1) \times (2n + 2 + p)$.*

4.2 Newton Steps

Probably the most popular approach to solving the central path equations is to calculate the so called "Newton Direction" and then preforming a line search in that direction [RTV06, §10]. The Newton Direction is found by solving the following system, which arises directly from (P)-(D),

$$\begin{array}{rcl} A\Delta\mathbf{x} & = & 0, \\ A^\top \Delta\mathbf{y} + \Delta\mathbf{s} & = & 0, \\ S\Delta\mathbf{x} + X\Delta\mathbf{s} & = & \mu\mathbf{1} - X\mathbf{s}. \end{array}$$

Through simple linear algebra is it possible to reduce this system to:

$$\begin{aligned}\Delta \mathbf{x} &= \mu \mathbf{s}^{-1} - \mathbf{x} - XS^{-1}\Delta \mathbf{s}, &\text{(3a)}\\ \Delta \mathbf{s} &= -A^\top \Delta \mathbf{y}, &\text{(3b)}\\ AXS^{-1}A^\top \Delta \mathbf{y} &= b - A\mu \mathbf{s}^{-1}. &\text{(3c)}\end{aligned}$$

It is easy to see that the majority of the "work" in solving this system lies in solving the final equation: equation (3c). As such it is important to consider methods of writing the linear program so that equation (3c) is as easy to solve a possible.

Remark 4.2 *As mentioned, if A is a $M \times N$ matrix and U upper bound constraint are given, then the final program contains a $(M + U) \times (N + U)$ matrix. As such equation* (3c) *reduces to solving a $(M + U) \times (M + U)$ system of linear equations.*

In the case of VLSI design embedding the upper bound constraints into the constraint matrix via the technique in Subsection 4.1 results in equation (3c) *being $(t + p + n + 1) \times (t + p + n + 1)$ system of linear equations.*

4.3 Open Questions

As with any algorithm, a key element to the success of interior point methods is finding a good starting point. In particular if A, b, and c form the linear program of a VLSI design problem can one:

- Find a primal-dual feasible starting point? (i.e. vectors $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{s}$ with $A\mathbf{x} = b$, $\mathbf{x} \geq 0$, $A^\top \mathbf{y} + \mathbf{s} = c$, $\mathbf{s} \geq 0$)
- Find a strictly primal-dual feasible starting point? (i.e. a feasible $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{s}$, such that $\mathbf{x} > 0$ and $\mathbf{s} > 0$)
- Find a (strictly) feasible starting point on (or close to) the central path? (i.e. a (strictly) feasible $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{s}$ with $X\mathbf{s} = \mu\mathbf{1}$ (or $\approx \mu\mathbf{1}$) for some $\mu > 0$)
- Find a (strictly) feasible starting point with a low duality gap? (i.e. a (strictly) feasible $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{s}$ with $c^\top \mathbf{x} - b^\top \mathbf{y}$ as small as possible)
- Find a strictly feasible starting point near the central path with a low duality gap?

5 Alternate Methods for Embedding Upper Bounds

In this section we outline two methods for embedding the upper bounds of the VLSI program into the constraint matrix in a manner which reduces the size of the system of linear equations resulting from equation (3c).

5.1 Releasing constraints

For ease of reference we shall refer to the method laid out in this subsection as the **`RC` method**.

The `RC` method for reducing the size of the final constraint matrix is quite simple, and works by noticing that the upper bound constraints $\mathbf{x_t} \leq 1$ are actually redundant in the problem. This is clear by the fact $T\mathbf{x_t} = \mathbf{1}$, where T is a $\{0, 1\}$ matrix with full column support and $\mathbf{x_t} \geq 0$. Thus if $\mathbf{x_{t}}_i > 1$ for any i, then column i of T will result in $T\mathbf{x_t} > \mathbf{1}$. Therefore, we can ignore the $\mathbf{x_t} \leq 1$

constraints completely, leaving only the single upper bound constraint $\mathbf{x_r} \leq M$. Restructuring the matrix A as discussed in Subsection 4.1 now only requires the addition of one row and one column.

Using the RC method results in having equation (3c) reduce to a $(t+p+1) \times (t+p+1)$ system of linear equations.

5.2 Using the Triangularity of A

For ease of reference we shall refer to the method laid out in this subsection as the **Tri method**.

To understand the Tri method to reducing the size of the final system of linear equations, let us begin by recalling the constraint matrix that we are interested in:

$$A = \begin{bmatrix} T & 0 & 0 \\ P & -\mathbf{1}_p & I \end{bmatrix}, \text{ where } T = \begin{bmatrix} \mathbf{1}_{m_1}^\top & 0 & 0 & \dots & 0 \\ 0 & \mathbf{1}_{m_2}^\top & 0 & \dots & 0 \\ \vdots & & & \ddots & \\ 0 & 0 & 0 & \dots & \mathbf{1}_{m_t}^\top \end{bmatrix}. \tag{4}$$

Notice that A takes a form that is particularly similar to a lower triangular matrix. Indeed the bottom right corner is exactly the identity, while the upper left corner is a sort of flattened version of the identity. When inserting the upper bound constraints into the constraint matrix we shall attempt to exploit this property as much as possible. Before we discuss this further, let us consider the case when the constraint matrix A of a linear program is actually lower triangular.

Example 5.1 (A lower triangular) *If the matrix A is an $M \times M$ lower triangular, then inverting A requires $\mathcal{O}(M^2)$ operations. Therefore solving the system*

$$AXS^{-1}A^\top \Delta\mathbf{y} = b - A\mu s^{-1}$$

only requires $\mathcal{O}(M^2)$ operations, instead of the $\mathcal{O}(M^3)$ operations required to solve a generic $M \times M$ system of linear equations. Since calculating $\Delta\mathbf{s}$ and $\Delta\mathbf{x}$ via equations (3b) and (3a) are of order $\mathcal{O}(M^2)$ operations, the solution complexity of system (3) is reduced by a factor of M.

In light of Example 5.1, our goal to is to rewrite A in a form which makes its triangular nature more accessible. This is done by inserting the upper bound constraints into the top portion of A as explained in Algorithm 5.2.

The important result of Algorithm 5.2 is that the matrix $\widehat{T}$ is lower triangular. As such the first $n+1+p$ columns of $\widehat{A}$ form a lower triangular matrix. The remaining $n+1$ columns are extremely sparse, containing only $n-t+1$ nonzero entries. The result is that by adding $n-t+1$ rows and $n-t+1$ columns we have created a new constraint matrix which distinctly locates the triangular nature of A. Since Algorithm 5.2 does nothing other than insert the upper bound constraints into constraint matrix it is clear that, with the correct choice of b, the linear program resulting from this new matrix is equivalent to the original linear program.

Proposition 5.2 *Suppose that the matrix A, along with vectors $b = \begin{bmatrix} \mathbf{1}_n \\ 0 \end{bmatrix}$ and c, are used to form a linear VLSI problem as in equation (2). Using Algorithm 5.2 define,*

$$\widehat{A} = \begin{bmatrix} \widehat{T} & 0 & 0 & \widehat{I} & 0 \\ 0 & 1 & 0 & 0 & 1 \\ P & -\mathbf{1}_p & I & 0 & 0 \end{bmatrix}, \quad \hat{\mathbf{x}} = \begin{bmatrix} \mathbf{x} \\ \mathbf{x_{s_1}} \\ \mathbf{x_{s_2}} \end{bmatrix}, \quad \hat{b} = \begin{bmatrix} \mathbf{1}_n \\ M \\ 0 \end{bmatrix}, \text{ and } \hat{c} = \begin{bmatrix} c \\ 0 \\ 0 \end{bmatrix}.$$

Algorithm 5.2 : [**Restructuring the constrain matrix for VLSI programs**]

- From the $t \times n$ matrix T create the $n \times n$ matrix $\widehat{T}$:

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

- Create the $n \times n - t$ matrix $\widehat{I}$:

$$\widehat{I} = \begin{bmatrix} \begin{matrix} I_{m_1-1} \\ 0 \end{matrix} & 0 & \dots & 0 \\ 0 & \begin{matrix} I_{m_2-1} \\ 0 \end{matrix} & \dots & 0 \\ \vdots & & & \ddots \\ 0 & 0 & \dots & \begin{matrix} I_{m_t-1} \\ 0 \end{matrix} \end{bmatrix}$$

- Using the $p \times n$ matrix P create the $(n+1+p) \times (n+1+p+n-t+1)$ matrix $\widehat{A}$:

$$\widehat{A} = \begin{bmatrix} \widehat{T} & 0 & 0 & \widehat{I} & 0 \\ 0 & 1 & 0 & 0 & 1 \\ P & -\mathbf{1}_p & I & 0 & 0 \end{bmatrix}$$

Then

$$\min\{c^\top \mathbf{x} : A\mathbf{x} = b, \quad \mathbf{x} \geq 0, \quad \mathbf{x_t} \leq \mathbf{1}, \quad \mathbf{x_r} \leq M\} = \min\{\hat{c}^\top \hat{\mathbf{x}} : \widehat{A}\hat{\mathbf{x}} = \hat{b}, \quad \hat{\mathbf{x}} \geq 0\}.$$

Proof: If A is defined as in equation (4) and $\widehat{A}$ is created via Algorithm 5.2, then

$$\left\{ \mathbf{x} : \begin{array}{rcl} A\mathbf{x} & = & \begin{bmatrix} \mathbf{1}_t \\ 0 \end{bmatrix} \\ \mathbf{x_t} & \leq & 1 \\ \mathbf{x_r} & \leq & M \\ \mathbf{x} & \geq & 0 \end{array} \right\} = \left\{ \mathbf{x} : \begin{array}{rcl} \exists\, \mathbf{x_{s_1}}, \mathbf{x_{s_2}} & \geq & 0 \text{ s.t.} \\ \widehat{A} \begin{bmatrix} \mathbf{x} \\ \mathbf{x_{s_1}} \\ \mathbf{x_{s_2}} \end{bmatrix} & = & \begin{bmatrix} \mathbf{1}_n \\ M \\ 0 \end{bmatrix} \\ \mathbf{x} & \geq & 0. \end{array} \right\}$$

Thus, our choice of $\hat{b}$ and $\hat{c}$ make the linear programs equivalent. □

Having shown that the linear programs generated by A and $\widehat{A}$ are equivalent, we now turn our attention to the problem of solving

$$\widehat{A}D^2\widehat{A}^\top \Delta\hat{\mathbf{y}} = R,$$

where $\widehat{A}$ is the matrix created in Algorithm 5.2, D^2 is the positive diagonal matrix XS^{-1}, and R is the fixed vector $\hat{b} - \widehat{A}\mu s^{-1}$. To improve notation let us introduce,

$$L = \begin{bmatrix} \widehat{T} & 0 & 0 \\ 0 & 1 & 0 \\ P & -\mathbf{1}_p & I \end{bmatrix}, \text{ so } \widehat{A} = \begin{bmatrix} L & \begin{bmatrix} \widehat{I} & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \end{bmatrix}.$$

The matrix L incorporates the lower triangular (and therefore easily invertible) portion of the matrix $\widehat{A}$. We also use the naturally corresponding block structure for D:

$$D = \begin{bmatrix} D_L & 0 & 0 \\ 0 & D_{\mathbf{s_1}} & 0 \\ 0 & 0 & D_{\mathbf{s_2}} \end{bmatrix}$$

Note that the labelling of $D_{\mathbf{s_1}}$ and $D_{\mathbf{s_2}}$ follow naturally from their correspondence to the slack variables $\mathbf{x}_{\mathbf{s_1}}$ and $\mathbf{x}_{\mathbf{s_2}}$ from the primal equation.

Using this notation, it is easy to confirm

$$\begin{aligned}
\widehat{A}D^2\widehat{A}^\top &= \begin{bmatrix} L & \begin{bmatrix} \widehat{I} & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix} \end{bmatrix} \begin{bmatrix} D_L^2 & 0 & 0 \\ 0 & D_{\mathbf{s_1}}^2 & 0 \\ 0 & 0 & D_{\mathbf{s_2}}^2 \end{bmatrix} \begin{bmatrix} L^\top \\ \begin{bmatrix} \widehat{I}^\top & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \end{bmatrix} \\
&= LD_L^2L^\top + \begin{bmatrix} \widehat{I}D_{\mathbf{s_1}}^2\widehat{I}^\top & 0 & 0 \\ 0 & D_{\mathbf{s_2}}^2 & 0 \\ 0 & 0 & 0 \end{bmatrix}
\end{aligned}$$

Therefore we find $\widehat{A}D^2\widehat{A}^\top\Delta\hat{\mathbf{y}} = R$ reduces to,

$$\Delta\hat{\mathbf{y}} = (L^\top)^{-1}D_L^{-2}L^{-1}\left(R - \begin{bmatrix} \widehat{I}D_{\mathbf{s_1}}^2\widehat{I}^\top\Delta\mathbf{y}_1 \\ D_{\mathbf{s_2}}^2\Delta\mathbf{y}_2 \\ 0 \end{bmatrix}\right). \tag{5}$$

The interesting point of equation (5) is that the right hand side does not involve the variable $\Delta\hat{\mathbf{y}}_3$. Thus if we knew a (approximate) value for $\Delta\hat{\mathbf{y}}_1$ and $\Delta\hat{\mathbf{y}}_2$ we could immediately compute (approximate) $\Delta\hat{\mathbf{y}}_3$ by use of equation (5).

Also of note is the fact that $\bar{L}^{-1}$ is readily calculable as,

$$L^{-1} = \begin{bmatrix} \widehat{T}^{-1} & 0 & 0 \\ 0 & 1 & 0 \\ P\widehat{T}^{-1} & \mathbf{1}_p & I \end{bmatrix} \text{ where } \widehat{T}^{-1} = \begin{bmatrix} \begin{matrix} I_{m_1-1} & 0 \\ -\mathbf{1}_{m_1-1}^\top & 1 \end{matrix} & 0 & 0 & \cdots & 0 \\ 0 & \begin{matrix} I_{m_2-1} & 0 \\ -\mathbf{1}_{m_2-1}^\top & 1 \end{matrix} & 0 & \cdots & 0 \\ \vdots & & & \ddots & \\ 0 & 0 & 0 & \cdots & \begin{matrix} I_{m_t-1} & 0 \\ -\mathbf{1}_{m_t-1}^\top & 1 \end{matrix} \end{bmatrix}.$$

In order to compute $\Delta\hat{\mathbf{y}}_1$ and $\Delta\hat{\mathbf{y}}_2$ we focus on equation (5). By noting that $\Delta\hat{\mathbf{y}}_3$ is not a variable of the right hand side, we can isolate the upper component of the equation. To do this we make use of the notation

$$\begin{bmatrix} \Gamma_{1,1} & \Gamma_{1,2} & \Gamma_{1,3} \\ \Gamma_{2,1} & \Gamma_{2,2} & \Gamma_{2,3} \\ \Gamma_{3,1} & \Gamma_{3,2} & \Gamma_{3,3} \end{bmatrix} = \Gamma = (LD_L^2L^\top)^{-1} \text{ and } \begin{bmatrix} R_1 \\ R_2 \\ R_3 \end{bmatrix} = R.$$

Straightforward linear algebra yields,

$$\begin{array}{rcll}
\Gamma_{1,1} & = & \widehat{T}^{\top^{-1}}(D_{L,1}^{-2} + P^\top D_{L,3}^{-2} P)\widehat{T}^{-1} & \text{(dimensions } n \times n) \\
\Gamma_{1,2} & = & \widehat{T}^{\top^{-1}} P^\top D_{L,3}^{-2} \mathbf{1}_p & \text{(dimensions } n \times 1) \\
\Gamma_{1,3} & = & \widehat{T}^{\top^{-1}} P^\top D_{L,3}^{-2} & \text{(dimensions } n \times p) \\
\Gamma_{2,1} & = & \mathbf{1}_p^\top D_{L,3}^{-2} P \widehat{T}^{-1} & \text{(dimensions } 1 \times n) \\
\Gamma_{2,2} & = & D_{L,2}^{-2} + \sum D_{L,3}^{-2} & \text{(dimensions } 1 \times 1) \\
\Gamma_{2,3} & = & \mathbf{1}_p^\top D_{L,3}^{-2} & \text{(dimensions } 1 \times p) \\
\Gamma_{3,1} & = & D_{L,3}^{-2} P \widehat{T}^{-1} & \text{(dimensions } p \times n) \\
\Gamma_{3,2} & = & D_{L,3}^{-2} \mathbf{1}_p & \text{(dimensions } p \times 1) \\
\Gamma_{3,3} & = & D_{L,3}^{-2} & \text{(dimensions } p \times p)
\end{array}$$

Thus we have,

$$\begin{array}{rcl}
\begin{bmatrix} \Delta\hat{\mathbf{y}}_1 \\ \Delta\hat{\mathbf{y}}_2 \\ \Delta\hat{\mathbf{y}}_3 \end{bmatrix} & = & \Gamma \left(\begin{bmatrix} R_1 \\ R_2 \\ R_3 \end{bmatrix} - \begin{bmatrix} \widehat{I} D_{\mathbf{s_1}}^2 \widehat{I}^\top \Delta\hat{\mathbf{y}}_1 \\ D_{\mathbf{s_2}}^2 \Delta\hat{\mathbf{y}}_2 \\ 0 \end{bmatrix} \right) \\
\Delta\hat{\mathbf{y}}_1 & = & \Gamma_{1,1}(R_1 + \widehat{I} D_{\mathbf{s_1}}^2 \widehat{I}^\top \Delta\hat{\mathbf{y}}_1) + \Gamma_{1,2}(R_2 + D_{\mathbf{s_2}}^2 \Delta\hat{\mathbf{y}}_2) + \Gamma_{1,3} R_3 \\
\Delta\hat{\mathbf{y}}_2 & = & \Gamma_{2,1}(R_1 + \widehat{I} D_{\mathbf{s_1}}^2 \widehat{I}^\top \Delta\hat{\mathbf{y}}_1) + \Gamma_{2,2}(R_2 + D_{\mathbf{s_2}}^2 \Delta\hat{\mathbf{y}}_2) + \Gamma_{2,3} R_3 \\
\Delta\hat{\mathbf{y}}_3 & = & \Gamma_{3,1}(R_1 + \widehat{I} D_{\mathbf{s_1}}^2 \widehat{I}^\top \Delta\hat{\mathbf{y}}_1) + \Gamma_{3,2}(R_2 + D_{\mathbf{s_2}}^2 \Delta\hat{\mathbf{y}}_2) + \Gamma_{3,3} R_3
\end{array}$$

Finally, noting that the matrix $\widehat{I} D_{\mathbf{s_1}}^2 \widehat{I}^\top$ has only $n-t$ nonempty rows, we see that the computation of $\Delta\hat{\mathbf{y}}_1$ and $\Delta\hat{\mathbf{y}}_2$ reduces to the solving of a $(n-t+1) \times (n-t+1)$ system of linear equations (we need only calculate $\Delta\hat{\mathbf{y}}_1$ for the rows which $\widehat{I} D_{\mathbf{s_1}}^2 \widehat{I}^\top$ is nonzero, as we can then use this information to complete the calculation of $\Delta\hat{\mathbf{y}}_1$). Once $\Delta\hat{\mathbf{y}}_1$ and $\Delta\hat{\mathbf{y}}_2$ are found, the computation of $\Delta\hat{\mathbf{y}}_3$ is a simple matrix vector multiplication.

Using the `Tri` method results in replacing equation (3c) with a $(n-t+1) \times (n-t+1)$ system of linear equations.

5.3 A simple test

As noted at the end of the respective subsections, our two methods result in equation (3c) reducing to either a $(t+p+1) \times (t+p+1)$ system of linear equations or a $(n-t+1) \times (n-t+1)$ system of linear equations. Since the standard upper bound embedding technique results in a $(t+p+n+1) \times (t+p+n+1)$ system of linear equations, either technique is an improvement over the standard approach.

If $t+p$ is considerably greater than $n-t$ or vice versa then the choice of which method to use would become obvious. However, in practice we find $t+p \in n/4 + [n/4, n] \in [n/2, 5n/4]$, while $n-t \approx 3n/4$, so both approaches result in systems of equations of the same approximate order. Therefore, more factors than just system size are likely to play a role in solution time. For example the sparsity of the various systems, or the technique used to solve the systems may play a large role (see Section 6 for further discussion on this point).

As a test problem we consider the final linear program developed in the solving of `prim2` from [AVY06]. The `prim2` test problem is extremely small in comparison to the other test problems.

Nonetheless, prim2 provides us with a feel for how full sized problems might behave. To solve the problem we input the upper bounds into the constraint matrix three separate ways, and then solving the resulting linear systems via interior point solvers SeDuMi and CPlex 10. For the sake of curiosity we also solve the problem via CPlex 10's dual simplex method (CPlex's interior point method is labelled $\texttt{CPlex}_{\texttt{ipm}}$ while its dual simplex method is labelled $\texttt{CPlex}_{\texttt{ds}}$ in Table 1). The timing results appearing in Table 1 below are based on the average runtime over 5 consecutive trials. (Tests were run on a 4x Pentium 3 700MHz cluster with 1GB RAM per node.)

Table 1: Average runtime using various solvers and bound inputting techniques (seconds)

Solver	std method	RC method	Tri Method
SeDuMi	11.246	8.074	8.912
$\texttt{CPlex}_{\texttt{ipm}}$	2.586	2.428	2.492
$\texttt{CPlex}_{\texttt{df}}$	0.252	0.306	0.248

The first way we input the bounds is via the standard bound inputting techniques discussed in Subsection 4.1, we label this std in Table 1. The remaining two bound input techniques correspond to the RC method of Subsection 5.1 and the Tri method of Subsection 5.2.

As one can see from Table 1, in this simple test, when using an interior point method inputting the bounds via the Tri method is only slightly worse than inputting the bounds via the RC method. In this case both the RC and Tri methods yield improvement over the std method. Of course neither SeDuMi nor CPlex makes use of the analysis of Subsection 5.2, so the results here are a very weak guideline for how runtimes may actually compare if a solver based on that analysis were developed.

When using the dual simplex method it would appear that the std method and Tri method are essentially equal, and better than the RC method. This is likely due to CPlex's preprocessor being able to more rapidly determine upper bounds for the problem.

The table also misleadingly suggests that the dual simplex method is faster than the interior point method for this problem. This is due to the small size of this particular test case and, as [HLT06] shows, is not true for larger test problems.

5.4 Open Questions

• The Tri method of constraint embedding shows some promise in its ability to impose new structure on the linear program generated by a VLSI problem. This leads to several interesting questions which summarize how the structure imposed by embedding constraints via the Tri method can be exploited in order to improve solution time.
Specifically:

• Can any of the question in Section 4 be more easily answered if the bounds are embedded via the Tri method?

• Can rapid estimates for $\Delta\hat{\mathbf{y}}$ be generated?

• Given an estimate for $\Delta\hat{\mathbf{y}}$, what can be done to refine the estimate? (This is discussed further in Section 6.)

More completely:

• Can an efficient linear program solver based on the analysis in Subsection 5.2 be created to solve the linear programs arising from VLSI design?

• In order to test a new solver, one will also have to determine a good collection of test problems upon which valid benchmarking of VLSI design problems can be preformed.

6 Solving Systems of Linear Equations

Although there are many different methods of solving systems of linear equations, the methods tend to fall into two categories: direct solving techniques, and iterative techniques.

In the first category, direct solving techniques, one attempts to directly compute the solution to through the tenets of linear algebra. Perhaps the most classical of such techniques is Gauss-Jordan elimination. In Gauss-Jordan elimination one seeks the exact solution to a system of linear equations by using the elementary row operations to put a matrix into row echelon form. Although Gauss-Jordan elimination has the advantage that it solves any (consistent) system in a finite number of steps, the running time of Gauss-Jordan elimination is quite high. Therefore, for large systems it is more reasonable to use an alternate method.

An alternate collection of direct solving methods for systems of linear equations is matrix factorization. In matrix factorization one attempts to decompose a given matrix into a product of easily invertible matrices. Generally this requires that the matrix to be factorized has some underlying structure, such as symmetry or positive definiteness, which can be exploited. Classical decomposition techniques include Cholesky factorization and Bunch-Parlett factorization. The Cholesky factorization is an algorithm that decomposes a symmetric positive-definite matrix into a lower triangular matrix and the transpose of the lower triangular matrix. The resulting decomposition is then easily inverted. Bunch-Parlett's algorithm is quite similar, but does not require the matrix be positive-definite, only symmetric. The nice thing about matrix decomposition techniques are they are easily done symbolically, and so do not have to be fully redone at every iteration.

The other standard method for solving systems of linear equations is iterative techniques. These methods work by improving on an approximate solution via simple updating algorithms. Classical iterative methods, such as the Gauss-Seidel or Jacobi method, are quite efficient when they converge. However, since these methods were developed through the tenets of linear algebra, they generally require some structure on the matrix to ensure convergence. In the case of the Gauss-Seidel and Jacobi methods, the matrix is required to be strictly diagonally dominant.

Recently, new iterative techniques have been developed based on the tenets of optimization. In these techniques one begins by writing a convex function which is minimized exactly at the solution to the system, and then uses any optimization code to minimize the function. For example the solution to $A\mathbf{x} = b$ is also the location of the minimum of the convex function $\frac{1}{2}||A\mathbf{x} - b||^2$. If A happens to be positive definite, one could also consider the convex function $\mathbf{x}^\top A\mathbf{x} - b^\top x$. In a way these techniques are very similar to previous iterative techniques as they improve an approximate solution via an optimization algorithm. However, the advantage of the optimization approach is it removes the need for the matrix to be strictly diagonally dominant, but retains the ability to converge quickly, especially if a good initial estimate to the solution is known.

Returning to linear programming (and specifically the linear programs associated with VLSI design), if one chooses to solve the system of equations generated by equation (3c) via an iterative method, it is imperative to have a good initial guess to the solution. In the case of the `std` or

`RC` constraint embedding methods this appears very difficult, however in the case of the `Tri` method it is possible that properties of $\Delta\hat{\mathbf{y}}$ can be used to provide a good starting point for solution finding.

6.1 Low rank updating

If one approaches solving the system of linear equations $Q\mathbf{x} = r$ via a direct solving or matrix factorization technique, one finds that it is much easier to solve systems where the matrix Q is sparse [ART+04]. In considering the systems which arise during interior point methods we note that Q is sparse if and only if $AA^\top$ is sparse (the diagonal matrix XS^{-1} has no effect on the sparsity of $AXS^{-1}A^\top$). A problem therefore arises if A has a dense column.

To see this, consider the matrix $A = \begin{bmatrix} U & \widehat{A} \end{bmatrix}$ where U is a dense column (or collection of dense columns) and the matrix $\widehat{A}$ is sparse. In this case the matrix $AA^\top$ becomes

$$AA^\top = UU^\top + \widehat{A}\widehat{A}^\top.$$

Since U is a dense matrix, the matrix $UU^\top$ is dense, and hence $AA^\top$ is dense.

To overcome this problem one can apply the famous Sherman-Morrison-Woodbury formula,

$$(Q + UU^\top)^{-1} = Q^{-1} - Q^{-1}U(I + U^\top Q^{-1}U)^{-1}U^\top Q^{-1},$$

to preform a low-rank update [ART+04]. In doing so we avoid inverting the large dense matrix $AA^\top$ and instead invert the large *sparse* matrix $\widehat{A}\widehat{A}^\top$ along with the *small* dense matrix $(I + U^\top Q^{-1}U)$.

In the case of VLSI design, the matrix

$$A = \begin{bmatrix} T & 0 & 0 \\ P & -\mathbf{1} & I \end{bmatrix}$$

has exactly one dense column. Rewriting the Sherman-Morrison-Woodbury formula in terms of the structure of the VLSI matrix may lead to some interesting properties which can be exploited to reduce the time required to solve the system of equations arising from equation (3c).

6.2 Open Questions

• Is any method of solving systems of linear equations particularly well suited for dealing with interior point methods and VLSI problems?
• Does the method of constraint embedding effect which method should be used to solve the resulting system of linear equations?
• Can a good initial guess for the $\Delta\hat{\mathbf{y}}$ arising from the `Tri` constraint embedding method be generated? Can this help increase convergence speed of a linear programming solver for VLSI design based this method?
In regards to the Sherman-Morrison-Woodbury formula we have:

• How does the Sherman-Morrison-Woodbury formula simplify when the matrix A is that of a VLSI design problem with embedded constraints?

• Does the method of constraint embedding effect how the Sherman-Morrison-Woodbury formula simplifies?

• Can this be exploited in a linear program solver specifically designed for dealing with the linear programs arising in VLSI design?

7 Conclusion

In examining the open questions throughout this work, one should notice that the majority of the questions can be summarized by

> *can the structure of the linear programs generated during VLSI design problems be exploited to create faster solution times?*

In Sections 2 and 3 we largely discussed previous approaches to exploit this structure. Section 2 examined how previous work successfully exploited this structure to create an efficient preprocessing routine specifically designed for VLSI problem, as well as discussed how further preprocessing might be accomplished by forcing nonzero elements to cluster along the diagonal. In Section 3 we turned our attention to the question of algorithm selection and parameter tuning. Again we noted that, since the linear programs of VLSI design all share a similar structure, it is likely these questions can be successfully answered (a statement which is not true for arbitrary linear programs).

Sections 4 to 6 examined using interior point methods to solve the linear programs of VLSI design. In Section 4 we gave an overview of interior point methods, and asked if the structure arising from VLSI problems could be exploited to find good starting points for interior point methods. In Section 5 we noted that the structure of the linear programs can be further enhanced by use of what we called the `Tri` method for embedding constraints and asked to what degree this information could be further exploited. In Section 6 we gave an overview of solving systems of linear equations and asked if the structure arising from VLSI problems could be exploited to improve the solve time for the linear system of equations resulting from using interior point methods.

In Sections 4 to 6 we also asked if a new linear program solver could be created, specifically tailored to the linear programs arising in VLSI design. Perhaps the easiest way to approach this goal would be by editing one of the many preexisting linear program solvers which have open source code (`LipSol`, `McIPM`, `SeDuMi`, etc...). Although the creation of such a solver would be a large project, once completed, if the solver was rapid had an easy to use interface, it would make a profound impact in the field of VLSI design.

8 Acknowledgements

The author would like to thank Dr. Tamas Terlaky, for his assistance in editing this work, and his aid in locating funding for this fellowship.

References

[AO06] C. Audet and D. Orban. Finding optimal algorithmic parameters using the mesh adaptive direct search algorithm. to appear *SIAM J. Opt.*, 2006.

[ART$^+$04] E.D. Andersen, C. Roos, T. Terlaky, T. Trafalis, and J.P. Warners. *Numerical Linear Algebra and Optimization*, chapter The use of low rank updates in interior-point methods, pages 3–14. Science Press, New York, 2004.

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

[BKV02] L. Behjat, D. Kucar, and A. Vannelli. A novel eigenvector technique for large scale combinatorial problems in VLSI layout. *J. Comb. Optim.*, 6(3):271–286, 2002. New approaches for hard discrete optimization (Waterloo, ON, 2001).

[BMW75] A. L. Brearley, G. Mitra, and H. P. Williams. Analysis of mathematical programming problems prior to applying the simplex algorithm. *Math. Programming*, 8:54–83, 1975.

[GS81] M. Guignard and K. Spielberg. Logical reduction method in zero-one programming (minimal preferred variables). *Oper. Res.*, 29(1):49–74, 1981.

[HLT06] W. L. Hare, M. J. J. Liu, and T. Terlaky. Efficient preprocessing for vlsi optimization problems. submitted to: *Comp. Opt. Theory and Appl.*, 2006.

[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.

[Ott82] R.H.J.M Otten. Eigensolutions in top-down layout design. pages 1017–1020. 1982.

[RTV06] C. Roos, T. Terlaky, and J.-Ph. Vial. *Interior point methods for linear optimization.* Springer, New York, 2006. Second edition of *Theory and algorithms for linear optimization* [Wiley, Chichester, 1997; MR1450094].

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