ISE

Industrial and
Systems Engineering

Global Routing in VLSI Design: Algorithms, Theory, and Computational Practice

Antoine Deza
McMaster University

Chris Dickson
McMaster University

Anthony Vannelli
University of Guelph

Hu Zhang
Canadian Imperial Bank of Commerce

Tamás Terlaky
Lehigh University

Report: 08T-008

Global Routing in VLSI Design: Algorithms, Theory, and Computational Practice

Antoine Deza [a,*] Chris Dickson [a] Tamás Terlaky [b]
Anthony Vannelli [c] Hu Zhang [d]

[a] *McMaster University, Department of Computing and Software, Hamilton, Ontario, L8S 4K1, Canada*

[b] *Lehigh University, Department of Industrial and Systems Engineering, Bethlehem, Pennsylvania, USA*

[c] *University of Guelph, College of Physical and Engineering Science, Guelph, Ontario, Canada*

[d] *Canadian Imperial Bank of Commerce, Toronto. Ontario, Canada*

Abstract

Global routing in VLSI (very large scale integration) design is one of the most challenging discrete optimization problems in computational theory and practice. In this paper, we present a polynomial time algorithm for the global routing problem based on integer programming formulation with a theoretical approximation bound. The algorithm ensures that all routing demands are satisfied concurrently, and the overall cost is approximately minimized.

We provide both a serial and parallel implementation as well as develop several heuristics used to improve the quality of the solution and reduce running time. We provide computational results on a two sets of well-known benchmarks and show that, with a certain set of heuristics, our new algorithms perform extremely well compared with other integer-programming models.

Key words: VLSI

* Corresponding author.
Email addresses: deza@mcmaster.ca (Antoine Deza), dicksocr@mcmaster.ca (Chris Dickson), terlaky@lehigh.edu (Tamás Terlaky), vannelli@uoguelph.ca (Anthony Vannelli), Hu.Zhang@cibc.ca (Hu Zhang).

1 Introduction

VLSI circuit layout is the process by which the physical layout of a circuit is realized from its functional description and specifications. Due to the exponential increase in complexity of integrated circuits, computer-aided design (CAD) tools have been instrumental in this design process. VLSI physical design is a multi-phase process, where each phase typically falls into one of the following three classes; Partitioning, placement, and routing. In the partitioning phase, we split the chip into smaller, more manageable pieces. The assumption is that each of these pieces may be designed independently of one another. In the placement phase, we fix the locations of all blocks within the chip, as well as produce a list of blocks which need to be connected with wire. In the routing phase, the goal is to find a realization of the connections provided from the placement phase. Typically, routing is broken into two distinct processes; Global routing, and detailed routing. In global routing, we wish to find the approximate interconnections between the blocks. Detailed routing takes the output from the global router and produces the exact geometric layout of the wires to connect the blocks. In this paper, we will focus on the global routing problem and provide a polynomial-time algorithm with an approximation bound. Additionally, we will provide a set of heuristics which improve the quality of the approximate solutions, as well as reduce the time taken to obtain them.

1.1 Global Routing in VLSI Design

In the global routing phase of VLSI design, we assume that the circuits are in a one-layer frame. We model the chip as a lattice graph, where each channel in the chip corresponds to an edge in the lattice graph. Pins of the chip components are found at the intersection of these edges, which correspond to vertices in the lattice graph. We define a *net* to be a group of pins which are to be connected. In an instance, we are given a set of nets, each of which has pins that must be connected by wire. Additionally, there are constraints to the number of wires that may pass through any given channel. A solution is a set of trees in the lattice graph, one for each net, corresponding to the wires in the chip routing the given nets. In a solution to the global routing problem, we should produce connections that obey these constraints. The goal is to minimize some property of these connections (such as wire-length or edge congestion).

The global routing problem is $\mathcal{NP}$-hard[15]. Thus, heuristics have been used to obtain approximate solutions. In general, the solution methodologies may be split into two classes: (i) sequential routing and (ii) concurrent routing.

In sequential routing, the nets are ordered based on certain criteria and routed one by one in this sequential order. This idea was first introduced by Lee [14] and is known as the *Maze Runner* heuristic. Several enhancements to this idea have been shown in [8,12]. One disadvantage to sequential routing is that the nets must be ordered in some artificial way. In general, nets are ordered based on their importance, bounding-box areas, or numbers of terminals [23]. The quality of the solution depends heavily on the ordering. As well, there is no theoretical guarantee on performance.

In concurrent routing, integer programming approaches are often utilized, which attempt to route all nets at once (concurrently). In a sense, this is more of a "global" approach to the problem. Generally, we model the problem as a $\{0,1\}$-integer linear programming (ILP) problem where we select one tree to route each net. This selection should minimize the desired objective function, while still enforcing the given edge capacity constraints.

The choice of the objective function is important to obtain a good solution. In [26] and [19], the goal is to minimize the total wire-length required for all nets. These methods do not take into account the number of bends in the trees (vias in the physical layer). Vias increase the cost of chip manufacturing, as well as decreasing the performance of the chip by increasing the heat generated. Other models attempt to minimize the maximum tree length [16], or minimize the maximum edge congestion. Minimizing the maximum edge congestion is essentially equivalent to the multicast congestion problem in communication networks [10,2]. Another approach for the objective function is to take a linear combination of the above properties. In this way, we can control many factors at once [25].

Generally, it is impractical to solve the ILP directly as practical problems in VLSI design are usually too large. Recent approaches to solve these ILPs relax the ILP to a linear program (LP), and round the solution of this LP to find an approximate feasible solution. In [24], a linear relaxation of the routing problem is formulated as a multi-commodity network flow problem. One can employ randomized rounding [20,21] to obtain an integer solution from the fractional solution to the LP-relaxation.

In this paper, we study the model proposed in [25]. The objective function is a convex combination of total wire-length and total number of vias of trees selected for routing the nets. This model covers the impact of wire-length, vias, and also edge congestion. For the global routing problem, this model generalizes the previous models developed in [3,4,21,26]. They consider three important factors: total wire-length, edge congestion, and the number of vias. The edge capacity is allowed to vary according to local requirements instead of posing additional edge length for areas of high congestion. For instance, one may wish to reduce the edge congestion to a relatively small number in and

around potential hot spots. The goal is to minimize a convex combination of the total wire-length and total number of bends in the trees. This combination can vary to conform to the user's actual requirements.

In the approximation algorithms for the routing problem in [25], they first apply a binary search strategy to reformulate the linear relaxations to *convex min-max resource-sharing problems* (*packing problems* in the linear case). Then they use the approximation algorithm in [10] as a subroutine to solve the packing problem, which generalizes the approximation algorithm for convex min-max resource-sharing problems by [6] to the case when the sub-problem is hard to approximate.

We will present the detailed ILP formulation in Section 2 and present an implementation of the asymptotic approximation algorithm in Section 3. In Section 4 we propose some heuristics used to improve the quality of the solution, as well as reduce computation time. In Section 5 and 6 we present the computational results and conclusions.

1.2 Our Contribution

In this paper, we present the methodology towards a high-performance serial and parallel implementation of the generalized model for the global routing problem proposed in [25]. This algorithm approximately solves the LP-relaxation to obtain approximate solutions to the global routing problem by applying randomized rounding. A 2-approximate Steiner minimal tree solver that was first presented by Mehlhorn in [18]. Although there is a theoretical 2-approximate solution in the worst case, we find that for our application the bound is closer to **1.10**. The serial version of our global routing algorithm uses a path saving technique to reduce the time in approximating Steiner trees. This can reduce running time by up to a factor a 10 in our test caes. Since we must store a significant amount of data to save paths, a parallel version of the algorithm was developed to lower the memory demand. We generate trees in parallel which is highly suitable to our algorithm, as this constitutes the majority of computation. Since the order in which we find trees does not matter, this part of our algorithm scales perfectly.

Additionally, we have also developed a class of heuristics to reduce running time as well as improve the solution quality. The idea is to confine a certain percentage of the total nets to a single tree generated in the first iteration. We investigate two approaches on choosing which nets become fixed; non-decreasing bounding-box area, and non-decreasing sum of bounding-box dimensions. We find that using the heuristic of bounding-box area, we can decrease the overflow by up to 25% as well as reducing the wire-length by up to

5%. Fixing based on the sum of bounding-box dimensions is also able to reduce overflow by up to 20%, while making no sacrifices in terms of wire-length.

2 Mathematical Formulation

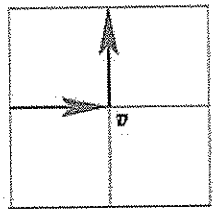


Fig. 1. A path with a bend on v.

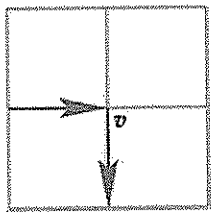


Fig. 2. A path with a bend on v.

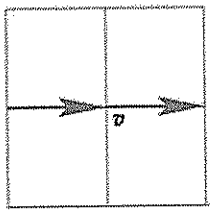


Fig. 3. A path without bend on v.

Formally, given an planar edge-weighted lattice graph $G = (V, E)$ (rectangular holes are allowed) and nets $S_1, \ldots, S_K \subseteq V$, the edge set is associated with a length function $l : E \to \mathbb{R}^+ \cup \{0\}$ and a capacity function $c : E \to \mathbb{R}^+$. We assume that $|S_k|$ is bounded by some constant for all $k = 1, \ldots, K$. The given edge capacity can be less than the physical channel capacity in order to reduce the possibility of hot spots in the solution. A feasible solution is a set of K trees spanning $S_1, \ldots, S_K$ with respect to the edge capacity constraints. The overall cost of the solution consists of two parts: (i) the edge cost and (ii) the total number of bends in the trees called the bend-dependent vertex cost (see Figures 1, 2, and 3). The goal is to minimize the overall cost defined as a linear combination $\alpha l_{\text{total}} + \beta v_{\text{total}}$, where l_{total} is the sum of edge length of all K trees and v_{total} is the sum of numbers of bends of all K trees, while $\alpha, \beta \geq 0$ are artificial weights corresponding to the impact of the total wire-length and the total number of vias whose values are set according to the design requirements and are given in advance. For simplicity, we denote by c_i the capacity of edge $e_i \in E$ from now on. In addition, by scaling, we can set $\alpha + \beta = 1$, i.e., the overall cost is a convex combination of the total edge length and the total number of bends.

The global routing problem in VLSI design is $\mathcal{NP}$-hard. It is at least as hard as the minimum Steiner tree problem in graphs because the global routing problem contains the minimum Steiner tree problem as a special case.

We now develop the ILP formulation of our generalized model. Denote by $\mathcal{T}_k$ the set of all trees in G connecting the vertices in S_k. It is worth noting that $|\mathcal{T}_k|$ can be exponentially large. We also denote by $x_k(T)$ the indicator variable

as follows:

$$x_k(T) = \begin{cases} 1, \text{ if } T \in \mathcal{T}_k \text{ is selected for the net } S_k; \\ 0, \text{ otherwise.} \end{cases}$$

In addition, we define by $l(T)$ and $v(T)$ the length of tree T and the number of bends in the tree T, respectively. Therefore, the ILP of the global routing problem is as follows:

$$\begin{aligned} \min\ & \alpha \textstyle\sum_{k=1}^{K} \sum_{T \in \mathcal{T}_k} l(T) x_k(T) + \\ & \beta \textstyle\sum_{k=1}^{K} \sum_{T \in \mathcal{T}_k} v(T) x_k(T) \\ \text{s.t. } & \textstyle\sum_{T \in \mathcal{T}_k} x_k(T) = 1, && \forall k = 1, \ldots, K; \\ & \textstyle\sum_{k=1}^{K} \sum_{T \in \mathcal{T}_k \& e_i \in T} x_k(T) \leq c_i, && \forall e_i \in E; \\ & x_k(T) \in \{0, 1\}, && \forall T \,\&\, k = 1, \ldots, K. \end{aligned} \tag{1}$$

Here the first set of constraints mean that for any set $\mathcal{T}_k$ we choose exactly one tree for S_k, and the second set of constraints are capacity constraints.

As shown in [25], the following lemma holds:

Lemma 2.1 *For any given $\varepsilon \in (0, 1)$, if we can solve the following linear program*

$$\begin{aligned} \min\ & \lambda \\ s.t.\ & \textstyle\sum_{k=1}^{K} \sum_{T \in \mathcal{T}_k \& e_i \in T} x_k(T)/c_i \leq \lambda, && \forall e_i \in E; \\ & \alpha \textstyle\sum_{k=1}^{K} \sum_{T \in \mathcal{T}_k} l(T) x_k(T)/g + \\ & \beta \textstyle\sum_{k=1}^{K} \sum_{T \in \mathcal{T}_k} v(T) x_k(T)/g \leq \lambda, \\ & \textstyle\sum_{T \in \mathcal{T}_k} x_k(T) = 1, && \forall k = 1, \ldots, K; \\ & x_k(T) \in [0, 1], && \forall T \,\&\, k = 1, \ldots, K, \end{aligned} \tag{2}$$

then we can find a $(1 + \varepsilon)$-approximate solution to the LP-relaxation of (1).

The formulation (2) is a convex min-max resource-sharing problem [6,10] as follows:

$$\min\{\lambda | f_m(x) \leq \lambda, m \in \{1, \ldots, M\}, x \in B\}, \tag{3}$$

where $f : B \rightarrow \mathbb{R}_+^M$ is a vector of M non-negative continuous convex functions

defined on a non-empty convex compact set $B \in \mathbb{R}^N$. In this way, we may approximately solve (1) by using existing algorithms for the convex min-max resource-sharing problem. We shall refer to this LP relaxation as the *fractional global routing problem.*

Since $|\mathcal{T}_k|$ may be exponentially large, many exact algorithms for LPs such as standard interior point methods cannot be applied to obtain a polynomial time algorithm. It is possible to solve such a problem by the volumetric-center [1] or the ellipsoid methods with separation oracle [7]. However, those approaches will lead to a large running time, which is very unsuitable for global routing, as instances of these problems are typically very large.

We will apply the approximation algorithm $\mathcal{L}$ in [10] for convex min-max resource-sharing problems. By applying this algorithm, we avoid the exponential size of $\mathcal{T}$. In $\mathcal{L}$, we generate K minimum Steiner trees for the K nets in each iteration. Thus, there is only a polynomial number of Steiner trees generated in total. In fact, it is shown in [25] that the approximation algorithm generates at most $O(Km(\log m + \varepsilon^{-2}\log\varepsilon^{-1}))$ Steiner trees, and the following result for the fractional global routing holds:

Theorem 2.1 *There exists an $r(1+\varepsilon)$-approximation algorithm for the fractional global routing problem (2) provided that an r-approximate minimum Steiner tree solver is available.*

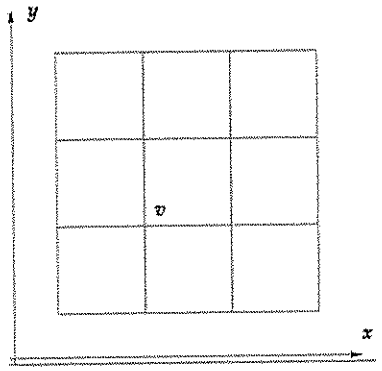


Fig. 4. Original lattice graph G.

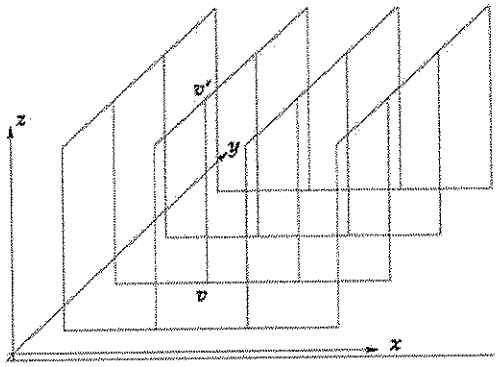


Fig. 5. Virtual layer graph H.

It is shown in [25] that the block problem is as follows:

$$\min_{x\in B} p^T f(x) = \sum_{k=1}^{K} W_k = \sum_{k=1}^{K} \min_{T\in\mathcal{T}_k} \left[\sum_{e_i\in T} \left(\frac{p_i}{c_i} + \frac{\alpha p_{m+1} l_i}{g} \right) + \frac{p_{m+1}\beta v(T)}{g} \right].$$

Here the first term can be regarded as the weights associated with edges in G, while the second term corresponds to the bend-dependent vertex cost. In order to deal with the bend-dependent vertex cost, the *virtual layer method* is proposed in [25] as follows.

We begin by partitioning the edge set E from G into two disjoint subsets E_x and E_y, where $E = E_x \cup E_y$. E_x contains only the horizontal edges from E, while E_y contains only vertical edges. A two-layer graph H is constructed as follows. For each vertex $v \in G$, there are two vertices v and v' in H. These vertices have the same x and y-coordinates as in G but differ in their z-coordinates. To construct the edge set of H we consider the edge sets E_x and E_y. E_x connects vertices of H in the lower (horizontal) layer, while E_y connects vertices of H in the upper (vertical) layer. In order to connect vertices v and v' we introduce an additional edge set E_z. Each edge in E_z connects a pair of vertices v and v' in H. (see Figure 5). We can see that if a path in G has a bend on vertex v_i, this corresponds to using an edge in E_z that connects vertices v_i and v_i' in H. Similarly, a path in H that uses an edge in E_z must have a bend on its corresponding path in G.

We now set the weights to the edges in H. For any edge $e_i \in E_x \cup E_y$, we assign a weight $w_i = p_i/c_i + \alpha p_{m+1} l_i/g$ according to their indices in the original graph G. For every edge in E_z, we assign a weight $p_{m+1}\beta/g$. In this weighted, two-layer graph H, a minimum Steiner tree for a net S_k corresponds to a tree for S_k in G with the minimum W_k. So when we apply Algorithm $\mathcal{L}$, the block problem corresponds to the classical Steiner tree problem in the graph H to minimize the overall edge weight of the Steiner tree connecting the vertices in S_k. We can apply an approximate solver for the Steiner tree problem as the block solver of Algorithm $\mathcal{L}$.

Once we have a fractional solution given by the approximation algorithm $\mathcal{L}$, we must round it to find a feasible integer solution. In addition, we must have a performance guarantee of the approximation ratio. We use randomized rounding as described in [21,20]. Then the following theorem holds:

Theorem 2.2 *There is an approximation algorithm for (2) such that the objective value is bounded by:*

$$\begin{cases} r(1+\varepsilon)OPT + (\exp(1)-1)(1+\varepsilon)\sqrt{r \cdot OPT \ln m}, & \text{if } r \cdot OPT > \ln m; \\ r(1+\varepsilon)OPT + \dfrac{\exp(1)(1+\varepsilon)\ln m}{1+\ln(\ln m/(r \cdot OPT))}, & \text{otherwise,} \end{cases}$$

where OPT denotes the optimal value of the instance, r is the approximation ratio of the block solver, and m is the number of edges in the grid graph.

3 Implementation

In this section, we present an implementation of the approximation algorithm in [25] for the ILP formulation of the global routing problem in VLSI design. We first present a basic outline of this algorithm, then go into some details about the methods for Steiner tree approximation, as well as rounding approximate solutions to the ILP formulation.

3.1 Outline

We now present a basic outline of the approximation algorithm used to solve the LP (2) in Section 2. A *graph re-weighting* technique is used to reduce edge congestion. We outline this in Algorithm 1.

Algorithm 1: Approximation algorithm for global routing in VLSI design.

Input: A graph $G = (V, E)$ and a set S of nets where $|S| = K$ and $S_k \subset V$ for $k \in \{1 \dots K\}$.

Output: A set of K trees where each tree in the set spans its corresponding net in S.

```
1  Initialization of variables and virtual layer graph generation
2  for k ← 1 to K do
3      Call approximate Steiner tree solver to generate a tree for S_k
4  end
5  Compute edge congestion
6  while stopping criteria not satisfied do
7      Reweight edges in graph
8      for k ← 1 to K do
9          Call approximate Steiner tree solver to generate a tree for S_k
10     end
11     Compute a step length τ and move to new solution
12     Update edge congestion
13 end
14 Perform rounding such that we choose one tree to route each net S_i
```

Our input is given as a lattice graph $G = (V, E)$. Usually, this is simplified to two integers corresponding to the length and the width of graph G. Additionally, we may be given a list of missing vertices (holes) and/or an edge length function. If no edge lengths are specified, then they are assumed to be of unit length. We are also given a non-empty set of nets. Each net S_k is a set of vertices (coordinates) in G, where $|S_k| \geq 2$ for $k \in \{1 \dots K\}$.

Line 1 involves initializing local variables as well as transforming the grid graph G into a *virtual layer graph* which will be denoted as H. In lines $2 - 4$

we generate a tree for each net in S. To achieve this, we simply call our approximate Steiner tree solver which will generate a tree when given the graph H and a net S_k. In line 5 we compute the *edge congestion* for each edge in G. The edge congestion for the edge e_i is equal to the number of trees crossing it.

We now enter the main loop of our algorithm. Line 7 reweights the edges in our virtual layer graph. The edge weights are chosen carefully such that highly congested edges will have a larger weight in H than those edges that are less congested. In this way, when we compute the next set of trees in lines $8-10$, the edges that are frequently used in previous iterations will be avoided. In line 11 we compute a step length $\tau \in (0, 1]$ for the current iteration. This step length can be thought of as a measure of "goodness" for the current iteration. The details of computing the step length are discussed in Section 4.

After each iteration of the main loop (lines 6 through 12), we compute the congestion for each edge e_i. However, since we keep the trees generated in previous iterations, we must measure how often each edge is used in *all* iterations. Without loss of generality, the edge congestion for an edge e_i can be scaled by its capacity such that it is a non-negative real number. We will denote this *scaled congestion* as f_i. Formally, $f_i = n_i/c_i$ where n_i is the number of edge crossing edge e_i and c_i is the capacity. A value of f_i that is strictly greater than 1 implies that this edge is over capacity. This leads to the concept of *fractional edge congestion*. We compute the new fractional edge congestion for edge e_i by the following formula:

$$f_i = (1 - \tau) f_i + \tau \hat{f}_i.$$

Here, $\hat{f}_i$ corresponds to the scaled edge congestion of edge e_i for the trees generated in the current iteration for all $i = 1, \ldots, m$. Additionally, we have an extra constraint that corresponds to the objective value. During initialization, the value of τ is set to be 1. Thus, for the first iteration, the fractional edge congestion is equal to the congestion of the current block solution. Now, define λ to be the maximum fractional edge congestion for all edges. That is:

$$\lambda = \max_{e_i \in E} f_i.$$

After each iteration, we wish to decrease the value of λ.

The stopping rules can be varied according to the problem being solved. The problem is *fractionally feasible* when $f_i \leq 1$ for all $i = 1, \ldots, m$.

Finally, in line 13 we finalize a trees, one for routing each net. The details of

this procedure are discussed in Section 3.3.

3.2 Steiner Tree Approximation

The Steiner minimal tree problem is $\mathcal{APX}$-hard. For certain instances, it is possible to get a true Steiner minimal tree fast (but not in polynomial time). Geosteiner [27,28] is a software package that computes minimum Steiner trees, however it operates only on planar lattice graphs. Also, these graphs are assumed to have unit length. Although the edge *lengths* in our grid graph may have unit length, the edge *weights* in the virtual layer graph may not have unit length. Thus, this package is unsuitable in our algorithm. Additionally, Geosteiner does not run in polynomial time in the worst case. From now on, the notion of Steiner minimal trees will be abbreviated as SMT and the abbreviation MST refers to the minimum spanning tree problem.

There are many known approximation algorithms for computing SMT's, where some have performance guarantees or approximation ratios while others do not. We will discuss only those with an approximation ratio, as this is needed to provide a performance guarantee for our overall algorithm. In general, a k-approximation algorithm guarantees that the computed Steiner tree is of no more than k times the length of an optimal SMT. Mehlhorn [18] presented a simple 2-approximation algorithm. Robins and Zelikovsky [22] developed a 1.55-approximation method, implementations of which exist, but yield large running times which is unsuitable for our applications. The best known lower bound of the approximation ratio is $\frac{95}{94}$ [5]. It should be noted that there is no polynomial time approximation algorithm that guarantees this ratio.

We choose to use the 2-approximation algorithm in [18] due to its simplicity and low running time as well as its theoretical performance. Computation results indicate that for our application, the bound is much closer to optimal. The algorithm is as follows:

Algorithm 2: Generates 2-approximate Steiner trees in graphs

Input: A weighted graph $G = (V, E)$ and a set of terminals $S \subseteq V$.
Output: A steiner tree T for the terminal set K in the graph G.

1 Compute the complete distance network N
2 Compute an MST M_N of N
3 Transform M_N into a reduced graph $N[M_N]$ by replacing each edge of M_N by the corresponding shortest path
4 Compute an MST M in $N[M_N]$
5 Transform T into a Steiner tree T by deleting all leaves that are not terminals

An example of this algorithm is illustrated in Figure 6. The computational

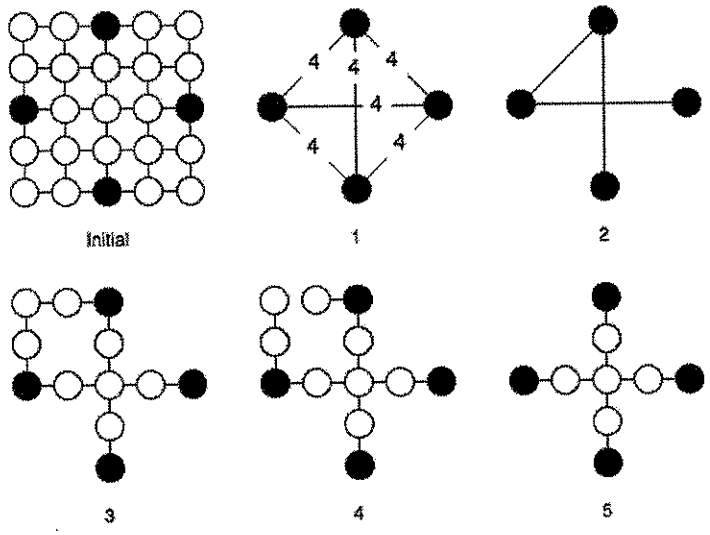


Fig. 6. Illustrates the various steps of Steiner tree approximation algorithm.

bottleneck of this algorithm is the computation of the complete distance network, which requires the solution to the single-source shortest path for each terminal in the set S. We use Dijkstra's algorithm with a binary heap as priority queue in order to achieve a complexity of $O(|E| \log |V|)$. There are other advanced data structures such a Fibonacci heaps or pairing heaps which give a better theoretical complexity result. However, these heaps require significant overhead and only have better performance in the case of vertices with high degree (dense graphs), while our underlying graphs are sparse. We use a $O(|V|^2)$ version of Prim's algorithm to compute minimum spanning trees. It should be noted that the graphs in which we run Prim's algorithm have significantly less vertices than the original lattice graph, so we would not expect to see a big improvement in running time if we used an MST algorithm with a better time complexity.

Additional improvements have been made to this algorithm that not only improve the running time but also the quality of the solution. These are discussed in Section 4.

3.3 Rounding

We implement randomized rounding in order to obtain an integer solution from our fractional solution. Assume that we perform a total of p iterations while solving the LP (2) in Section 2. We know that for each net S_k we will have a total of $p+1$ Steiner trees corresponding to this net. Each tree for net S_k has a corresponding value of $x \in (0, 1]$. Additionally, for each net, the sum of corresponding x values is 1. We regard this value as the probability that

this tree will be chosen to route the given net.

We can think of this randomized rounding as a lottery system. For each net, we have a set of trees, each with a given probability. Trees with an x value close to 1 will almost always be picked while trees with an x value close to zero will rarely be picked. Once we do this for each net, we have our final integer solution.

In practice, we repeat randomized rounding several times in order to obtain the best possible solution. The amount of time spent in rounding is extremely small compared to the time spent generation trees and solving the LP. Also, in the case that we cannot generate a feasible integer solution, we only keep solutions which have fewer constraint violations than the solutions that came previously. In the case of ties in the number of edge capacity violations, we keep the solution that has the lowest objective value.

4 Heuristics and Improvements

We now show some practical improvements we have made to this algorithm. We will present the details of choosing the step length τ in this section. As well, we will discuss some improvements made to the running time of the Steiner tree solver. A multithreaded version of the algorithm is discussed, as well as several heuristics used to improve the quality of the solution.

4.1 Potential Function Minimization

We base our LP solver on a given algorithm for solving convex min-max resource-sharing problems. A potential function for convex min-max resource-sharing problems (3) is introduced in [10] as follows:

$$\phi_t(x) = \ln \theta - \frac{t}{M} \sum_{m=1}^{M} \ln(\theta - f_m(x)), \tag{4}$$

where t is a parameter depending on the error tolerance ϵ and the parameter θ is the solution of the following equation:

$$\frac{t}{M} \sum_{m=1}^{M} \frac{\theta}{\theta - f_m(x)} = 1. \tag{5}$$

It is shown in [10] that a good approximation of the minimum of λ can be attained at an x minimizing the potential function $\phi_t(x)$. The approximation

algorithm for convex min-max resource-sharing problems in [10] is based on this property and is applied in [25] for developing the approximation algorithm for the VLSI global routing problem.

In this algorithm, there is a given formula to compute the step length τ. However, in practice we notice that this produces extremely small values for τ. This causes the algorithm to converge very slowly and thus requires many iterations though the complexity bound in [25] still holds. Therefore, we need to fecide a relatively larger step length for speedup. On the other hand, we cannot choose τ to be too large. Otherwise our algorithm will begin to cycle and not converge.

Our heuristic to determine the step length τ is to find a new iterate x' between the old iterate x and the block solution $\hat{x}$ by line search such that the new fractional congestion f' minimizes the potential function $\phi_t(x)$ over all x' between x and $\hat{x}$. We have used a bisection method in order to minimize this function. Specifically, we approximate the derivative of the potential function by using divided-differences. We then find the zero of this function using the bisection method. It should be noted that we have several fail-safe mechanisms for this line search. We have safe-guarded a maximum number of iterations in case of numerical instability. Also, in the case that a zero does not exist, we simply use the default step length. However, generally when no zero of the derivative to the potential function can be found, the stopping criteria for solving the LP have been met and the approximation bound has been reached. That is to say, no step length can further reduce the congestion, so we can go no further.

4.2 Recording Shortest Paths

With regards to the Steiner tree solver, there are some simple improvements that can be made to significantly reduce the running time. When we compute Steiner trees, it is necessary to first compute a complete distance network of the terminal set. This involves $\binom{|S_k|}{2}$ calls to Dijkstra's algorithm for each net S_k for $k = 1 \ldots K$. However, since in each iteration of our algorithm, we are working with the same graph, the shortest paths from any given vertex in H do not change. By storing the paths, we can eliminate the unnecessary calls to Dijkstra's algorithm. Once we call Dijkstra's algorithm for a given terminal, we can reduce the complexity of finding shortest paths to $O(|V|)$ as we need only to do a linear search to find the destination vertex, and trace its path back to the source vertex. This technique yields a great improvement in running time, especially for large instances with many nets. The only drawback is that this significantly increases the memory demand on the system. After each iteration, we must re-weight the graph H. Thus, the stored paths are only valid for the current iteration and must be computed again in the following

iteration.

4.3 Parallel Tree Generation

Similar to the improvement we made in the Steiner tree solver, we are able to exploit the fact that the graph weights remain constant throughout a given iteration. Because of this, we may generate trees in any order without changing the result of the solution. This naturally leads to the idea of parallelization. If N_p is the number of processors on our machine, then we may assign a total of N_p threads to generate trees. We can assume that, for each instance, each net is labeled from 1 to K where K is the total number of nets. We assign each thread a lower bound and an upper bound which represent the range of nets for which it must produce trees. Specifically, each thread will generate K / N_p trees, where / represents integer division. We also assign the last thread the additional K mod N_p trees. It is worth noting that since K is generally much larger than N_p, these additional trees do not have a large effect on upsetting the workload balance for each thread. In Section 5 we will provide computational results on the time improvement using this technique.

4.4 Hybridization of Concurrent and Sequential Routing

The motivation for this heuristic is that sequential routers are generally able to find a good solution in terms of feasibility, but not in terms of wire-length. However, if we begin our algorithm with a "good" set of trees, then we may be able to improve the total wire-length of the solution, while still maintaining as much feasibility as possible.

In our implementation, we allow the solutions from a sequential router called Labyrinth [13] to warm start our algorithm. Labyrinth uses the maze runner heuristic introduced by Lee in [14]. While being very good at finding feasible solutions, Labyrinth has serveral limitations. First, the grid graph must be uniform. That is, there may not be holes in the graph. Also, capacity must be uniform across the graph, restricted to a single horizontal and vertical value. Additionally, this program does not take into account vias or bends in the trees. We may use these warm solutions to reduce the running time of our own algorithm, but in order to fully exploit this technique, further investigation is needed.

4.5 Fixing Trees

Another class of heuristics that have been implemented deal with a fixing a subset of nets to a single tree generated in the first iteration. The idea is to determine a certain subset of the K total nets, and generate only a single tree for this net. A similar heuristic has been implemented in [13] which allows the user to route all 2-terminal nets first. A side effect of our heuristic is that after the first iteration, we reduce the number of nets we need to find trees for in subsequent iterations. This reduces the problem size, and thus reduces the overall time taken to solve the problem.

We use bounding-box area, and the sum of bounding-box dimensions as the properties for determining which nets become fixed. First, the nets are sorted in non-decreasing order based on the given property. We then select a certain percentage of the total nets for which we wish to fix. A tree is then generated for each of the nets we have selected. The remaining steps of the algorithm are run as usual. The idea for sorting the nets in non-decreasing order is as follows. If we assume that our heuristic is to use bounding-box area, then selecting the nets with the smallest bounding-box area will reduce the probability that the fixed nets will overlap. This increases the probability that the congestion will be spread out more evenly over the area of the chip.

The justification for using the sum of bounding-box dimensions is as follows. Many of the nets in a given instance have a low number of terminals (two or three). Since nets with two colinear terminals have a bounding-box with area zero, we wish to include some nets with more than two terminals in the set of nets to be fixed. However, if we only use bounding box area as a heuristic to fix nets, we are guaranteed to fix all colinear two terminal nets first. By using the dimension sum heuristic, we add the possibility to fix nets with a higher number of terminals. This is desirable as some two terminal nets may be very long, while some three terminal nets may be very close together.

Another issue that arises in the discussion of this class of heuristics, is how to appropriately choose the percentage of nets to fixed for a given instance. In general, there is no way to determine ahead of time what percentage will work the best for a given problem. Additionally, we have tried sorting in non-increasing order, but this method showed no improvement. In Section 5 we will aim to determine some trends for a given set of benchmarks and show that the use of any of these heuristics leads to some improvement.

5 Computational Results

In this section we provide the computational results for our algorithm, as well as for all the heuristics described in Section 4. We will be using the well-known MCNC benchmark collection for our computational tests [17]. All experiments are performed on an 8x AMD Opteron 885 workstation with 64GB of RAM running OpenSUSE 10.2 Linux.

We begin with a comparison of the two main versions of our code. The first uses the path saving technique. The table in Figure 7 compares the running times of the algorithm without the heuristic versus using the heuristic. Only the running time for solving the LP is given. The time taken to round the fractional solution is independent of the method used to solve the LP. It should be noted that only the largest sets of test data are shown in this table.

Circuit	Dimensions		Nets	Tree Generation Time (s)		Imp
	x	y		Path Saving	No Saving	
prim2	26	26	2043	1	6	500%
bio	46	46	3460	18	65	261%
ind1	15	15	1412	1	1	0%
ind2	72	72	10542	135	659	388%
ind3	54	54	18037	56	560	900%
avq.small	80	80	16649	238	1160	387%
avq.large	86	86	18666	322	1420	341%

Fig. 7. Comparison of tree generation times with and without path saving.

As we can see, the running time is greatly reduced using this method. From the benchmarks specified in the table, we can see there is an average of a six times reduction in running time. This reduction depends greatly on the instance of the problem. The exact reduction depends on the number of nets that use a given vertex over all nets in the instance. For example, an instance with 20 nets that has distinct terminals in each net shows no improvement. The more times a terminal is repeated throughout the K total nets, the more time improvement we see. Fortunately, the instances of these problems are very large, and terminals are repeated frequently throughout an instance. However, one must consider the space versus time tradeoff when using this method. As the graph increases in size, the amount of memory required to store the shortest paths will increase rapidly. This heuristic is thus suited for smaller instances and proves to be impractical when considering very large scale problems. However, it can still provide some use. There are techniques emerging that apply refinement techniques to the global routing problem [29]. These techniques start with a small graph and gradually refine the dimensions until the true size is attained. Because the first few steps would typically be small, this heuristic is very beneficial to reduce computation time.

We now present the results for parallel tree generation in our algorithm. A

shared memory model is utilized with POSIX threads underlying the multithreading. We run our algorithm on a select subset of the MCNC benchmarks using 1, 2, 4, 8, and 16 threads in the tree generation phase. The table in Figure 8 illustrates this result. Again, we only test are the larger benchmarks in the data set.

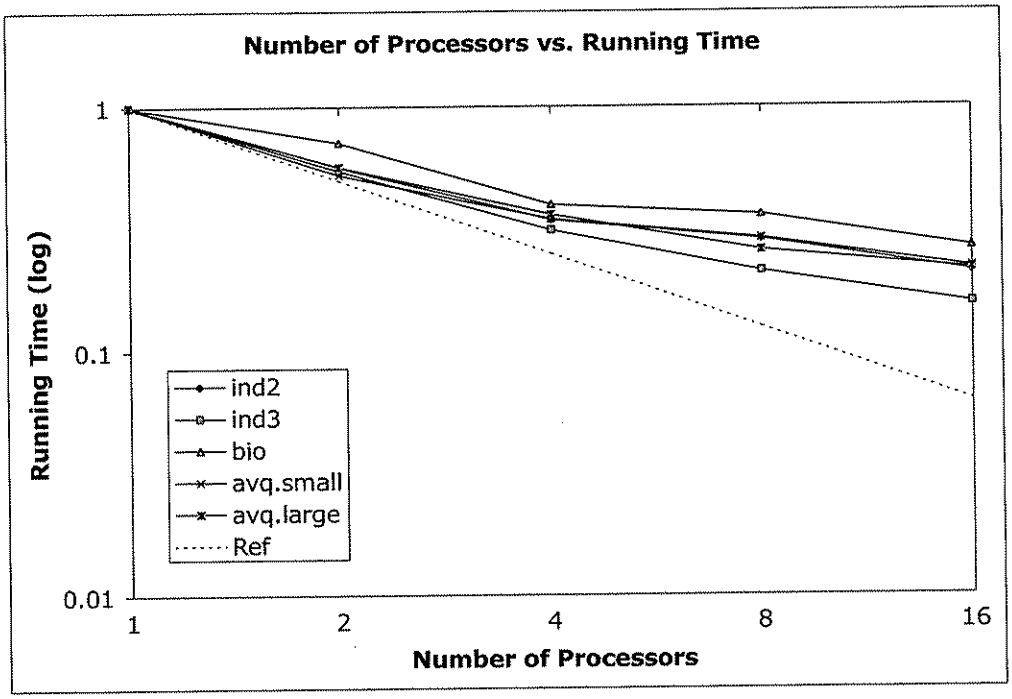


Fig. 8. Effect of multithreading tree generation

The dotted line labeled "Ref" represents perfect scaling. That is, each time we double the number of threads, the running time is halved. Clearly, in practice there is overhead involved in multithreading. Also, only the tree generation phase of our algorithm is parallelized. The other steps of the algorithm such as updating the edge weights in the graph, and computing step lengths are not computed in parallel. However, we can still see that our algorithm scales extremely well. This is due to the fact that the majority of the time in each iteration is spent in generating trees. Thus, speeding up tree generation has a large effect on speeding up the whole algorithm.

We now evaluate the set of heuristics that involve fixing a certain percentage of the nets after the first iteration. The graphs in Figure 9 show the results for fixing nets based on their bounding box area. The x-axis shows the percentage of the total nets that are fixed after the first iteration. The y-axis shows the scaled values of the tree property we wish to evaluate. The three tree properties we focus on are wire-length, edge overflow (as a percentage of the total number of edges) and maximum edge congestion, also known as maximum routing

density (MRD). We scale each y-value to the reference value which occurs when we fix 0% of the nets. We can make several observations from these graphs. First, we can see that for almost all instances, the wire-length is inversely related to the percentage of nets we fix. This is intuitive as in the first iteration, the nets will be short. As we progress throughout the algorithm, the nets grow in length to detour around congested areas. If we fix nets after the first iteration, these nets will not grow in length. However, we must be careful as fixing too many nets will cause congested edges that can never be feasible. This is illustrated in Figure 9(e). We can see that as we fix too many nets, we have edges that are very highly congested. It should also be noted that in Figure 9(c), we see that fixing nets does reduce the total number of infeasible edges. On average, it appears that the 50% to 70% range is optimal for reducing the total number of overflowed edges.

Another heuristic used for fixing nets was the sum of bounding box dimensions. Figures 9(b),9(d), and 9(f) illustrate the results of these tests. For our benchmark set, this heuristic appears to work well at reducing the number of infeasible edges, however the results for reducing the wire-length were mixed. We can see that fixing too many nets indeed reduces the wire-length, but we pay the price in terms of feasibility. We see that fixing 80% or more of the nets causes some edges to be highly congested. This is seen in Figure 9(f).

Our final results in Figures 10 and 11 show tables comparing our best results to that of another concurrent router. The other router uses and ILP based algorithm proposed in [29]. The column "WL Lower Bound" represents the best possible wire-length if we ignore edge capacities. That is, if the optimal SMT is chosen to route each net. GeoSteiner v3.1 is used to find optimal SMT's [9]. In [29], they make the assumption that any net with 10 or more terminals may be ignored by the global router. In order to compare results, we also make this assumption.

For wire-length minimization, we find that our algorithm finds a more feasible solution than in [29] in all but one of the test cases. As well, it can be seen that wire-length is not greatly sacrificed in order to achieve a reduction in the maximum routing demand. On average, we reduce the maximum routing demand by 25.8% while only increasing the wire-length by 1.4%.

6 Conclusion and Future Work

In this paper, we have provided an implementation of a polynomial-time approximation algorithm for the global routing problem in VLSI design. This algorithm has a theoretical approximation bound, however, in practice, our approximate solutions are far closer to optimal than the bound suggests. From

Table 10 we found there is very little distance between the lower bounds for the optimal solutions and our approximate solutions. On average, we find that we are within 3% of the lower bound on wire-length, and in some cases, less than 1%.

A number of techniques and heuristics were developed that can be used to decrease the objective function value, as well as reduce computation time.

We found that by preserving the shortest paths computed throughout an iteration of our algorithm, we can reduce the running time of our serially implemented algorithm by up to a factor of 9 and nearly a factor of 5 on average. Additionally, we provided a parallel implementation of the algorithm which allows for a significant reduction in running time, as well as lower memory usage compared to our serial version which uses path saving. The tree generation phase was multi-threaded in order to minimize the time spent in this step of the algorithm. Since this is the most costly part of the algorithm in terms of running time, we see excellent scaling results as we increase the number of processors, especially in the largest instances that contain many nets. Our computational experiments also showed that confining a certain percentage of the total nets to a single tree not only led to better feasibility results, but helped to reduce the objective function value. In some cases, we are able to reduce the wire-length by 6% while at the same time, reducing the number of overflown edges by nearly 22%. In general, we see that for our test data, fixing 50% to 70% of the nets based on bounding-box area or bounding-box sum does not lead to an increase in the maximum routing demand and in many cases, we see a reduction in overflow as well as wire-length. Additionally, we showed that our algorithm is very competitive with other ILP based models that have been developed and, in many cases, provides better feasibility results with similar objective values.

Our future work involves changing the way the edge congestion is estimated. We believe that by updating the edge congestion several times throughout a given iteration, we can improve the quality of the solution as well as reduce the number of iterations required to obtain it.

Acknowledgments Research supported by an NSERC Discovery grant, a MITACS grant and the Canada Research Chair program.

References

[1] K. M. Anstreicher, Towards a practical volumetric cutting plane method for convex programming, *SIAM Journal on Optimization*, 9 (1999), 190-206.

[2] A. Baltz and A. Srivastav, Fast approximation of minimum multicast congestion

- implementation versus theory, *RAIRO Operations Research*, 38 (2004), 319-344.

[3] L. Behjat, New modeling and optimization techniques for the global routing problem, *Ph.D. Thesis*, University of Waterloo, 2002.

[4] L. Behjat, A. Vannelli and W. Rosehart, Integer linear programming models for global routing, *INFORMS Journal on Computing*, 18(2) (2005), 137-150.

[5] M. Chlebík and J. Chlebíková, Approximation hardness of the Steiner tree problem, *Proceedings of the 8th Scandinavian Workshop on Algorithm Theory* (SWAT 2002), LNCS 2368, 170-179.

[6] M. D. Grigoriadis and L. G. Khachiyan, Coordination complexity of parallel price-directive decomposition, *Mathematics of Operations Research*, 2 (1996), 321-340.

[7] M. Grötschel, L. Lovász, and A. Schrijver, The ellipsoid method and its consequences in combinatorial optimization, *Combinatorica*, 1 (1981), 169-197.

[8] F. Hadlock, Finding a maximum cut of a planar graph in polynomial time, *SIAM Journal on Computing*, 4(3) (1975), 221-225.

[9] GeoSteiner Homepage. Retrieved 03/28/07 from *http://www.diku.dk/geosteiner/*.

[10] K. Jansen and H. Zhang, Approximation algorithms for general packing problems and their application to the multicast congestion problem, *Mathematical Programming*, 114(1) (2008), 183-206.

[11] R. Kastner, Methods and algorithms for coupling reduction, *M.S. Thesis*, Department of Electrical and Computer Engineering, Northwestern University, 2000.

[12] E. S. Kuh and M. Marek-Sadowska, Global routing, in *Layout design and verification (T. Ohtsuki Eds.)*, Elsevier Science Publishers B.V., Amsterdam, 1985, vol. 1, 133-168.

[13] Labyrinth: A global router and routing development tool. Retrieved 03/18/07 from *http://www.ece.ucsb.edu/~kastner/labyrinth/*.

[14] C. Y. Lee, An algorithm for path connection and its application, *IRE Transactions on Electronic Computers*, 10 (1961), 346-365.

[15] T. Lengauer, Combinatorial algorithms for integrated circuit layout, J. Wiley, New York, 1990.

[16] T. Lengauer and M. Lungering, Provably good global routing of integrated circuits, *SIAM Journal on Optimization*, 11(1) (2000), 1-30.

[17] MCNC. Retrieved 03/26/07 from *http://www.cbl.ncsu.edu/benchmarks/layoutsynth92/*.

[18] K. Mehlhorn, A faster approximation algorithm for the Steiner problem in graphs, *Information Processing Letters* 27 (1988), 125-128.

[19] W. R. Pulleyblank, Two Steiner tree packing problems, *Proceedings of the 27th Annual ACM Symposium on Theory of Computing* (STOC 1995), 383-387.

[20] P. Raghavan, Probabilistic construction of deterministic algorithms: approximating packing integer programs, *Journal of Computer and System Sciences*, 37 (1988), 130-143.

[21] P. Raghavan and C. D. Thompson, Randomized rounding: a technique for provably good algorithms and algorithmic proofs, *Combinatorica*, 7 (4) (1987), 365-374.

[22] G. Robins and A. Zelikovsky, Improved Steiner tree approximation in graphs, *Proceedings of the 11th Annual ACM-SIAM Symposium on Discrete Algorithms* (SODA 2000), 770-779.

[23] N. Sherwani, Algorithms for VLSI physical design automation, Kluwer Academic Publishers, Dordrecht, 1999.

[24] E. Shragowitz and S. Keel, A global router on a multi-commodity flow model, *Interaction*, 5 (1987), 3-16.

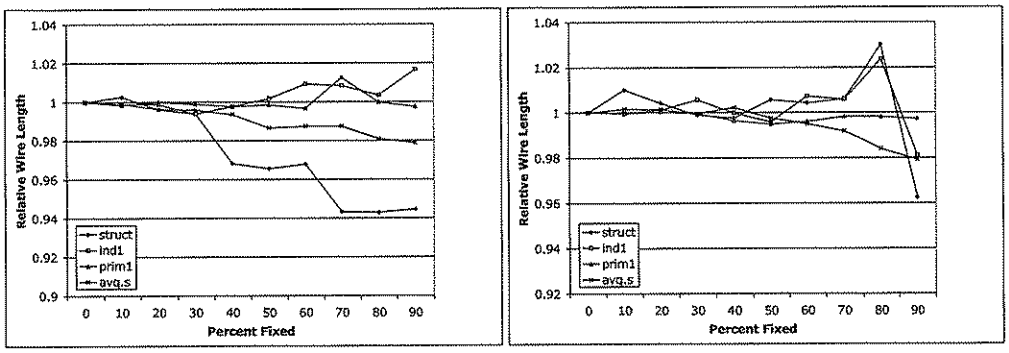
[25] T. Terlaky, A. Vannelli, and H. Zhang, On routing in VLSI design and communication networks, *Discrete Applied Mathematics*, 156 (11) (2008), 2178-2194.

[26] A. Vannelli, An adaptation of the interior point method for solving the global routing problem, *IEEE Transactions on Computer-Aided Design*, 10 (2) (1991), 193-203.

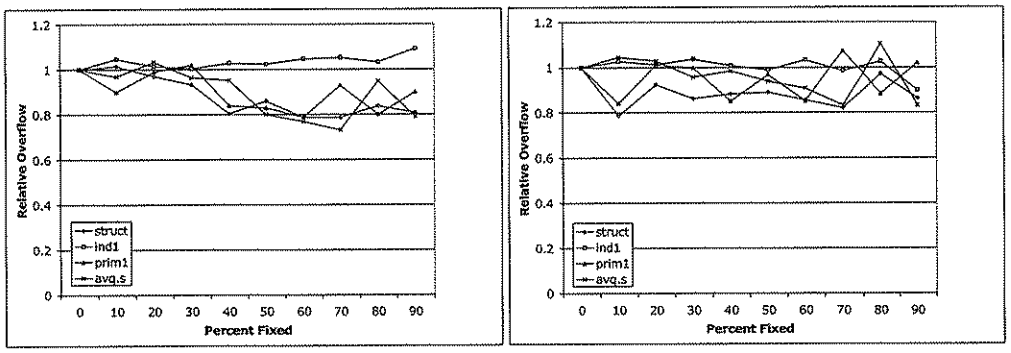
[27] D. M. Warme, Spanning Trees in Hypergraphs with Applications to Steiner Trees, *Ph.D. Thesis*, Computer Science Department, The University of Virginia, 1998.

[28] D. M. Warme, P. Winter and M. Zachariasen, Exact algorithms for plane Steiner tree problems: a computational study, in *Advances in Steiner Trees (D.Z. Du, J.M. Smith and J.H. Rubinstein Eds.)*, Kluwer Academic Publishers, 2000, 81-116.

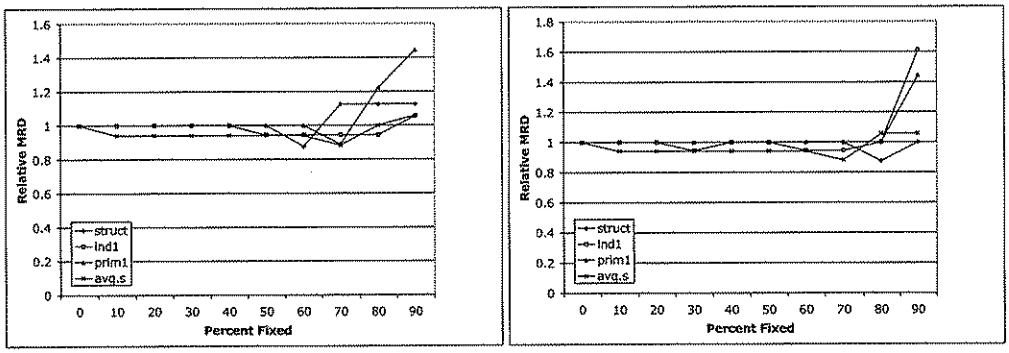
[29] Z. Yang, S. Areibi and A. Vannelli, An ILP based hierarchical global routing approach for VLSI ASIC design, *Optimization Letters*, 1(3) (2007), 281-297.



(a) Area Wire-length (b) Sum Wire-length



(c) Area Overflow (d) Sum Overflow



(e) Area MRD (f) Sum MRD

Fig. 9. Heuristics used to fixed a given percentage of nets

Circuit	Dimensions		Capacity		WL Lower	Wirelength Minimization				Yang06 (WLMM)			
	x	y	horiz	vert	Bound	Wirelength	Vias	MRD	Time	Wirelength	Vias	MRD	Time
fract	8	8	4	4	25480	26173	128	6	19	27515	99	9	3
struct	21	21	5	5	318886	321812	802	9	12	320663	495	10	8
prim1	19	19	6	6	621878	640890	867	9	5	644423	609	17	16
prim2	26	26	10	10	3160108	3342461	3381	15	38	3187597	2493	24	80
bio	46	46	5	5	1018806	1080592	3202	8	206	1032444	2093	12	117
ind1	15	15	10	10	951440	985588	1532	18	10	986911	1098	25	31
ind2	72	72	11	11	12067540	12460247	14006	16	2444	12091689	9490	21	2170
ind3	54	54	29	29	47130739	48769516	23508	36	2091	47205901	16457	33	2131
avq.s	80	80	10	10	9065565	9192349	19703	17	3395	9096280	12146	21	4163
avq.l	86	86	9	9	10382010	10667658	22748	15	5989	10411364	12994	23	4515

Fig. 10. Wire-length minimization results

Circuit	Dimensions		Capacity		WL Lower	Via Minimization				Yang06 (VMM)			
	x	y	horiz	vert	Bound	Wirelength	Vias	MRD	Time	Wirelength	Vias	MRD	Time
fract	8	8	4	4	25480	26241	114	8	3	28143	85	10	3
struct	21	21	5	5	318886	323521	739	9	9	321915	425	12	8
prim1	19	19	6	6	621878	642286	747	10	8	651485	540	15	15
prim2	26	26	10	10	3160108	3332261	3238	16	43	3197111	2266	23	80
bio	46	46	5	5	1018806	1086112	3149	8	325	1039595	1865	12	118
ind1	15	15	10	10	951440	1035751	1550	17	7	989845	1015	26	32
ind2	72	72	11	11	12067540					12114767	8272	25	2167
ind3	54	54	29	29	47130739					47395661	14023	33	2123
avq.s	80	80	10	10	9065565					9122104	10076	20	4163
avq.l	86	86	9	9	10382010					10446320	10831	18	4509

Fig. 11. Via minimization results