A GRAPH THEORETIC TECHNIQUE TO SPEED UP FLOORPLAN AREA OPTIMIZATION*

Ting-Chi Wang and D. F. Wong

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712-1188

TR-91-26 August 1991

* This work was partially supported by the National Science Foundation under grant MIP-8909586, by an ACM SIGDA scholarship, and by an IBM Faculty Development Award.

A Graph Theoretic Technique to Speed up Floorplan Area Optimization*

Ting-Chi Wang and D. F. Wong
Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

Abstract

A well known approach for the floorplan area optimization problem is to first determine a list of all non-redundant implementations of the entire floorplan and then select an optimal floorplan from the list [3,5,8,9,10]. For large floorplans, this approach may fail due to insufficient memory space available to store the implementations of sub-floorplans generated during the computation. An effective method to reduce memory usage is as follows: During the computation, whenever the set of non-redundant implementations of a sub-floorplan exceeds a certain predefined size, we only retain a subset of the implementations that can best approximate the original set. In this paper, we present two algorithms to optimally select implementations for rectangular and L-shaped sub-floorplans. Our algorithms are designed specifically for the floorplan optimization algorithm in [9] but they can also be applied to other algorithms such as [3,5,10] as well. The common key idea of our two algorithms is based on reducing the problem of optimally selecting a subset of implementations to a constrained shortest path problem, which we can solve optimally in polynomial time. We have incorporated the two algorithms into [9] and obtained very encouraging experimental results.

*This work was partially supported by the National Science Foundation under grant MIP-8909586, by an ACM SIGDA scholarship, and by an IBM Faculty Development Award.

1 Introduction

A general approach to floorplan design is to first determine the topology of the floorplan primarily using the interconnection information among the modules [1,2,4,7]. The floorplan topology specifies the relative positions of the modules and is usually represented by a floorplan tree [4,9]. Based on the topology, several optimization problems can then be addressed to minimize various cost measures. If each module is rectangular and has a finite number of implementations, one of the optimization problems, called *floorplan area optimization problem*, is to select an appropriate implementation for each module such that, without changing the floorplan topology, the total area of the floorplan is minimized.

A well known approach for the floorplan area optimization problem is to first determine a list of all non-redundant implementations of the entire floorplan and then select an optimal floorplan from the list [3,5,8,9,10]. The computation is done in a bottom-up fashion starting at the leaves of the floorplan tree (representing the modules) and ending at the root (representing the entire floorplan). For large floorplans, this approach may fail due to insufficient memory space available to store the implementations of sub-floorplans generated during the computation. An effective method to reduce memory usage is as follows: During the computation, whenever the set of non-redundant implementations of a sub-floorplan exceeds a certain predefined size, we only retain a subset of the implementations that can best approximate the original set. In this paper, we present two algorithms to optimally select implementations for rectangular and L-shaped sub-floorplans. Our algorithms are designed specifically for the floorplan optimization algorithm in [9] but they can also be applied to other algorithms such as [3,5,10] as well.

The common key idea of the two algorithms is based on reducing the problem of optimally selecting a subset of implementations to a constrained shortest path problem, which we can solve optimally in polynomial time. We have incorporated the two algorithms into [9], and experimental results indicate that for the test examples where [9] was able to run alone, with the incorporation of the two algorithms, both memory usage and running time were reduced significantly while the solutions remained of comparable quality. Furthermore, for the test examples where [9] failed to run, the two algorithms helped to produce satisfactory solutions.

The rest of this paper is organized as follows. Section 2 defines some important terminologies. Section 3 gives a review of [9] and addresses the approach to reduce memory usage by [9]. Section 4 consists of three subsections: Section 4.1 introduces the constrained shortest path problem and provides a polynomial time optimal algorithm to solve it. Based on the constrained shortest path algorithm, we present in Section 4.2 and 4.3 two implementation selection algorithms to optimally select a subset of implementations for rectangular and L-shaped sub-floorplans, respectively. Experimental results are reported in Section 5, and some concluding remarks are made in Section 6.

2 Preliminaries

A *floorplan* for m modules consists of an enveloping rectangle subdivided by horizontal and vertical line segments into m non-overlapping rectangles called *basic rectangles*. Each basic rectangle must be large enough to accommodate the module assigned to it.

A *floorplan tree* is a hierarchical description of a floorplan. It specifies how the floorplan is obtained by recursively partitioning a rectangle into a finite number of parts. Figure 1 shows a floorplan and its corresponding floorplan tree. Each leaf in the floorplan tree corresponds to a basic rectangle and each internal node corresponds to a composite rectangle in the floorplan.

A *block* in a floorplan is defined as a *connected* set of basic rectangles. Two types of blocks will be considered in this paper: *L-shaped* and *rectangular* blocks. Figure 2 illustrates an L-shaped block and a rectangular block, where the L-shaped block consists of three basic rectangles and the rectangular block consists of four basic rectangles. An *implementation* of an L-shaped block is represented by a 4-tuple (w_1, w_2, h_1, h_2) with $w_1 \geq w_2$ and $h_1 \geq h_2$, where w_1 and w_2 represent the widths of the bottom edge and the top edge, respectively, and h_1 and h_2 represent the heights of the left edge and the right edge, respectively. (See Figure 2.) An *implementation* of a rectangular block is represented by a pair (w, h), where w is the width and h is the height. (See Figure 2.)

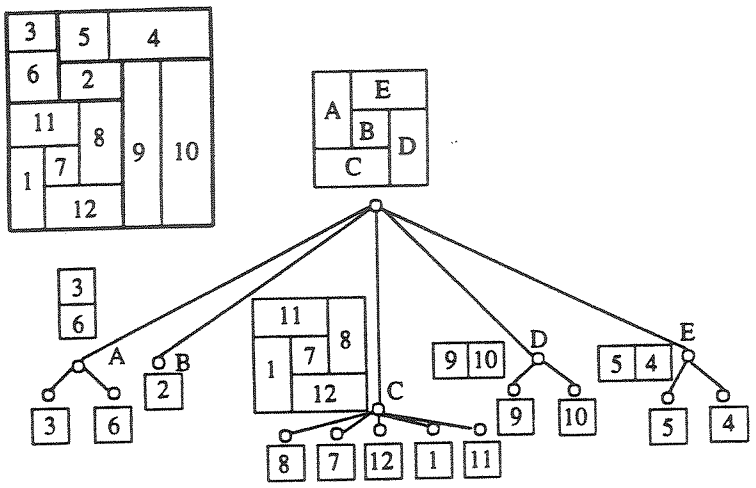

Figure 1: A floorplan and its floorplan tree.

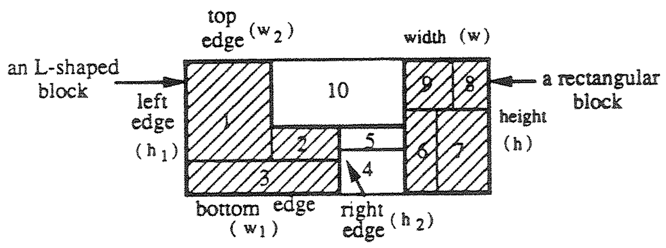

Figure 2: An L-shaped and a rectangular blocks.

Definition 1 Let $I_1 = (w_1, w_2, h_1, h_2)$ and $I_2 = (w'_1, w'_2, h'_1, h'_2)$ be two implementations of an L-shaped block. We say that I_1 *dominates* I_2 if the following four inequalities hold:

$$(1)\ w_1 \geq w'_1;\ (2)\ w_2 \geq w'_2;\ (3)\ h_1 \geq h'_1;\ (4)\ h_2 \geq h'_2.$$

Similarly, if $I_1 = (w, h)$ and $I_2 = (w', h')$ are two implementations of a rectangular block, we say that I_1 *dominates* I_2 if the following two inequalities hold:

$$(1)\ w \geq w';\ (2)\ h \geq h'.$$

Definition 2 Let I_1 and I_2 be two implementations of a block (L-shaped or rectangular). We say that I_1 is a *redundant implementation* if I_1 dominates I_2.

Definition 3 A list $\{(w_{1,1}, w_{2,1}, h_{1,1}, h_{2,1}), (w_{1,2}, w_{2,2}, h_{1,2}, h_{2,2}), ..., (w_{1,n}, w_{2,n}, h_{1,n}, h_{2,n})\}$ is an *L-list* if the following four inequalities hold for all i and j, $1 \leq i < j \leq n$:

$$(1)\ w_{1,i} \geq w_{1,j};\ (2)\ w_{2,i} = w_{2,j};\ (3)\ h_{1,i} \leq h_{1,j};\ (4)\ h_{2,i} \leq h_{2,j}.$$

Definition 4 A list $\{(w_1, h_1), (w_2, h_2), ..., (w_n, h_n)\}$ is an *R-list* if the following two inequalities hold for all i and j, $1 \leq i < j \leq n$:

$$(1)\ w_i \geq w_j;\ (2)\ h_i \leq h_j.$$

Definition 5 An *irreducible L-list* is an L-list in which there are no redundant implementations. Similarly, an *irreducible R-list* is an R-list in which there are no redundant implementations.

3 Our Approach

In this section, we present our approach to reduce memory usage by [9]. We first briefly review the algorithm in [9]. The inputs to this algorithm consist of a floorplan tree specifying the topology of the floorplan, and a finite set of non-redundant implementations for each module.

Given the floorplan tree T, it is first restructured into a binary tree T' such that each internal node in T' either corresponds to a rectangular block or an L-shaped block. Figure 3 illustrates an example of how to restructure T into T'. At the beginning, the algorithm uses an irreducible R-list to store all non-redundant implementations for each leaf in T'. (Note that all non-redundant implementations of a leaf are the same as those of its corresponding module.) Next, the algorithm in a bottom-up fashion determines all non-redundant implementations for each internal node in T'. Let u be an internal node in T'. If u represents a rectangular block, the algorithm constructs an irreducible R-list to store all the non-redundant implementations for u. If u represents an L-shaped block, the algorithm constructs a set of irreducible L-lists to store all the non-redundant implementations for u. During the process of generating all non-redundant implementations for u, the algorithm only considers the *candidates* which are very likely to be non-redundant, and then eliminates the redundant ones from the candidates. The details of the algorithm can be found in [9].

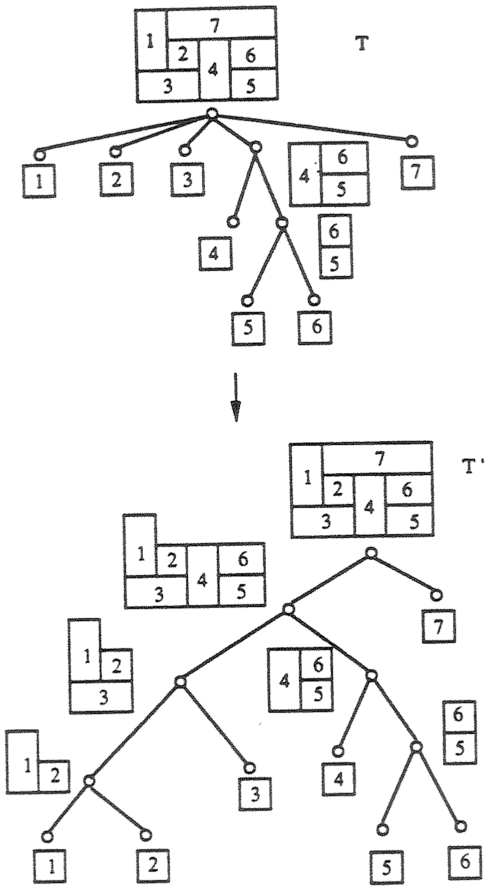

Figure 3: Restructuring T into a binary tree T'.

We have the following observation regarding the memory space requirement of [9]. Every time when the algorithm generates all non-redundant implementations for an internal node in T', all of them have to be stored as each such implementation is possible to be part of an optimal solution. Hence, for large floorplans[1], the numbers of non-redundant implementations of the higher level nodes could become very large and consequently the algorithm may fail due to insufficient memory space. In order to reduce memory usage, it is necessary to provide effective and efficient methods to control the number of non-redundant implementations for each internal node in T'. Our approach to meet this goal is as follows.

[1]In this paper the term large floorplan refers to either a floorplan with a large number of modules or a floorplan with a large number of non-redundant implementations for each module.

Every time after the algorithm generates all non-redundant implementations for an internal node, we check the number of non-redundant implementations of that internal node. If it exceeds a predefined limit, we reduce the number of implementations to within the limit by applying an implementation selection algorithm. Since each internal node represents either a rectangular block or an L-shaped block, two different implementation selection algorithms are needed. For a rectangular block, we design an algorithm which can properly select a subset of non-redundant implementations from the irreducible R-list. For an L-shaped block, since its non-redundant implementations are stored in a set of irreducible L-lists, we design an algorithm, which can properly select a subset of non-redundant implementations from an irreducible L-list, and sequentially apply it to each of the irreducible L-lists.

In the next section, we shall present two optimal implementation selection algorithms for rectangular and L-shaped blocks, respectively. Both algorithms run in polynomial time. The common key idea of the two algorithms is based on reducing the problem of optimally selecting a subset of non-redundant implementations from an irreducible list (R-list or L-list) to the constrained shortest path problem. As we shall see in next section, the constrained shortest path problem can be optimally solved in polynomial time.

4 Optimal Selection of Implementations

In this section, we first introduce the *Constrained Shortest Path Problem (CSPP)* and present a polynomial time optimal algorithm for this problem. Then, two algorithms based on reducing the problem of optimally selecting a subset of implementations from an irreducible R-list or L-list to the CSPP are introduced, respectively.

4.1 Constrained Shortest Path Problem

Let $G = (V, E)$ be a *weighted directed acyclic graph (DAG)*, where $V = \{v_1, v_2, ..., v_n\}$ and E are the vertex set and the edge set, respectively. For each edge $(v_i, v_j) \in E$, let $w(v_i, v_j)$ be its weight and we assume $w(v_i, v_j) > 0$. Given two vertices $s, t \in V$ and a positive integer $k \leq n$, the solution

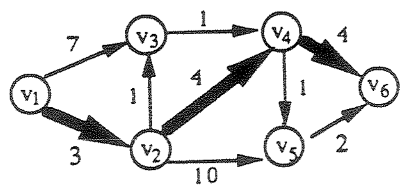

Figure 4: An example of the constrained shortest path problem.

to the CSPP is either a simple path of minimum total weight from s to t with exactly k vertices in G or an indication that no such path exists. Note that the CSPP is different from the classical shortest path problem in the sense that the number of vertices on the path is restricted to be exactly k. Figure 4 shows a weighted DAG in which the number on each edge represents the weight. It is easy to see that the shortest path from v_1 to v_6 is the path $v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow v_4 \rightarrow v_5 \rightarrow v_6$ with total weight of 8. However, if we consider the CSPP with $k = 4$, the constrained shortest path from v_1 to v_6 should be the path $v_1 \rightarrow v_2 \rightarrow v_4 \rightarrow v_6$ with total weight of 11. Note that there are two other paths from v_1 to v_6 with 4 vertices, i.e., $v_1 \rightarrow v_3 \rightarrow v_4 \rightarrow v_6$ and $v_1 \rightarrow v_2 \rightarrow v_5 \rightarrow v_6$, but their total weights, 12 and 15, are not minimum.

Let $W(s, v_i, l)$ be the minimum total weight of a constrained shortest path in G from s to v_i with exactly l vertices. For convenience, if there exists no such path from s to v_i with l vertices, we simply let $W(s, v_i, l) = \infty$. We now present a polynomial time optimal algorithm called Constrained_Shortest_Path to solve the CSPP as shown below. The algorithm Constrained_Shortest_Path is based on the dynamic programming technique.

```
Algorithm: Constrained_Shortest_Path
Input: A weighted DAG $G = (V, E)$, two vertices $s, t \in V$, and a positive integer $k \leq |V|$;
Output: The set of $k$ vertices on the constrained shortest path from $s$ to $t$ (if exists);
Begin
  /* Initialization */
  $W(s, s, 1) := 0$;
  for each vertex $v_i \in V - \{s\}$ do
    $W(s, v_i, 1) := \infty$;
```

```
end for
/* Calculate $W(s,t,k)$ */
for $l := 2$ to $k$ do
  for each vertex $v_i \in V - \{s\}$ do
    /* Calculate $W(s, v_i, l)$ */
    $W(s, v_i, l) := \infty$;
    for each edge $(v_j, v_i) \in E$ do
      if $W(s, v_j, l-1) + w(v_j, v_i) < W(s, v_i, l)$ then
        $W(s, v_i, l) := W(s, v_j, l-1) + w(v_j, v_i)$;
        update the vertex, i.e., $v_j$, which gives rise to the current $W(s, v_i, l)$;
      end if
    end for
  end for
end for
if $W(s,t,k) = \infty$ then
  return("Can not find such a path.");
else /* Determine the set of $k$ vertices on the constrained shortest path. */
  $P := \emptyset$; $v := t$; $l := k$;
  while $l > 2$ do
    $P := P \bigcup \{v_j\}$, where $v_j$ is associated with $W(s, v, l)$;
    $v := v_j$; $l := l - 1$;
  end while
  if $k \geq 2$
    $P := \{s\} \bigcup P \bigcup \{t\}$;
  else /* $k = 1$ and $s = t$ */
    $P := \{s\}$;
  end if
  return(P);
```

```
    end if
  End.
```

Theorem 1 *The algorithm Constrained_Shortest_Path correctly solves the CSPP in $O(k(|V|+|E|))$ time, where $k, |V|$, and $|E|$ are the number of vertices on the constrained shortest path, the number of vertices, and the number of edges, in G, respectively.*

Proof. To show the correctness of the algorithm Constrained_Shortest_Path, we first prove the computation of $W(s, v_i, l)$ is correct for all v_i and l with $v_i \in V - \{s\}$ when $1 < l \leq k$, and $v_i \in V$ when $l = 1$. The proof is by induction on l. When $l = 1$, there are two cases. The first case is that if $v_i = s$, it is true that $W(s, s, 1)$ is correctly set to 0. On the other hand, if $v_i \neq s$, $W(s, v_i, 1)$ is correctly set to ∞ since there is no path from s to v_i with only one vertex. Suppose the algorithm Constrained_Shortest_Path correctly calculates $W(s, v_i, l)$ for all v_i and l with $1 \leq l \leq q < k$. When $l = q + 1$, it is easy to see that for any $v_i \neq s$, we have

$$W(s, v_i, q+1) = \min(\infty, \min_{(v_j, v_i) \in E}(W(s, v_j, q) + w(v_j, v_i))). \tag{1}$$

Clearly, Equation (1) is implemented in the algorithm Constrained_Shortest_Path. Hence, based on the induction hypothesis, $W(s, v_i, q+1)$ can also be calculated correctly by the algorithm Constrained_Shortest_Path. Therefore, the computation of $W(s, v_i, l)$ is correct for any v_i and l with $1 \leq l \leq k$. Since the algorithm Constrained_Shortest_Path correctly calculates any $W(s, v_i, l)$ and also correctly records which vertex adjacent to v_i gives rise to such $W(s, v_i, l)$, it is not hard to see that an optimal path P of total weight $W(s, t, k)$ can be retrieved in linear time by the algorithm Constrained_Shortest_Path if $W(s, t, k) \neq \infty$ (i.e., P exists). On the other hand, if no such P exists, the algorithm returns "Can not find such a path.". Consequently, the algorithm Constrained_Shortest_Path is correct.

The time complexity of the algorithm is dominated by the computation of $W(s, v_i, l)$ for all v_i and l with $2 \leq l \leq k$. For each l, every vertex v_i other than s and every edge incident into v_i need to be examined once, and hence the computation of $W(s, v_i, l)$ for each l takes $O(|V| + |E|)$ time. Therefore, the overall time complexity of the algorithm Constrained_Shortest_Path should

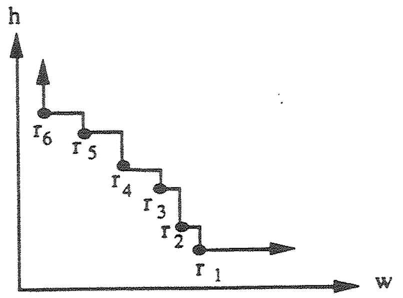

Figure 5: A staircase curve.

be $O(k(|V| + |E|)$ as there are $O(k)$ possible values for l. □

4.2 Rectangular Blocks

In this subsection, we present an algorithm to optimally select a subset of non-redundant implementations from an irreducible R-list. That is, given an irreducible R-list R storing n non-redundant implementations of a rectangular block B, and a positive integer $k < n$, the algorithm selects from R a subset of k implementations which can best approximate R.

Let $R = \{r_1, r_2, ..., r_n\}$ with each r_i equal to (w_i, h_i), and let $R' = \{r_{d_1}, r_{d_2}, ..., r_{d_k}\}$ be a subset of k implementations selected from R with $1 = d_1 < d_2 < ... < d_k = n$. We may assume $2 \leq k < n$. We can use a *staircase curve* C_R to represent R such that R constitutes the set of *corners* on C_R with r_1 and r_n being the rightmost and the leftmost corners, respectively. (Figure 5 shows the staircase curve C_R for $R = \{r_1, r_2, ..., r_6\}$.) Note that any point on or above C_R represents a *feasible* (either redundant or non-redundant) implementation of B, but only the corners are non-redundant. Similarly, we can use another staircase curve $C_{R'}$ to represent R', and any point on or above $C_{R'}$ is also a feasible implementation of B. Intuitively, we can consider the bounded area between C_R and $C_{R'}$ as the amount of feasible implementations discarded due to selecting R' from R. Hence, we define the cost $ERROR(R, R')$ of selecting R' from R as the bounded area between C_R and $C_{R'}$, and our objective is to select R' such that $ERROR(R, R')$ is minimized. Figure 6 shows an example in which $R = \{r_1, r_2, r_3, r_4, r_5, r_6\}$, $R' = \{r_1, r_3, r_4, r_6\}$, and by definition, $ERROR(R, R')$ is the sum of the areas of A_1 and A_2.

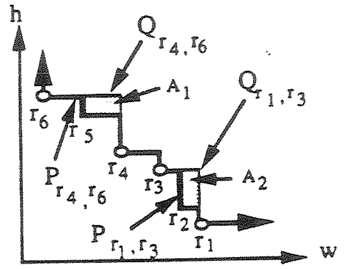

Figure 6: Illustration of bounded area between curves.

Clearly, $C_{R'}$ can be partitioned into a set of subcurves $Q_{r_{d_1},r_{d_2}}, Q_{r_{d_2},r_{d_3}}, \ldots, Q_{r_{d_{k-1}},r_{d_k}}$, where for all q, $1 \leq q < k$, subcurve $Q_{r_{d_q},r_{d_{q+1}}}$ has r_{d_q} and $r_{d_{q+1}}$ as its two endpoints. Similarly, for all q, $1 \leq q < k$, let $P_{r_{d_q},r_{d_{q+1}}}$ be the subcurve of C_R with r_{d_q} and $r_{d_{q+1}}$ as its two endpoints. Hence, in Figure 6, A_1 corresponds to the bounded area between Q_{r_4,r_6} and P_{r_4,r_6}, and A_2 corresponds to the bounded area between Q_{r_1,r_3} and P_{r_1,r_3}. If we define $error(r_{d_q}, r_{d_{q+1}})$ as the bounded area between $Q_{r_{d_q},r_{d_{q+1}}}$ and $P_{r_{d_q},r_{d_{q+1}}}$, we have

$$ERROR(R, R') = \sum_{q=1}^{k-1} error(r_{d_q}, r_{d_{q+1}}). \tag{2}$$

Equation (2) tells us that $ERROR(R, R')$ can be calculated in terms of $error(r_{d_q}, r_{d_{q+1}})$ for $q = 1, 2, \ldots, k-1$. We present below an $O(n^2)$ algorithm called Compute_R_Error to calculate $error(r_i, r_j)$ for all i and j, $1 \leq i < j \leq n$. Both the correctness and the time complexity of the algorithm Compute_R_Error can be easily proved, and hence the proof is omitted.

```
Algorithm: Compute_R_Error
Input: An irreducible R-list {$r_1 = (w_1, h_1), r_2 = (w_2, h_2), \ldots, r_n = (w_n, h_n)$};
Output: $error(r_i, r_j)$ for all $i$ and $j$, $1 \leq i < j \leq n$;
Begin
  for $i := 1$ to $n - 1$ do
    $error(r_i, r_{i+1}) := 0$;
  end for
  for $l := 2$ to $n - 1$ do
```

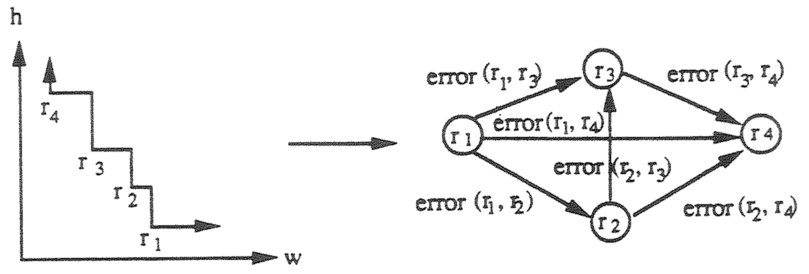

Figure 7: Constructing a weighted DAG.

```
    for $i := 1$ to $n - l$ do
      $error(r_i, r_{i+l}) := error(r_i, r_{i+l-1}) + (w_i - w_{i+l-1})(h_{i+l} - h_{i+l-1})$;
    end for
  end for
End.
```

We now describe how to reduce the problem of selecting an optimal R' from R to an instance of the CSPP. Given R, we construct a weighted DAG $G = (V, E)$ with $V = R$ and $E = \{(r_i, r_j) | \forall i, j, 1 \leq i < j \leq n\}$. Each edge (r_i, r_j) has its weight equal to $error(r_i, r_j)$. (See Figure 7 for an illustration of the construction.) Note that the construction takes $O(n^2)$ time. We have the following lemma.

Lemma 1 *Let R be an irreducible R-list, and let G be the weighted DAG constructed from R. We have, each path P in G from r_1 to r_n with k vertices corresponds to a subset of k implementations of R. Moreover, $ERROR(R, P)$ is equal to the total weight of P.*

Proof. Since G is a DAG and each vertex in G corresponds to an implementation in R, each path P in G from r_1 to r_n with k vertices clearly corresponds to a subset of k implementations of R. Suppose $P = \{r_{d_1}, r_{d_2}, ..., r_{d_k}\}$ with $d_1 = 1$ and $d_k = n$. Then, the total weight of P should be

$$\begin{aligned}\sum_{q=1}^{k-1} w(r_{d_q}, r_{d_{q+1}}) &= \sum_{q=1}^{k-1} error(r_{d_q}, r_{d_{q+1}}) \\ &= ERROR(R, P).\end{aligned}$$

□

It follows from Lemma 1 that the problem of selecting an optimal R' from R is equivalent to the CSPP for G with $s = r_1$ and $t = r_n$. The following algorithm R_Selection can optimally select a subset R' of k implementations from R such that $ERROR(R, R')$ is minimized.

```
Algorithm: R_Selection
Input: An irreducible R-list $R = \{r_1, ..., r_n\}$, and a positive integer $k < n$;
Output: An optimal subset $R'$ of $k$ implementations from $R$;
Begin
  Apply the algorithm Compute_R_Error to calculate $error(r_i, r_j)$ for all $i$ and $j$, $1 \leq i < j \leq n$;
  Construct the corresponding weighted DAG $G$ from $R$;
  Apply the algorithm Constrained_Shortest_Path to $G$ with $s = r_1$ and $t = r_n$;
End.
```

Theorem 2 *The algorithm R_Selection optimally selects a subset of k implementations from an irreducible R-list of n implementations in $O(kn^2)$ time.*

Proof. The correctness of the algorithm R_Selection follows directly from Lemma 1 and Theorem 1. Its time complexity is dominated by the algorithm Constrained_Shortest_Path with $|E| = O(n^2)$ and hence is $O(kn^2)$. □

4.3 L-shaped Blocks

In this subsection, we present an algorithm to optimally select a subset of non-redundant implementations from an irreducible L-list. Since each L-shaped block has a set of irreducible L-lists to store its non-redundant implementations, to reduce the number of non-redundant implementations of the L-shaped block, we can sequentially apply the algorithm to reduce the size of each irreducible L-list.

Let $L = \{l_1, l_2, ..., l_n\}$ be an irreducible L-list storing n implementations of an L-shaped block B, where $l_i = (w_{1,i}, w_{2,i}, h_{1,i}, h_{2,i})$ for all i, $1 \leq i \leq n$. Let $L' = \{l_{d_1}, l_{d_2}, ..., l_{d_k}\}$ be a subset of k implementations selected from L with $1 = d_1 < d_2 < \ldots < d_k = n$. Again, we may assume $2 \leq k < n$. Since each $l_i \in L$ can be viewed as a point in the 4-dimensional real space R^4, we use $dist(l_i, l_j)$ to denote the Manhattan distance[2] between l_i and l_j, for all $l_i, l_j \in L$. In other words, we have

$$dist(l_i, l_j) = |w_{1,i} - w_{1,j}| + |w_{2,i} - w_{2,j}| + |h_{1,i} - h_{1,j}| + |h_{2,i} - h_{2,j}|.$$

Since $w_{2,i} = w_{2,j}$, we have

$$dist(l_i, l_j) = |w_{1,i} - w_{1,j}| + |h_{1,i} - h_{1,j}| + |h_{2,i} - h_{2,j}|.$$

In general, we expect the less distance between l_i and l_j, the more similar in shape they are. Hence, $dist(l_i, l_j)$ measures the amount of difference in shape between l_i and l_j. For each discarded implementation $l_i \in L - L'$, we define $cost(l_i)$ as the distance between l_i and the implementation in L' which is closet to l_i. That is, for any $l_i \in L - L'$, we have

$$cost(l_i) = \min_{l_j \in L'} dist(l_i, l_j).$$

Intuitively, we can consider $cost(l_i)$ as the cost of not selecting l_i from L, and hence the implementation in L' which has the minimum distance from l_i is responsible for replacing l_i. Therefore, we define the total cost $ERROR(L, L')$ of selecting L' from L as the sum of the costs of the implementations not selected into L'. That is,

$$ERROR(L, L') = \sum_{l_i \in L - L'} cost(l_i).$$

Now, our goal is to select L' such that $ERROR(L, L')$ is minimized. Before describing how to determine an optimal L', we first state some important lemmas.

Lemma 2 *Let L be an irreducible L-list, and let $l_{i'}, l_i, l_j, l_{j'} \in L$ with $i' < i < j < j'$. We have $dist(l_i, l_j) \leq dist(l_{i'}, l_j)$ and $dist(l_i, l_j) \leq dist(l_i, l_{j'})$.*

Proof. Since L is an irreducible L-list and $l_{i'}, l_i, l_j, l_{j'} \in L$ with $i' < i < j < j'$, we can assume

$$l_{i'} = (w_{1,i'}, w_{2,i'}, h_{1,i'}, h_{2,i'}),$$

[2]In fact, we can use any L_p metric [6] to measure the distance. It is well known that the Manhattan distance is the L_1 metric. Note that all the lemmas and theorem presented in this subsection remain correct for any L_p metric.

$$l_i = (w_{1,i}, w_{2,i}, h_{1,i}, h_{2,i}),$$

$$l_j = (w_{1,j}, w_{2,j}, h_{1,j}, h_{2,j}),$$

$$l_{j'} = (w_{1,j'}, w_{2,j'}, h_{1,j'}, h_{2,j'})$$

with

$$w_{1,i'} \geq w_{1,i} \geq w_{1,j} \geq w_{1,j'},$$

$$w_{2,i'} = w_{2,i} = w_{2,j} = w_{2,j'},$$

$$h_{1,i'} \leq h_{1,i} \leq h_{1,j} \leq h_{1,j'},$$

$$h_{2,i'} \leq h_{2,i} \leq h_{2,j} \leq h_{2,j'}.$$

Therefore, we have

$$\begin{aligned} dist(l_i, l_j) &= |w_{1,i} - w_{1,j}| + |h_{1,i} - h_{1,j}| + |h_{2,i} - h_{2,j}| \\ &= (w_{1,i} - w_{1,j}) + (h_{1,j} - h_{1,i}) + (h_{2,j} - h_{2,i}) \\ &\leq (w_{1,i'} - w_{1,j}) + (h_{1,j} - h_{1,i'}) + (h_{2,j} - h_{2,i'}) \\ &= |w_{1,i'} - w_{1,j}| + |h_{1,i'} - h_{1,j}| + |h_{2,i'} - h_{2,j}| \\ &= dist(l_{i'}, l_j). \end{aligned}$$

Similarly, we have

$$\begin{aligned} dist(l_i, l_j) &\leq |w_{1,i} - w_{1,j'}| + |h_{1,i} - h_{1,j'}| + |h_{2,i} - h_{2,j'}| \\ &= dist(l_i, l_{j'}). \end{aligned}$$

□

For any discarded implementation $l_i \in L - L'$, there exist exactly two consecutive implementations $l_{d_q}, l_{d_{q+1}} \in L'$ with $d_q < i < d_{q+1}$. We define such l_{d_q} and $l_{d_{q+1}}$ as the *left neighbor* and the *right neighbor* of l_i, respectively. For example, if $L = \{l_1, l_2, l_3, l_4, l_5\}$ and $L' = \{l_1, l_2, l_5\}$, then the left and the right neighbors of l_3 and l_4 both are l_2 and l_5, respectively. We now present Lemma 3 which shows that $cost(l_i)$ can be computed only by considering the distance between l_i and its two neighbors.

Lemma 3 *Let L and $L' \subseteq L$ be two irreducible L-lists. For any $l_i \in L - L'$, if l_{d_q} and $l_{d_{q+1}}$ are its left and right neighbors, then $cost(l_i) = \min(dist(l_{d_q}, l_i), dist(l_i, l_{d_{q+1}}))$.*

Proof. By definition, we have

$$cost(l_i) = \min_{l_{d_p} \in L'} dist(l_i, l_{d_p}).$$

Based on Lemma 2, for any $l_{d_p} \in L'$ with $p < q$ (i.e., $d_p < d_q < i$), we have

$$dist(l_{d_q}, l_i) \leq dist(l_{d_p}, l_i),$$

and for any $l_{d_p} \in L'$ with $p > q + 1$ (i.e., $i \leq d_{q+1} \leq d_p$), we have

$$dist(l_i, l_{d_{q+1}}) \leq dist(l_i, l_{d_p}).$$

Hence,

$$cost(l_i) = \min_{l_{d_p} \in L'} dist(l_i, l_{d_p}) = \min(dist(l_{d_q}, l_i), dist(l_i, l_{d_{q+1}})).$$

□

For any two consecutive implementations $l_{d_q}, l_{d_{q+1}} \in L'$, let $X = \{l_i, l_{i+1}, ..., l_j\} \subseteq L - L'$ with $i = d_q + 1$ and $j = d_{q+1} - 1$. We define the cost of discarding X from L as

$$error(l_{d_q}, l_{d_{q+1}}) = \sum_{l_i \in X} cost(l_i),$$

where each $cost(l_i)$ can be computed just based on Lemma 3. Now, it is clear that $ERROR(L, L')$ can be computed in terms of $error(l_{d_q}, l_{d_{q+1}})$ for $q = 1, 2, ..., k - 1$, as follows.

$$ERROR(L, L') = \sum_{q=1}^{k-1} error(l_{d_q}, l_{d_{q+1}}). \tag{3}$$

We present below an $O(n^3)$ algorithm called Compute_L_Error to compute $error(l_i, l_j)$ for all i and j, $1 \leq i < j \leq n$. Again, both the correctness and the time complexity of the algorithm Compute_L_Error can be easily proved, and hence the proof is omitted here.

Algorithm: Compute_L_Error

Input: An irreducible L-list $L = \{l_1, l_2, ..., l_n\}$;

```
Output: $error(l_i, l_j)$, for all $i$ and $j$, $1 \le i < j \le n$;
Begin
  for $i := 1$ to $n - 1$ do
    for $j := i + 1$ to $n$ do
      $error(l_i, l_j) = 0$;
      for $q := i + 1$ to $j - 1$ do
        $error(l_i, l_j) := error(l_i, l_j) + \min(dist(l_i, l_q), dist(l_q, l_j))$;
      end for
    end for
  end for
End.
```

We now describe how to reduce the problem of determining an optimal L' from L into an instance of the CSPP. Given L, we construct a weighted DAG $G = (V, E)$ with $V = L$ and $E = \{(l_i, l_j) | \forall i, j, 1 \le i < j \le n\}$. Each edge (l_i, l_j) has its weight equal to $error(l_i, l_j)$. Note that the construction of G is very similar to that described in Section 4.2 except $error(l_i, l_j)$ is defined differently. We have the following lemma and its proof is similar to that of Lemma 1.

Lemma 4 *Let L be an irreducible L-list, and let G be the weighted DAG constructed from L. We have, each path P in G from l_1 to l_n with k vertices corresponds to a subset of k implementations of L. Moreover, $ERROR(L, P)$ is equal to the total weight of P.* □

It follows from Lemma 4 that the problem of determining an optimal L' from L is equivalent to the CSPP for G with $s = l_1$ and $t = l_n$. The following algorithm L_Selection can optimally select a subset L' of k implementations from L such that $ERROR(L, L')$ is minimized.

```
Algorithm: L_Selection
Input: An irreducible L-list $L = \{l_1, ..., l_n\}$, and a positive integer $k < n$;
Output: An optimal subset $L'$ of $k$ implementations from $L$;
Begin
  Apply the algorithm Compute_L_Error to compute $error(l_i, l_j)$ for all $i$ and $j$, $1 \le i < j \le n$;
```

```
    Construct the corresponding weighted DAG $G$ from $L$;
    Apply the algorithm Constrained_Shortest_Path to $G$ with $s = l_1$ and $t = l_n$;
End.
```

Theorem 3 *The algorithm L_Selection optimally selects a subset of k implementations from an irreducible L-list of n implementations in $O(n^3)$ time.*

Proof. The correctness of the algorithm L-Selection follows directly from Lemma 4 and Theorem 1. Its time complexity is dominated by the algorithm Compute_L_Error and hence is $O(n^3)$. □

Finally, we explain how to apply the algorithm L_Selection to each of the irreducible L-lists of an L-shaped block B. Suppose B has p irreducible L-lists storing a total of N implementations. Also suppose we want to reduce the number of implementations of B from N to K. Let L be one of the p L-lists, and $|L|$ be the number of implementations stored in L. When we apply the algorithm L_Selection to L, the limit on the number of implementations for L is set to $\lfloor \frac{K|L|}{N} \rfloor$. Consequently, the limit on the number of implementations for each of the p L-lists is dynamically adjusted.

5 Experimental Results

We have implemented the algorithms R_Selection and L_Selection in C language on a Sun SPARC station running Unix operating system. In addition, we have also incorporated the two algorithms into [9], and used 16 test examples which are based on the following 4 floorplans FP1–FP4 to test them. (For each of the 4 floorplans, we have 4 test examples corresponding to 4 different sets of modules.)

FP1: The 25-module floorplan as shown in Figure 8(a);

FP2: The 49-module floorplan as shown in Figure 8(b);

FP3: The 120-module floorplan as shown in Figure 8(d) in which each rectangular block B consists of the 24-module floorplan in Figure 8(c);

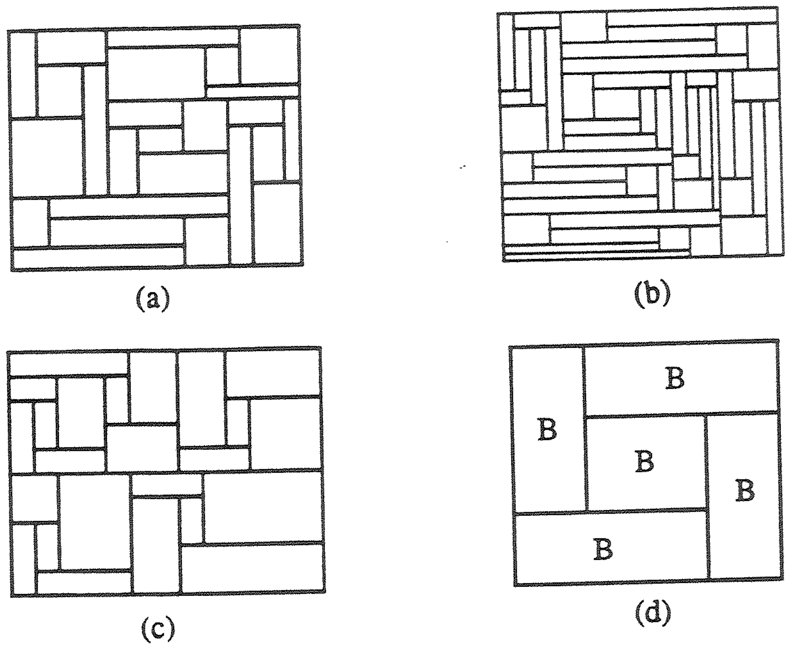

Figure 8: Test floorplans.

FP4: The 245-module floorplan as shown in Figure 8(d) in which each rectangular block B consists of the 49-module floorplan in Figure 8(b);

We first used the test examples of FP1–FP3 to measure the performance of the algorithm R_Selection. For each of the test examples, we ran R_Selection three times using different limits on the number of implementations for each rectangular block. The results are reported in Tables 1–3, where N is the number of non-redundant implementations of each module, K_1 is the user specified limit on the number of implementations for each rectangular block, M is the maximum number of implementations ever stored in memory during the computation[3], A_{OPT} is the optimal area of the floorplan produced by [9], and A_R is the area of the floorplan produced by [9] with R_Selection. It is clear that M and $\frac{A_R - A_{OPT}}{A_{OPT}}$ measure the memory usage and the quality of solutions produced by using R_Selection, respectively. As can be seen from Tables 1–3, for the test examples where [9] ran successfully, the algorithm R_Selection helped to significantly reduce both memory usage and running times while its solutions are very close to optimal. Furthermore, we observed that [9]

[3]Note that M is not equal to the total number of implementations ever generated because as soon as R_Selection eliminates some implementations, we no longer need to store them.

Test Case #	N	[9]		[9] + R_Selection			$\frac{A_R - A_{OPT}}{A_{OPT}}$
		M	CPU (sec)	K_1	M	CPU (sec)	
1	20	67871	16.2	20	15834	5.3	1.21%
				30	19607	7.4	0.99%
				40	26656	10.5	0.04%
2	20	139580	40.4	20	18143	5.7	0.46%
				30	25375	9.2	0.09%
				40	35950	14.3	0.09%
3	40	257944	181.0	40	52891	33.6	0.18%
				50	59142	38.8	0.18%
				60	72721	46.8	0.0%
4	40	402398	997.8	40	79810	50.6	0.05%
				50	84928	61.3	0.23%
				60	96672	75.6	0.23%

Table 1: The experimental results of FP1 (25 modules).

failed to run for the last three test examples of FP3 due to insufficient memory space, and hence incorporating the algorithm R_Selection into [9] becomes necessary.

Finally, we used the test examples of FP4 to test the performance of R_Selection and L_Selection. Note that [9] failed to run for each of these test examples. Table 4 reports the results produced by using R_Selection with and without L_Selection. In Table 4, K_2 is the user specified limit on the number of implementations for each L-shaped block, and A_{R+L} is the area of the floorplan produced by [9] with R_Selection and L_Selection. Clearly, $\frac{A_{R+L} - A_R}{A_R}$ measures the effectiveness of L_Selection. As shown in Table 4, for the first two test examples, the algorithm L_Selection helped to reduce memory usage while its running times and solution qualities remain comparable. However, for the last two test examples, using L_Selection is necessary to get solutions because R_Selection alone was unable to prevent the failure caused by insufficient memory space. Since we observed that the number of implementations of an L-shaped block in general is much larger than that of a rectangular block, it is much more expensive to use L_Selection than R_Selection. Therefore in our current implementation we employed two special techniques to speed up the computation. Firstly, instead of using L_Selection whenever the number X of implementations of an L-shaped block exceeds K_2, we use L_Selection only when $\frac{K_2}{X} < \theta$, for some user specified parameter θ, $0 < \theta \leq 1$. In other words, we only use L_Selection when X is sufficiently larger than

Test Case #	N	[9]		[9] + R_Selection			$\frac{A_R - A_{OPT}}{A_{OPT}}$
		M	CPU (sec)	K_1	M	CPU (sec)	
1	20	343603	76.8	20	45474	10.3	1.99%
				30	63025	15.3	0.28%
				40	90199	23.8	0.03%
2	20	495836	117.7	20	46541	11.0	0.27%
				30	59299	15.2	0.76%
				40	90548	23.0	0.19%
3	40	608891	187.8	40	230002	73.0	0.13%
				50	239653	81.4	0.1%
				60	252735	90.3	0.1%
4	40	473772	407.7	40	131751	51.7	0.15%
				50	142940	57.8	0.15%
				60	158288	65.5	0.0%

Table 2: The experimental results of FP2 (49 modules).

Test Case #	N	[9]		[9] + R_Selection			$\frac{A_R - A_{OPT}}{A_{OPT}}$
		M	CPU (sec)	K_1	M	CPU (sec)	
1	20	439234	90.3	20	50841	15.6	1.29%
				30	53514	20.8	0.4%
				40	57947	27.6	0.16%
2	20	> 806553	-	20	61566	37.0	-
				30	67717	51.1	-
				40	74211	65.7	-
3	40	> 798494	-	40	230891	170.5	-
				50	236818	192.1	-
				60	244656	221.3	-
4	40	> 800552	-	40	269127	210.8	-
				50	282360	241.3	-
				60	299268	284.9	-

Table 3: The experimental results of FP3 (120 modules).

Test Case #	N	[9] + R_Selection K_1	M	CPU (sec)	[9] + R_Selection + L_Selection K_2	M	CPU (sec)	$\frac{A_{R+L}-A_R}{A_R}$
1	20	40	379943	106.8	1000	232666	101.3	6.43%
					1500	239972	109.9	4.2%
					2000	292279	94.7	3.15%
2	20	40	462441	145.4	1000	209744	136.1	6.25%
					1500	255946	133.3	4.02%
					2000	268903	171.1	2.44%
3	40	40	> 782146	-	1000	341626	338.0	-
					1500	507107	455.5	-
					2000	600522	532.8	-
4	40	40	> 775390	-	1000	346208	341.7	-
					1500	458501	494.7	-
					2000	589706	552.2	-

Table 4: The experimental results of FP4 (245 modules).

K_2. Secondly, since L_Selection is extremely slow when X is very large, whenever $X > S$ for some user specified parameter S, we first use a heuristic version of L_Selection to reduce the number of implementations to S, and then use L_Selection to further reduce it to K_2 implementations.

The experimental results reported in Tables 1–4 indicate that the algorithm R_Selection is both efficient and effective in producing good quality solutions for most of the test examples. However, for very large floorplans, we need to use the algorithm L_Selection in addition to the algorithm R_Selection. Since L_Selection is much more expensive than R_Selection, we suggest to use R_Selection first, and only use L_Selection whenever R_Selection alone fails.

6 Concluding Remarks

We have presented in this paper two algorithms R_Selection and L_Selection to control the number of implementations generated for rectangular and L-shaped blocks, respectively. We also presented experimental results to show how effective and efficient the two algorithms are after incorporating them into [9].

Finally, we would like to point out other applications of the two implementation selection algorithms. Firstly, the two algorithms can be applied to other existing floorplan design algorithms (e.g., [3,5,10]) to reduce both memory usage and computation time. Secondly, if we consider the case where each module in the floorplan has an infinite set of implementations specified by a continuous shape curve, the floorplan area optimization problem can still be solved by first approximating each such curve by a large number of points and then applying [9] together with the two algorithms.

References

[1] D. P. Lapotin and S. W. Director, "Mason: A Global Floor-planning Tool," in *Proc. IEEE. Intl. Conf. on Computer-Aided Design,* 1985, pp. 143-145.

[2] U. Lauther, "A Min-Cut Placement Algorithm for General Cell Assemblies Based on a Graph Representation," *Journal of Digital Systems,* Vol. IV, Issue 1, pp. 21-34, 1980.

[3] T. Lengauer and R. Müller, "A Robust Framework for Hierarchical Floorplanning with Integrated Global Wiring," in *Proc. IEEE International Conf. on Computer-Aided Design,* 1990, pp. 148-151.

[4] R. H. J. M. Otten, "Automatic Floorplan Design," in *Proc. 19th ACM/IEEE Design Automation Conf.,* 1982, pp. 261-267.

[5] M. Pedram and B. Preas, "A Hierarchical Floorplanning Approach," in *Proc. IEEE International Conf. on Computer Design,* 1990, pp. 332-338.

[6] F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*, 1985.

[7] N. R. Quin and M. A. Breur, "A Force Directed Component Placement Procedure for Printed Circuit Boards," *IEEE Trans. on Circuits and Systems,* Vol. CAS-26, No. 6, pp. 377-388, 1979.

[8] L. Stockmeyer, "Optimal Orientations of Cells in Slicing Floorplan Designs," *Information and Control,* Vol. 59, pp. 91-101, 1983.

[9] T.-C. Wang and D. F. Wong, "An Optimal Algorithm for Floorplan Area Optimization", in *Proc. 27th ACM/IEEE Design Automation Conf.*, 1990, pp. 180-186.

[10] G. Zimmerman, "A New Area and Shape Function Estimation Technique for VLSI Layouts," in *Proc. 25th ACM/IEEE Design Automation Conf.*, 1988 pp. 60-65.