A Unified Method to Handle Different Kinds of Placement Constraints in Floorplan Design

*Evangeline F.Y. Young   **Chris C.N. Chu   *M.L. Ho

*Dept. of Computer Science and Engineering
The Chinese University of Hong Kong
New Territories, Hong Kong
fyyoung,mlho@cse.cuhk.edu.hk

**Dept. of Electrical and Computer Engineering
Iowa State University
Ames, IA 50011-3060
cnchu@iastate.edu

Abstract

In floorplan design, it is common that a designer will want to control the positions of some modules in the final packing for various purposes like data path alignment, I/O connection, etc.. There are several previous works [7, 9, 8, 4, 1] focusing on a few particular kinds of placement constraint. In this paper, we will present the first "unified method" to handle all of them simultaneously, including pre-placed constraint, range constraint, boundary constraint, alignment, abutment and clustering, etc., in general non-slicing floorplans. We have used incremental updates and an interesting idea of reduced graph to improve the runtime of the algorithm. We tested our method using some benchmark data with about one eighth of the modules having placement constraints and the results is very promising. Good packings with all the constraints satisfied can be obtained efficiently.

1. Introduction

Floorplan design is an important step in physical design of VLSI circuits to plan the positions of a set of circuit modules on a chip in order to optimize the circuit performance. In this floorplanning step, it is common that a designer will want to control the positions of some modules in the final packing for various reasons. For example, a designer may want to restrict the separation between two modules if there are a lot of interconnections between them, or he may want to align them vertically in the middle of the chip for better data path design, etc.. This will also happen in design re-use in which a designer may want to keep the positions of some modules unchanged in the new floorplan. Unfortunately, an effective method to control the absolute or relative positions between the modules in floorplan design is non-trivial and this inadequacy has limited the application and usefulness of many floorplanning algorithms in practice.

Some previous works have been done to handle some particular kinds of placement constraints. The paper [7] focuses on handling pre-placed constraint in which some modules are fixed in positions. The paper [9] works on boundary constraint in which some modules are constrained to be placed along one of the four sides of the chip for I/O connection. The paper [8] generalizes the approach in [7] to handle range constraint in which some modules are restricted to be placed within some rectangular ranges. Different approaches are used to handle different kinds of constraints and there is no unified method that can handle all of them simultaneously. Besides, all these previous works are based on a restricted type of floorplan representation, called *slicing* floorplan. A slicing floorplan is one that can be obtained by recursively cutting a rectangle into two parts by either a vertical line or a horizontal line. A *non-slicing* floorplan is one that is not necessarily slicing. Therefore, a non-slicing floorplan is more general and it can describe any type of packing. For non-slicing floorplan, there are only a few previous works [4, 1] that can handle pre-placed constraints.

In this paper, we will present the first "unified method" that can handle different kinds of placement constraints simultaneously, including pre-placed constraint, range constraint, boundary constraint, alignment, abutment and clustering, etc., in general non-slicing floorplans. The user can input a mixed set of constraints and the unified method will be able to handle all of them simultaneously. Our method makes use of constraint graphs to handle the constraints and can thus be used with any kind of floorplan representation that computes the module positions by constraint graphs, e.g., sequence pair, BSG, O-Tree, B*-Tree, etc.. We modify the constraint graphs to enforce the required constraints in the result packing. This is done by augmenting the graphs with positive, negative or zero weighted edges. These augmented edges will restrict the modules to be placed correctly according to the requirements. This technique of adding edges to constraint graphs has been used before for layout compaction [3] and packing of rectilinear blocks [2]. We found that this method could be generalized to handle different kinds of placement constraints simultaneously in floorplan design. However, a direct implementation of the original method is very expensive computationally and thus impractical. It will take $O(n^3)$ time for each iteration of the annealing process where n is the number of modules. We improved this runtime by using an interesting idea of reducing the size of the constraint graphs and by updating the constraint graphs incrementally.

We tested our method with some MCNC benchmarks (ami33, ami49 and playout) and a randomly generated data set with 100 modules. Ami33, ami49 and playout were chosen because they are the largest data sets (with 33, 49 and 62 modules respectively)

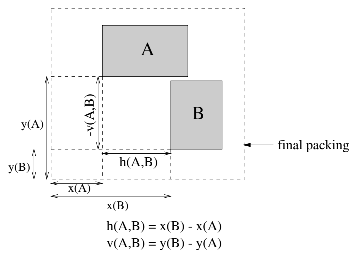

Figure 1. Notations $h(A,B)$ and $v(A,B)$.

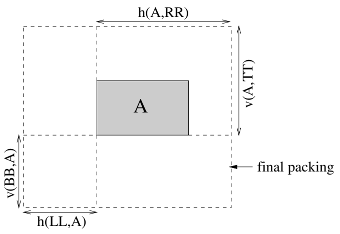

Figure 2. Notations $h(LL,A)$, $h(A,RR)$, $v(BB,A)$ and $v(A,TT)$.

among all the MCNC benchmarks. Sequence pair representation [5] is used in our implementation. The results is promising and a tight packing with all the constraints satisfied can be obtained efficiently. In the following sections, we will first describe the problem and have a brief review on the sequence pair representation and constraint graph. Section 4 will give a detailed explanation of our unified approach. Section 5 will explain the techniques to reduce the size of the constraint graphs and to update them incrementally. Experimental results will be given in Section 6.

2. Problem Definition

In floorplanning, we are given the information of a set of modules, including their areas and interconnection and our goal is to plan their positions on a chip to minimize the total chip area and interconnect cost. In this paper, we address this floorplanning problem with placement constraint, i.e., besides the module information, we are also given some constraints in placement between the modules and our goal is to plan their positions on the chip such that all the placement constraints can be satisfied and the area and interconnect cost are minimized.

We consider two general kinds of placement constraints, absolute and relative. For relative placement constraint, users can restrict the horizontal or vertical distance between two modules to a certain value, or to a certain range of values. We use the notation $h(A,B)$ to denote the horizontal displacement from A's lower left corner to B's. Note that this value is positive if B's lower left corner is on the right hand side of A's and is negative otherwise. We use $v(A,B)$ to denote the vertical displacement from A's lower left corner to B's. Similarly, this value is positive if B's lower left corner is above A's and is negative otherwise. (Figure 1 illustrates these definitions.) A relative placement constraint between two modules A and B can be written as:

$$h(A,B) = [\alpha,\beta] \quad \text{or} \quad v(A,B) = [\alpha,\beta]$$

where $\alpha,\beta \in \mathbf{R}$ and $\alpha \leq \beta$. When $\alpha = \beta$, we are restricting the distance between the two modules to a single value and we will write them simply as $h(A,B) = \alpha$ or $v(A,B) = \alpha$ respectively.

Absolute placement constraint is specified similarly except that one of the two modules in the relationship is a boundary of the chip. We use LL, RR, BB and TT to denote the left, right, bottom and top boundary of the chip respectively. Therefore notations $h(LL,A)$ and $h(A,RR)$ denote the horizontal distances of the lower left corner of A from the left and right boundary of the chip respectively. Similarly, we use $v(A,TT)$ and $v(BB,A)$ to denote the vertical distances of the lower left corner of A from the top and bottom boundary of the chip respectively. (Figure 2 illustrates these definitions.) An absolute placement constraint of a module A can be written as:

$$\begin{array}{ccc} h(LL,A) = [\alpha,\beta] & \text{or} & h(A,RR) = [\alpha,\beta] \quad \text{or} \\ v(BB,A) = [\alpha,\beta] & \text{or} & v(A,TT) = [\alpha,\beta] \end{array}$$

where $\alpha,\beta \in \mathbf{R}$ and $\alpha \leq \beta$. If $\alpha = \beta$, we are restricting the distance between the module and the boundary to a certain value and we will simply write it as $h(LL,A) = \alpha$, $h(A,RR) = \alpha$, $v(BB,A) = \alpha$ or $v(A,TT) = \alpha$ respectively.

These two types of specifications are general enough to express all common types of placement constraints. For example, if we want to restrict the placement of module A, B and C such that they all align horizontally, we can specify the following relative placement constraints:

$$v(A,B) = 0 \quad \wedge \quad v(B,C) = 0$$

As another example, if we want to restrict the placement of module A at the lower right corner of the chip, we can specify the following absolute placement constraints:

$$h(A,RR) = w_A \quad \wedge \quad v(BB,A) = 0$$

where w_A is the width of A. We can now define our floorplanning problem with placement constraint, **FP/PC**, as follows:

Problem FP/PC: *Given the information of a set of modules including their areas and interconnections, a set Γ_1 of relative placement constraints and a set Γ_2 of absolute placement constraints, the goal is to pack the modules in a rectangular region such that all the given placement constraints are satisfied and the area and interconnect costs are minimized.*

We assume that the input set of placement constraints will not be contradictory to each other, i.e., there exists a feasible packing in which all the constraints can be satisfied simultaneously. (If the input requirements are inherently inconsistent, the floorplanner will still generate a packing that satisfys the requirements as much as possible.)

3. Preliminaries

We use sequence-pair in our implementation to represent a general non-slicing floorplan. A sequence-pair of a set of modules is a pair of combinations of the module names. For example, $s = (abcd, bacd)$ is a sequence-pair of the module set $\{a,b,c,d\}$. We can derive the relative positions between the modules from these two combinations. and use a pair of constraint graphs to represent these relationships. A horizontal (vertical) constraint graph G_h (G_v) for a set of n modules is a directed graph with n vertices, the vertices represent the modules and the edges represent the horizontal (vertical) relationships between the module positions. We will have an edge from a to b labeled w_a in G_h where w_a is the width of a if and only if module b is on the right hand side of module a. Similarly, we will have an edge from a to b labeled h_a in G_v where h_a is the height of a if and only if module b is above module a. We can build these graphs directly from a sequence-pair representation s: insert an edge from a to b in G_h labeled w_a if and only if $s = (\,..\, a \,..\, b \,..,\, ..\, a \,..\, b \,..\,)$, and insert an edge from b to a in G_v labeled h_b if and only if $s = (\,..\, a \,..\, b \,..,\, ..\, b \,..\, a \,..\,)$. We can compute the positions of each module from the constraint graphs by putting the x-coordinate and y-coordinate of a module i as the length of the longest path from a source to i in the horizontal and vertical constraint graph respectively.

4. Handling Placement Constraints in Constraint Graphs

There are two kinds of placement constraints, relative and absolute. A relative placement constraint describes the relationship between two modules, while an absolute placement constraint describes the relationship between a module and the chip. We will first discuss the approach to handle relative placement constraint and will later discuss how this approach can be used to handle absolute placement constraint by making a simple modification to the constraint graphs.

4.1. Relative Placement Constraint

In relative placement constraint, users can restrict the horizontal or vertical distance between two modules to a certain range of values. For example, users can specify that $h(A, B) = [\alpha, \beta]$ (or $v(A, B) = [\alpha, \beta]$) where $\alpha, \beta \in \mathbf{R}$ and $\alpha \leq \beta$ meaning that B is at a distance of α to β on the right hand side of A (B is at a distance of α to β above A). When $\alpha = \beta$, we are restricting the distance to a certain value. Notice that both α and β can be zero, positive, negative, $+\infty$ or $-\infty$. (It is trivial to have $\alpha = -\infty$ and $\beta = +\infty$, so we assume that this will not happen.) In order to realize the required constraints in the final packing, we will add a single edge or a pair of edges to the corresponding constraint graph G as described below. We use $w(e)$ to denote the weight of an edge e.

Case 1 If $\alpha = -\infty$, insert an edge $e = (B, A)$ into G with $w(e) = -\beta$.

Case 2 If $\beta = +\infty$, insert an edge $e = (A, B)$ into G with $w(e) = \alpha$.

Case 3 Otherwise, insert two edges $e_1 = (A, B)$ and $e_2 = (B, A)$ into G s.t. $w(e_1) = \alpha$ and $w(e_2) = -\beta$.

Theorem 1 The relative placement constraint $h(A, B) = [\alpha, \beta]$ (or $v(A, B) = [\alpha, \beta]$) can be achieved in the final packing by inserting edges into the horizontal (vertical) constraint graph as described in the above cases if the packing is feasible.

Proof Without loss of generality, we only prove the correctness for the horizontal direction. The proof for the vertical direction will follow similarly. To prove the correctness of these steps, we need to show that if the packing is feasible after inserting these edges, the constraint $h(A, B) = [\alpha, \beta]$ will be satisfied in the packing. In the following, G_h denotes the horizontal constraint graph and $x(A)$ denotes the x-coordinate of the lower left corner of A. Assume that the packing is feasible, i.e., both constraint graphs have no positive cycle and the position of each module can be found by computing the longest paths from a source to its corresponding vertex in the two constraint graphs. We made the following two observations which follow simply from definition:

Observation 1 If there is an edge from A to B labeled s in G_h, $x(B) \in [x(A) + s, +\infty]$.

Observation 2 $x(B) \in [x(A) + s, +\infty]$ and $x(A) \in [-\infty, x(B) - s]$ are equivalent.

Consider the three different cases for the constraint $h(A, B) = [\alpha, \beta]$:

Case 1 $\alpha = -\infty$, i.e., we want $x(B)$ to lie in $[-\infty, x(A) + \beta]$. According to Observation 2, this condition is equivalent to $x(A) \in [x(B) - \beta, +\infty]$, which, by Observation 1, can be achieved by inserting an edge from B to A labeled $-\beta$.

Case 2 $\beta = +\infty$, i.e., we want B to lie in $[x(A) + \alpha, +\infty]$. According to Observation 1, this can be achieved by inserting an edge from A to B labeled α.

Case 3 $-\infty < \alpha \leq \beta < +\infty$, i.e., we want B to lie in the range $[x(A) + \alpha, x(A) + \beta]$. Notice that $[x(A) + \alpha, x(A) + \beta]$ is equivalent to $[x(A) + \alpha, +\infty] \wedge [-\infty, x(A) + \beta]$. The first condition can be achieved by inserting an edge from A to B labeled α. The second condition $x(B) \in [-\infty, x(A) + \beta]$ is equivalent to $x(A) \in [x(B) - \beta, +\infty]$ according to Observation 2 and this can be achieved by inserting an edge from B to A labeled $-\beta$. Therefore we need to insert a pair of edges, one from A to B labeled α and another one from B to A labeled $-\beta$. Q.E.D.

4.2. Absolute Placement Constraint

Absolute placement constraint restricts the absolute placement of a module with respect to the whole chip. Users can restrict the placement of a module such that its distance from the boundary of the chip is within a certain range of values. We can handle this kind of constraint using a method similar to that for relative placement constraints, i.e., by inserting a single edge or a pair of edges to the constraint graphs. To achieve this, we augment the horizontal and vertical constraint graphs each with two extra nodes. For the horizontal constraint graph, we add two nodes: one is a source with zero weighted out-going edges to all the other nodes, and the other one is a sink with zero weighted in-coming edges from all the other nodes. The source represents the left boundary and the sink represents the right boundary of the final packing. Similarly, we add two nodes to the vertical constraint graph: one is a source with zero weighted out-going edges to all the other nodes and one is a sink with zero weighted in-coming edges from all the other nodes. The source represents the bottom boundary and the sink represents the top boundary of the final packing.

In the following, we use v_l and v_r to denote the two additional nodes in the horizontal constraint graph: v_l represents the left boundary and v_r represents the right boundary. Similarly, we use v_t and v_b to denote the two additional nodes in the vertical constraint graph: v_t represents the top boundary and v_b represents the bottom boundary. After adding these nodes, we can handle absolute placement constraint easily as described below. Notice that there is no such case for $h(A, LL)$, $h(RR, A)$, $v(A, BB)$ nor $v(TT, A)$ and α and β are non-negative numbers because we will not consider packing modules outside the boundary of the chip. In the following, we have only shown the cases in the horizontal direction. The other two cases in the vertical direction can be derived similarly.

- $h(LL, A) \in [\alpha, \beta]$: If $\beta = +\infty$, insert an edge $e_1 = (v_l, A)$ in G_h with $w(e_1) = \alpha$; else, insert edges $e_1 = (v_l, A)$ and $e_2 = (A, v_l)$ in G_h with $w(e_1) = \alpha$ and $w(e_2) = -\beta$.
- $h(A, RR) \in [\alpha, \beta]$: If $\beta = +\infty$, insert an edge $e_1 = (A, v_r)$ in G_h with $w(e_1) = \alpha$; else, insert edges $e_1 = (A, v_r)$ and $e_2 = (v_r, A)$ in G_h with $w(e_1) = \alpha$ and $w(e_2) = -\beta$.

The proof of correctness for absolute placement constraint follows directly from that for relative placement constraint and we will not repeat it here.

4.3. Examples of some Commonly Used Placement Constraint

Using the above specifications for absolute and relative placement constraint, we can describe many different kinds of placement constraints. In this section, we will pick a few commonly used ones and show how each can be specified using a combination of the relative and absolute placement constraints. In the following, we use $x(A)$ and $y(A)$ to denote the x and y coordinates of the lower left corner of module A respectively and we use h_A and w_A to denote the height and width of A respectively.

4.3.1 Alignment

To align module A, B, C and D horizontally (Figure 3(a)), we can impose the following constraints:

$$v(A, B) = 0 \ \wedge \ v(B, C) = 0 \ \wedge \ v(C, D) = 0$$

We restrict the vertical distances between these modules to be zero, they will thus all align horizontally. Six additional edges will be inserted into the vertical constraint graph.

4.3.2 Abutment

To abut module A, B and C horizontally (Figure 3(b)), we can impose the following constraints:

$$\begin{array}{lll} v(A, B) = 0 & \wedge & v(B, C) = 0 \quad \wedge \\ h(A, B) = w_A & \wedge & h(B, C) = w_B \end{array}$$

where w_A and w_B are the widths of module A and B respectively. In this formulation, the vertical distances between these modules are zero, so they will align horizontally. On the other hand, B is restricted to be on the right hand side of A by w_A units and C on the right hand side of B by w_B units, so they will be abutting with each other horizontally. Four additional edges will be inserted into each constraint graph.

4.3.3 Preplace Constraint

To preplace module A with its lower left corner at (p, q) (Figure 4(a)), we can impose the following constraints:

$$h(LL, A) = p \ \wedge \ v(BB, A) = q$$

We restrict $x(A)$ to be p units from the left boundary and $y(A)$ to be q units from the bottom boundary, so A will be preplaced with its lower left corner at (p, q) in the final packing. Two additional edges will be inserted into each constraint graph.

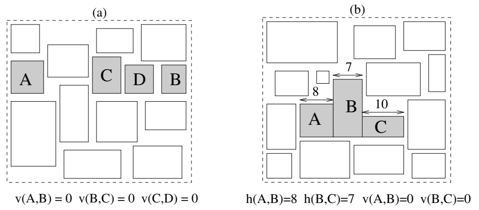
(a)
A
C
D
B
v(A,B) = 0 v(B,C) = 0 v(C,D) = 0
(b)
7
8
A
B
10
C
h(A,B)=8 h(B,C)=7 v(A,B)=0 v(B,C)=0

Figure 3. Examples of Alignment and Abutment Constraints

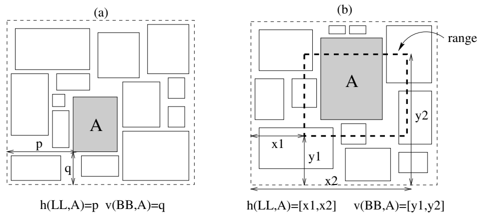

Figure 4. Examples of Preplace and Range Constraints

4.3.4 Range Constraint

To restrict the position of A to within the range $\{(x, y)|x_1 \leq x \leq x_2, y_1 \leq y \leq y_2\}$ (Figure 4(b)), we can impose the following constraints:

$$h(LL, A) = [x_1, x_2] \ \wedge \ v(BB, A) = [y_1, y_2]$$

In this formulation, we restrict $x(A)$ to be x_1 to x_2 units from the left boundary and $y(A)$ to be y_1 to y_2 units from the bottom boundary, so A will lie in the required range $\{(x, y)|x_1 \leq x \leq x_2, y_1 \leq y \leq y_2\}$. Two additional edges will be inserted into each constraint graph.

4.3.5 Boundary Constraint

To place module A at the upper right corner of the final packing, and place B along the top boundary (Figure 5(a)), we can impose the following constraints:

$$h(A, RR) = w_A \wedge \ v(A, TT) = h_A \wedge \ v(B, TT) = h_B$$

In this formulation, we restrict the horizontal distance between A and the right boundary to be the width of A and the vertical distance between A and the top boundary to be the height of A, so module A will be placed at the upper right corner in the final packing. Besides, B is restricted to be h_B units from the top boundary, so B will abut with the top boundary as required. We need to insert two edges into the horizontal constraint graph and four edges into the vertical constraint graph.

4.3.6 Clustering

To cluster module A, B and C around D at a distance of at most p units away vertically or horizontally (Figure 5(b)), we can impose the following constraints:

$$\begin{array}{lll} h(D, A) = [-p, +p] & \wedge & h(D, B) = [-p, +p] \quad \wedge \\ h(D, C) = [-p, +p] & \wedge & v(D, A) = [-p, +p] \quad \wedge \\ v(D, B) = [-p, +p] & \wedge & v(D, C) = [-p, +p] \end{array}$$

In this formulation, we restrict the horizontal and vertical distances of A, B and C from D to be at most p units in both directions, so they will cluster around D at a distance of at most p units away. Six additional edges will be inserted into each constraint graph.

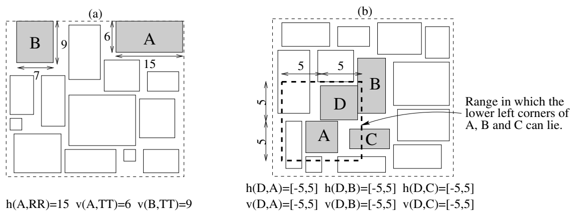
(a)
B
9
6
A
7
15
h(A,RR)=15 v(A,TT)=6 v(B,TT)=9
(b)
5
5
B
D
A
C
5
5
Range in which the lower left corners of A, B and C can lie.
h(D,A)=[-5,5] h(D,B)=[-5,5] h(D,C)=[-5,5]
v(D,A)=[-5,5] v(D,B)=[-5,5] v(D,C)=[-5,5]

Figure 5. Examples of Boundary and Clustering Constraints

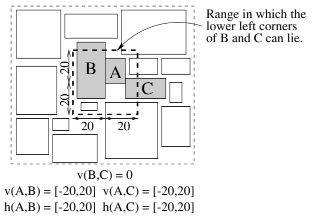
Range in which the lower left corners of B and C can lie.
B
A
C
20
20
20
20
v(B,C) = 0
v(A,B) = [-20,20] v(A,C) = [-20,20]
h(A,B) = [-20,20] h(A,C) = [-20,20]

Figure 6. An Example of Arbitrarily Mixed Constraints

4.3.7 General Placement Constraint

We can use combinations of the above relative and absolute placement constraints to specify all mixed constraints in general. For example, to restrict the placement such that module B and C align with each other horizontally and they cluster around A at a distance of at most 20 units away (Figure 6), we can impose the following constraints:

$$v(B,C) = 0 \ \wedge$$
$$v(A,B) = [-20,+20] \wedge \ v(A,C) = [-20,+20] \ \wedge$$
$$h(A,B) = [-20,+20] \wedge \ h(A,C) = [-20,+20]$$

The first constraint align B and C horizontally and the next four cluster B and C around A to within a distance of 20 units away. We need to add four additional edges to the horizontal constraint graph and six to the vertical constraint graph.

5. Algorithm and Implementation

We use simulated annealing with sequence pair representation. In each step of the annealing process, we will augment the constraint graphs with edges as described in the above section. We call these edges *constraining edges*. However it is possible that some constraints cannot be satisfied after adding these constraining edges, the packing is then *infeasible*. Feasibility of a packing can be checked by detecting positive cycles in the constraint graphs. If a packing is infeasible, we will pack the modules as if there is no placement constraint and compute a penalty term in the cost function to penalize the violations. This strategy ensures that all feasible solutions are reachable, and can drive the packing solution to one that satisfys the constraints as much as possible in case the user requirements are inherently inconsistent. We observed a stable convergency in the annealing process using this scheme and all the placement constraints can be satisfied at the end of the annealing process in all our experiments. We will describe the algorithm in details in the following sub-sections.

5.1. Detecting Positive Cycles by Reduced Graphs

After augmenting those constraint graphs with the constraining edges, we need to test their feasibility by detecting positive cycles in them. A direct implementation of some classical algorithm (e.g., the modified Floyd-Warshall algorithm [6]) to check positive cycles will take $O(n^3)$ time where n is the total number of modules. In order to improve the timing complexity, we will reduce the size of the constraint graphs before checking for cycles. This is possible because of the following observation. We use $G_h(V, E_h)$ and $G_v(V, E_v)$ to denote the original horizontal and vertical constraint graphs respectively. $G'_h(V, E'_h)$ and $G'_v(V, E'_v)$ are obtained from G_h and G_v respectively by adding the constraining edges.

Observation 3 Any cycle in G'_h (G'_v) must contain some edges in $E'_h - E_h$ ($E'_v - E_v$).

This observation is true because the original constraint graphs G_h and G_v obtained from the sequence pair representation must be acyclic. Therefore, any cycle in G'_h and G'_v must contain at least one constraining edge. From this observation, we can infer that any cycle in G'_h (G'_v) must contain at least two modules which have placement constraints. Therefore, instead of detecting positive cycles in G'_h and G'_v, we will construct two reduced graphs $H_h(V^*, E^*_h)$ and $H_v(V^*, E^*_v)$ from G_h and G_v respectively where V^* is the set of all modules with placement constraints, E^*_h is the set of all edges $\{e(i,j) | i,j \in V^* \wedge w(e) = d_{G_h}(i,j)\}$ where $d_{G_h}(i,j)$ denotes the longest path from i to j in G_h and E^*_v is the set of all edges $\{e(i,j) | i,j \in V^* \wedge w(e) = d_{G_v}(i,j)\}$ where $d_{G_v}(i,j)$ denotes the longest path from i to j in G_v. The constraining edges will be inserted into H_h and H_v to give H'_h and H'_v respectively. We will then check for positive cycles in H'_h and H'_v and this is equivalent to checking cycles in G'_h and G'_v according to the following theorem.

Theorem 2 A positive cycle exists in H'_h (H'_v) if and only if a positive cycle exists in G'_h (G'_v).

We will skip the proof here due to the lack of space. Constructing H_h and H_v takes $O(mn^2 + c)$ time where c is the total number of constraining edges and m is number of modules with placement constraints. This can be done by performing a single-source-longest-path in G_h and G_v once for each v where $v \in V^*$. Checking cycles in H'_h and H'_v by the modified Floyd-Warshall algorithm [6] takes $O(m(m^2 + c))$ time because $m^2 + c$ is an upper bound on the number of edges in H'_h and H'_v. This timing complexity can be further reduced in practice by performing incremental updates as described in the next sub-section.

5.2. Moves and Incremental Updates

In every iteration of the annealing process, we will modify the sequence pair by one of the following three kinds of moves:

[M1] Change the width and height of a module.
[M2] Exchange two modules in both sequences.
[M3] Exchange two modules in the first sequence.

The constraint graphs will not change much after each move, so we do not need to reconstruct them once in every iteration. We can take advantage of this incremental updates in two different places: construction of G_h and G_v, and construction of H_h and H_v.

5.2.1 Incremental Updates of G_h and G_v

In move M1, a module A is picked and changed in its width and height, so the structures of the constraint graphs will remain the same except that all the out-going edges from A will have their weights changed. In our implementation, the weights on the edges are stored at the source vertex because all the edges out-going from the same vertex will have the same weight. Therefore, we only need to update the weight of vertex A in both G_h and G_v after M1 and this will take constant time. In move M2, two modules A and B are picked and switched in positions in both sequences. The structure of the constraint graphs will again remain the same except that the vertices corresponding to A and B will switch in position. This will affect the weights of the out-going edges from these two vertices. Therefore we only need to update the weights in these two vertices in both G_h and G_v and this will again take constant time. In move M3, two modules A and B are picked and switched in positions in the first sequence. The structure of the constraint graphs will change after this move. However, only those modules lying between A and B in the first sequence will be affected and there are $\frac{n}{3}$ of them on the average. Besides, each update can be done very efficiently (either an edge $e(i,j)$ in G_v is deleted and a new edge $e(i,j)$ is inserted into G_h, or an edge $e(i,j)$ in G_h is deleted and a new edge $e(i,j)$ is inserted into G_v). Therefore, G_h and G_v can be updated very efficiently in $O(n)$ time.

5.2.2 Incremental Updates of H_h and H_v

H_h and H_v are obtained from G_h and G_v by keeping only those vertices with placement constraints. The weight of an edge $e(i,j)$ in H_h (H_v) is obtained from the longest path from i to j in G_h (G_v). After move M1, M2 or M3 of the annealing process, the edge weights in H_h and H_v may change because the longest path between two vertices in G_h and G_v will have changed. Fortunately this will only affect a fraction of the edges in H_h and H_v.

In move M1, a module A is selected and changed in width and height. The weight of an edge $e(i,j)$ in H_h or H_v will be affected if i can reach A in the constraint graphs G_h or G_v. This happens if i is lying before A in the second sequence and there are $\frac{m-1}{2}$ of them on the average. We need to perform once the single-source-longest-path algorithm in G_h or G_v for each of them and update the weights of all the edges $e(i,j)$ in H_h or H_v for all $j \in V^*$. In M2 and M3, two modules A and B are selected and switched in positions in the sequence pair. Similarly, an edge $e(i,j)$ in H_h or H_v will be affected if i can reach A or B in G_h or G_v before or after the move. This happens if i is lying before A or B in the second sequence and there are about $\frac{m-2}{2}$ of them on the average. Similarly, we need to perform once the single-source-longest-path algorithm for each of these affected modules and update the weights of the corresponding edges in H_h and H_v. Therefore updating H_h and H_v takes $O(mn^2)$ time with a very small constant factor in front.

5.3. Time Complexity

In each step of the annealing process, we modify the sequence pair by performing move M1, M2 or M3. After the move, we need to update G_h, G_v, H_h and H_v. Updating G_h and G_v takes $O(n)$ as explained above. Updating H_h and H_v takes $O(mn^2)$ time in the worst case. After updating these graphs, we need to check for positive cycles in H_h' and H_v' which are obtained from H_h and H_v respectively by inserting the constraining edges. The cycle checking step takes $O(m(m^2+c))$ time. Therefore the total time taken is $O(n+mn^2+m(m^2+c))$, i.e., $O(mn^2)$.

5.4. Annealing Schedule and Cost Function

The temperature schedule of the annealing process is of the form $T(k) = rT(k-1)$ for all $k \geq 1$. At each temperature step, enough number of moves are attempted until the total number of moves exceeds a certain number N where N is a user defined constant. The temperature is initialized to a large value at the beginning and the annealing process terminates when the temperature is low enough. The best solution found will then be used to go through a "final baking" process in which only better solutions will be accepted.

The cost function is defined as $A + \lambda W + \gamma P$ where A is the total area of the packing. In our current implementation, W is the half perimeter estimation of the interconnect cost but this term can be replaced by other more sophisticated interconnect cost estimations. P is a penalty term which is zero when all the placement constraints are satisfied, and is otherwise the sum of squares of the violations.

5.5. Handling Infeasible Packings

If a packing is infeasible, i.e., positive cycles exist in the constraint graphs, we will pack the modules as if there is no constraint and compute a penalty term P. For example, if an edge $e = (A,B)$ labeled α is inserted into the horizontal constraint graph because of a given placement constraint, the penalty term due to this edge in case of an infeasible packing will be $(\min\{x(B) - x(A) - \alpha, 0\})^2$. This gives a good estimation of how far the modules are from their desired positions. Note that we need to accept infeasible intermediate solutions in the annealing process because it may happen in some cases that a good feasible solution can only be reached from an initial starting point with some infeasible intermediate solutions in between during the searching process.

6. Experimental Results

We tested our floorplanner on a set of MCNC benchmark data (ami33, ami49 and playout) and a randomly generated data set with 100 modules. For each experiment, the temperature is set to 1.5×10^6 initially and is lowered at a constant rate of 0.95 to 0.98 until it is below 1×10^{-10}. The number of iterations at one temperature step is 80. λ in the cost function is set such that the costs of wirelength and total area are approximately equal. γ is set at a high value (30 to 40) to ensure that all the placement constraints can be satisfied at the end. All the experiments were carried out on a 400 MHz Sun Ultra III.

We tested our floorplanner using the benchmark data and a randomly generated data set by imposing different combinations of placement constraints to the modules. The result is shown in Table 1. For each data set, the result reported in each row is an average of six experiments using three different sets of placement constraints. We can see from the table that the algorithm is indeed very efficient. The percentage deadspace ranges from 5.9% to 8.4% and all the placement constraints can be satisfied in all the experiments. Figure 7 and 8 show five result packings of the benchmark data. (Note that the origin $(0, 0)$ is at the upper right corner in all these packings.)

We have also compared our results with [8] (using 300MHz Pentium II) that focuses on handling range constraint in slicing floorplan. We repeat the same experiments on range constraint using our new unified method and the results is shown in Table 2. The result reported in each row is an average of five different experiments using the benchmark data, ami33, ami49 and playout. We can see that our unified method is faster although the method in [8] can give packings with smaller deadspace sometimes. This is because they allow the modules to have flexible shape within the aspect ratio range of $[0.25, 4.0]$ while we consider only a discrete number of shapes for each module.

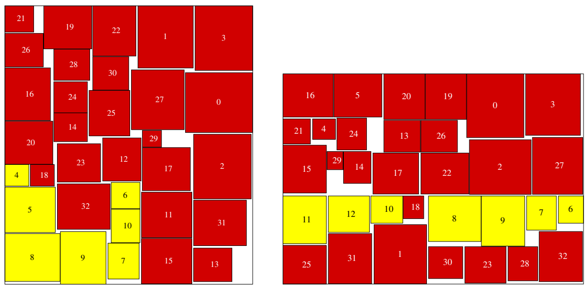

Figure 7. Left: Module 4, 5, 6, 7, 8, 9 and 10 cluster at the lower left corner of the chip. Right: Module 6, 7, 8, 9, 10, 11 and 12 align horizontally.

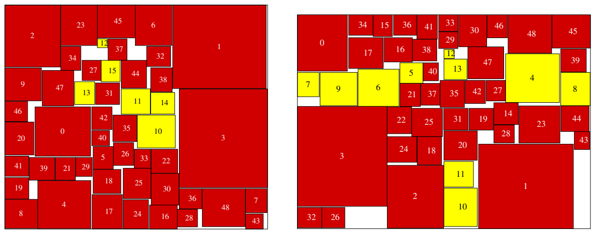

Figure 8. Left: Module 10, 12, 13, 14 and 15 cluster around module 11. Right: Module 4, 5, 6, 7, 8 and 9 almost align horizontally, and module 10, 11, 12 and 13 align vertically.

# Constraints	Time (sec)	Deadspace %	# Constraints	Time (sec)	Deadspace %
ami33 (#module = 33; #net = 123)			playout (#module = 64; #net = 1611)		
4	18.05	6.57	4	31.33	7.21
8	20.04	6.60	10	36.18	6.61
12	19.89	7.56	16	41.77	7.27
14	25.63	7.26	20	41.77	7.96
16	24.05	6.99	24	45.95	8.35
ami49 (#module = 49; #net = 408)			random100 (#module = 100; #net = 1611)		
4	34.69	6.29	4	289.03	8.06
8	36.90	7.37	10	310.61	8.06
12	38.63	7.75	16	325.51	7.41
16	39.53	7.48	22	357.39	7.98
20	42.37	6.08	28	374.71	6.92

Table 1. Results for ami33, ami49, playout and random100. All placement constraints were satistied in each experiment.

	[8]		Our Method	
Data Set	Deadspace(%)	Time (sec)	Deadspace(%)	Time (sec)
ami33	1.56	53.85	2.95	43.62
ami49	3.14	118.02	2.92	65.41
playout	3.00	230.85	2.76	109.46

Table 2. Comparisons with the Results in [8]

References

[1] Y. Chang, Y. Chang, G. Wu, and S. Wu. B*-Trees: A New Representation for Non-Slicing Floorplans. *Proceedings of the 37th ACM/IEEE Design Automation Conference*, 2000.

[2] K. Fujiyoshi and H. Murata. Arbitrary Convex and Concave Rectilinear Block Packing Using Sequence-Pair. *International Symposium on Physical Design*, pages 103–110, 1999.

[3] Y.-Z. Liao and C. Wong. An Algorithm to Compact a VLSI Symbolic Layout with Mixed Constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2(2):62–69, 1983.

[4] H. Murata, K. Fujiyoushi, and M. Kaneko. VLSI/PCB Placement with Obstacles Based on Sequence-Pair. *International Symposium on Physical Design*, pages 26–31, 1997.

[5] H. Murata, K. Fujiyoushi, S. Nakatake, and Y. Kajitani. Rectangle-Packing-Based Module Placement. *Proceedings IEEE International Conference on Computer-Aided Design*, pages 472–479, 1995.

[6] Thomas H. Cormen and Charles E. Leiserson and Ronald L. Rivest. *Introduction to Algorithms*. McGraw Hill, eighth edition, 1992.

[7] F. Young and D. Wong. Slicing Floorplans with Pre-placed Modules. *Proceedings IEEE International Conference on Computer-Aided Design*, pages 252–258, 1998.

[8] F. Young and D. Wong. Slicing Floorplans with Range Constraints. *International Symposium on Physical Design*, pages 97–102, 1999.

[9] F. Young, D. Wong, and H. H. Yang. Slicing Floorplans with Boundary Constraints. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 18(9):1385–1389, 1999.