MB*-tree: A Multilevel Floorplanner for Large-Scale Building-Module Design

Hsun-Cheng Lee, Yao-Wen Chang, Jer-Ming Hsu, and Hannah H. Yang

Synopsys Inc., Taipei, Taiwan
Department of Electrical Engineering & Graduate Institute of Electronics Engineering, National Taiwan University, Taipei 106, Taiwan
National Center for High-Performance Computing, Hsinchu, Taiwan
Strategic CAD Labs, Intel Corporation, Hillsboro, OR 97124, USA

Abstract

We present in this paper a multilevel floorplanning/placement framework based on the B*-tree representation, called MB*-tree, to handle the floorplanning and packing for large-scale building modules. The MB*-tree adopts a two-stage technique, clustering followed by declustering. The clustering stage iteratively groups a set of modules based on a cost metric guided by area utilization and module connectivity, and at the same time establishes the geometric relations for the newly clustered modules by constructing a corresponding B*-tree for them. The declustering stage iteratively ungroups a set of the previously clustered modules (i.e., perform tree expansion) and then refines the floorplanning/placement solution by using a simulated annealing scheme. In particular, the MB*-tree preserves the geometric relations among modules during declustering, which makes the MB*-tree an ideal data structure for the multilevel floorplanning/placement framework. Experimental results show that the MB*-tree obtains significantly better silicon area and wirelength than previous works. Further, unlike previous works, MB*-tree scales very well as the circuit size increases.

Keywords: floorplanning, placement, physical design, layout, multilevel framework

1 Introduction

Design complexities are growing at a breathtaking speed with the continued improvement of the nanometer IC technologies. On one hand, designs with hundreds of million transistors are already in production (ICs with billions of transistors are even expected within this decade), IP modules are widely reused, and a large number of buffer blocks are used for delay optimization as well as noise reduction in very deep-submicron interconnect-driven floorplanning [1, 7, 12, 14, 22], which all drive the need of a tool to handle large-scale building modules. On the other hand, the highly competitive IC market requires faster design convergence, faster incremental design turnaround, and better silicon area utilization. Efficient and effective design methodology and tools capable of placing and optimizing large-scale modules are essential for such large designs.

Many floorplan representations have been proposed [5, 9, 15, 16, 17, 19, 20, 21, 23] in the literature. However, traditional floorplanning/placement algorithms do not scale well as the design size, complexity, and constraints increase, mainly due to their inflexibility in handling non-slicing floorplans, and/or intrinsically non-hierarchical data structures (representations). The B*-tree, in contrast, has been shown an efficient, effective, and flexible data structure for non-slicing floorplans [5]. It is particularly suitable for representing a non-slicing floorplan with large-scale modules and for creating or incrementally updating a floorplan. What is more important, its binary-tree based structure directly corresponds to the framework of a hierarchical, divide-and-conquer scheme, and thus the properties inherited from the structure can substantially facilitate the operations for multilevel large-scale building module floorplanning/placement.

Based on the B*-tree representation, we present in this paper a multilevel floorplanning/placement framework, called MB*-tree, to handle the floorplanning and packing for large-scale building modules with high efficiency and quality. MB*-tree is inspired by the success of the multilevel framework in graph/circuit partitioning such as Chaco [10], hMetis [13], and ML [2], placement

*This work was partially supported by the National Science Council of Taiwan by Grant No. NSC-91-2215-E-002-038. E-mails: hclee@synopsys.com; ywcharg@cc.ee.ntu.edu.tw; gis89530@cis.nctu.edu.tw; hyang@ichips.intel.com.

such as MPL [4], and routing such as MRS [6], MR [18], and MARS [8]. Unlike multilevel partitioners and placers, however, multilevel floorplanning poses unique difficulties as the shapes of modules to be clustered together can significantly affect the area utilization of a floorplan, and a floorplan design within a cluster needs to be explored along with the global floorplan optimization. The clustering approach also helps to directly address floorplan congestion and timing issues, since different clustering algorithms can be developed to localize inter-module communication and reduce critical path length.

The MB*-tree algorithm adopts a two-stage technique, clustering followed by declustering. (See Figure 1 for an illustration of the multilevel framework.) The clustering stage iteratively groups a set of modules (could be basic modules and/or previously clustered modules) based on a cost metric guided by area utilization and module connectivity, and at the same time establishes the geometric relations for the newly clustered modules by constructing a corresponding B*-tree. The clustering procedure repeats until a single cluster containing all modules is formed, denoted by a one-node B*-tree that bookkeeps the entire multilevel clustering information. For soft modules, we apply Lagrangian relaxation during clustering to determine the module shapes. Then, the declustering stage iteratively ungroups a set of the previously clustered modules (i.e., expanding a node into a subtree according to the B*-tree topology constructed at the clustering stage), and then apply simulated annealing to refine the floorplanning/placement solution based on a cost metric defined by area utilization and wirelength. The refinement shall lead to a “better” B*-tree structure that guides the declustering at the next level. It is important to note that we always keep only one B*-tree structure for processing at each iteration, and the MB*-tree preserves the geometric relations among modules during declustering (i.e., the tree expansion), which makes the MB*-tree an ideal data structure for the multilevel floorplanning/placement framework.

Experimental results show that the MB*-tree scales very well as the circuit size increases while the famous previous works, sequence pair, O-tree, and B*-tree alone, do not. For circuit sizes ranging from 49 to 9,800 modules and from 408 to 81,600 nets, the MB*-tree consistently obtains high-quality floorplans with dead spaces of less than 3.7% in empirically linear runtime, while sequence pair, O-tree, and B*-tree can handle only up to 196, 196, and 1,960 modules in the same amount of runtime and result in the dead spaces of as large as 13.00% (@ 196 modules), 9.86% (@ 196 modules), and 27.33% (@ 1960 modules), respectively. We also performed experiments based on a large industrial design with 189 modules and 9777 nets. The results show that our MB*-tree algorithm obtained significantly better silicon area and wirelength than previous works.

The remainder of this paper is organized as follows. Section 2 formulates the module floorplanning/placement problem. Section 3 gives a brief overview on the B*-tree representation. Section 4 presents our two-stage algorithm, clustering followed by declustering, for the problem addressed in this paper. Section 5 presents our approach for handling soft modules. Section 6 gives the experimental results, and finally the concluding remarks are given in Section 7.

2 Problem Formulation

Let $M = \{m_1, m_2, \ldots, m_n\}$ be a set of $n$ rectangular modules. Each module $m_i \in M$ is associated with a three tuple $(h_i, w_i, a_i)$, where $h_i$, $w_i$, and $a_i$ denote the width, height, and aspect ratio of $m_i$, respectively. The area $A_i$ of $m_i$ is given
by $h_i/w_i$, and the aspect ratio $a_i$ of $m_i$ is given by $h_i/w_i$. Let $r_{i,\min}$ and $r_{i,\max}$ be the minimum and maximum aspect ratios, i.e., $h_i/w_i \in [r_{i,\min}, r_{i,\max}]$. A placement (floorplan) $P = \{(x_i,y_i)\mid m_i \in M\}$ is an assignment of rectangular modules $m_i$'s with the coordinates of their bottom-left corners being assigned to $(x_i,y_i)$'s so that no two modules overlap (and $h_i/w_i \in [r_{i,\min}, r_{i,\max}], \forall i$). We consider in this paper both hard and soft modules. A hard module is not flexible in its shape but free to rotate. A soft module is free to rotate and change its shape within the range $[r_{i,\min}, r_{i,\max}]$. The objective of placement/floorplanning is to minimize a specified cost metric such as a combination of the area and wirelength induced by the assignment of $m_i$'s, where $A_{tot}$ is measured by the final enclosing rectangle of $P$ and $W_{tot}$ the summation of half the bounding box of pins for each net (or the center-to-center interconnections among all modules).

3 Review of the B*-tree Representation

As mentioned earlier, we apply the B*-tree representation to handle the problem of multilevel large-scale building-module floorplanning/placement. Thus, we shall first give a review of the B*-tree representation.

Given a compacted placement $P$ that can neither move down nor move left (called an admissible placement [9]), we can represent it by a unique B*-tree $T$ [5]. (See Figure 2(b) for the B*-tree representing the placement of Figure 2(a).) A B*-tree is an ordered binary tree (a restriction of O-tree with faster and more flexible operations) whose root corresponds to the module on the bottom-left corner. Using the depth-first search (DFS) procedure, the B*-tree $T$ for an admissible placement $P$ can be constructed in a recursive fashion. Starting from the root, we first recursively construct the left subtree and then the right subtree. Let $R_i$ denote the set of modules located on the right-hand side and adjacent to $m_i$. The left child of the node $n_i$ corresponds to the lowest module in $R_i$ that is unvisited. The right child of $n_i$ represents the lowest module located above $m_i$, with its $x$-coordinate equal to that of $m_i$.

![Figure 2](image-url)  
Figure 2: An admissible placement and its corresponding B*-tree.

As shown in Figure 2, we make $n_1$ the root of $T$ since $m_1$ is on the bottom-left corner. Constructing the left subtree of $n_1$ recursively, we make $n_2$ the left child of $n_1$. Since the left child of $n_2$ does not exist, we then construct the right subtree of $n_2$ (which is rooted by $n_3$). The construction is recursively performed in the DFS order. After completing the left subtree of $n_1$, the same procedure applies to the right subtree of $n_1$.

Figure 2(b) illustrates the resulting B*-tree for the placement shown in Figure 2(a). The construction takes only linear time. The B*-tree keeps the geometric relationship between two modules as follows. If node $n_j$ is the left child of node $n_i$, module $m_j$ must be located on the right-hand side of $m_i$, with $x_j = x_i + w_i$. Besides, if node $n_j$ is the right child of $n_i$, module $m_j$ must be located above module $m_i$, with the $x$-coordinate of $m_j$ equal to that of $m_i$; i.e., $x_j = x_i$. Also, since the root of $T$ represents the bottom-left module, the coordinate of the module is $(x_{root}, y_{root}) = (0,0)$.

Inheriting from the nice properties of ordered binary trees, the B*-tree is simple, efficient, effective, and flexible for handling non-slicing floorplans. It is particularly suitable for representing a non-slicing floorplan with various types of modules and for
creating or incrementally updating a floorplan. What is more important, its binary-tree based structure directly corresponds to the framework of a hierarchical scheme, which makes it a superior data structure for multilevel large-scale building module floorplanning/placement.

4 The MB*-tree Algorithm

In this section, we shall present our MB*-tree algorithm for multilevel large-scale building module floorplanning/placement. As mentioned earlier, the algorithm adopts a two-stage approach, clustering followed by declustering, by using the B*-tree representation.

The clustering operation results in two types of modules, namely primitive modules and cluster modules. A primitive module \( m \) is a module given as an input (i.e., \( m \in M \)) while a cluster one is created by grouping two or more primitive modules. Each cluster module is created by a clustering scheme \( \{m_i, m_j\} \), where \( m_i \) (\( m_j \)) denotes a primitive or a cluster module. Figure 3 shows a cluster module with four primitive modules; a possible way to form the cluster module is by the clustering scheme \( \{\{m_1, m_2\}, \{m_3, m_4\}\} \).

![Diagram of a cluster with four primitive modules](image)

Figure 3: A cluster with the four primitive modules, \( a, b, c, \) and \( d \). The placement can be obtained by applying the clustering scheme \( \{\{m_1, m_2\}, \{m_3, m_4\}\} \), resulting in a dead space of 36 units.

We outline the two-stage approach, clustering followed by declustering, as follows.

1. Clustering: The clustering stage iteratively groups a set of (primitive or cluster) modules (say, two modules) based on a cost metric defined by area utilization, wirelength, and connectivity among modules, and at the same time establishes the geometric relations among the newly clustered modules by constructing a corresponding B*-subtree. The clustering procedure repeats until a single cluster containing all modules is formed (or the number of modules is smaller than a predefined threshold), denoted by a one-node B*-tree that bookkeeps the entire clustering scheme.

2. Declustering: The declustering stage iteratively ungroups a set of previously clustered modules (i.e., expanding a node into a subtree according to the B*-tree topology constructed at the clustering stage) and then refines the floorplan solution based on a simulated annealing scheme. The refinement shall lead to a “better” B*-tree structure that guides the declustering at the next level. It is important to note that we always keep only one B*-tree for processing at each iteration, and the multilevel B*-tree based floorlanner preserves the geometric relations among modules during declustering (i.e., the tree expansion), which makes the B*-tree an ideal data structure for the multilevel floorplanning framework.

In the following subsections, we detail the clustering and declustering algorithms for hard modules.

4.1 Clustering

In this stage, we iteratively group a set of (primitive or cluster) modules until a single cluster is formed (or until the number of cluster modules is smaller than a threshold) based on a cost metric of area and connectivity. We shall first consider the clustering metric.
The clustering metric is defined by the two criteria: area utilization (dead space) and the connectivity density among modules.

- Dead space: The area utilization for clustering two modules $m_i$ and $m_j$ can be measured by the resulting dead space $s_{ij}$, representing the unused area after clustering $m_i$ and $m_j$. Let $s_{tot}$ denote the dead space in the final floorplan $P$. We have $s_{tot} = A_{tot} - \sum_{m_i \in M} A_i$, where $A_i$ denotes the area of module $m_i$ and $A_{tot}$ the area of the final enclosing rectangle of $P$. Since $\sum_{m_i \in M} A_i$ is a constant, minimizing $A_{tot}$ is equivalent to minimizing the dead space $s_{tot}$. For the example shown in Figure 3, $s_{12} = 0$, $s_{13} = 36$, and $s_{tot} = 36$.

- Connectivity density: Let the connectivity $c_{ij}$ denote the number of nets between two modules $m_i$ and $m_j$. The connectivity density $d_{ij}$ between two (primitive or cluster) modules $m_i$ and $m_j$ is given by

$$d_{ij} = c_{ij}/(n_i + n_j),$$

where $n_i$ ($n_j$) denotes the number of primitive modules in $m_i$ ($m_j$). Often a bigger cluster implies a larger number of connections. The connectivity density considers not only the connectivity but also the sizes of clusters between two modules to avoid possible biases. For the example shown in Figure 4, we apply the clustering scheme $\{\{m_1, m_2\}, \{m_3, m_4\}\}$ (based on connectivity density), instead of $\{\{m_1, m_2\}, m_3, m_4\}$ (based on connectivity).

Obviously, the cost function of dead space is for area optimization while that of connectivity density is for timing and wiring area optimization. Therefore, the metric for clustering two (primitive or cluster) modules $m_i$ and $m_j$, $\phi : \{m_i, m_j\} \rightarrow \mathbb{R^+} \cup \{0\}$, is then given by

$$\phi(\{m_i, m_j\}) = \alpha \delta_{ij} + \frac{\beta K}{\delta_{ij}},$$

where $\delta_{ij}$ and $K/\delta_{ij}$ are respective normalized costs for $s_{ij}$ and $K/d_{ij}$, $\alpha$, $\beta$ and $K$ are user-specified parameters/ constants.

![Figure 4: An example connectivity between each pair of modules. We apply the clustering scheme $\{\{m_1, m_2\}, \{m_3, m_4\}\}$ based on connectivity density, instead of $\{\{m_1, m_2\}, m_3, m_4\}$ (based on connectivity).](image)

Based on $\phi$, we cluster a set of modules into one at each iteration by applying the aforementioned methods until a single cluster containing all primitive modules is formed or the number of modules is smaller than a given threshold (and thus can be easily handled by the classical program). During clustering, we shall record how two modules $m_i$ and $m_j$ are clustered into a new cluster module $m_k$. Figure 6 shows two ways to cluster two modules $m_i$ and $m_j$. If $m_i$ is placed left to (below) $m_j$, then $m_i$ is horizontally (vertically) related to $m_j$, denoted by $m_i \rightarrow (\uparrow)m_j$. If $m_i \rightarrow (\uparrow)m_j$, then $n_j$ is the left (right) child of $n_i$ in its corresponding B*-tree. The relation for each pair of modules in a cluster is established and recorded in the corresponding B*-subtree during clustering. It will be used for determining how to expand a node into a corresponding B*-subtree during declustering.

Figure 5 shows our 2-way clustering algorithm. Line 1 computes the initial cost matrix $\Phi = (\phi_{ij})$, where $\phi_{ij} = \alpha \delta_{ij} + \beta K/\delta_{ij}$. Line 2 assigns to $n$ the number of the input primitive modules. Lines 3–9 perform clustering step by step ($n - 1$ steps in total). At Step $k$, we pick two modules $m_i$ and $m_j$ with the minimum $\phi_{ij}$ (Extract_Min($\phi_{ij}$) in Line 4) and then cluster them into a new
cluster module $m_{n+k}$ (cluster($m_i, m_j$) in Line 5). Line 6 records the clustering scheme $q_k$ for $\{m_i, m_j\}$. Line 7 randomly decides the relation of $m_i$ and $m_j$ and constructs the corresponding the B*-subtree. We then update the set of declustered modules (Line 8) and the entries associated with $m_{n+k}$ in the cost matrix $\Phi$ (Line 9). We repeat the 2-way clustering process $n-1$ times until all modules are clustered into a single cluster. The clustering scheme $q_{n-1}$ for the last two modules bookkeeps the entire clustering scheme $Q$. Thus, we assign $q_{n-1}$ to $Q$ (Line 10) and return the entire scheme (Line 11).

Algorithm: Clustering($\mathcal{M}, \alpha, \beta, K$)

**Input:** $\mathcal{M}$—the set of primitive modules;

**Output:** $Q$—the clustering scheme.

1. Compute the cost matrix, $\Phi = \langle \phi_{ij} \rangle$, where $\phi_{ij} = \alpha s_{ij} + \frac{\beta K}{d_{ij}}$.
2. $n \leftarrow |\mathcal{M}|$;
3. for $k \leftarrow 1$ to $n-1$ do
4. Extract $\text{Min}(\phi_{ij})$;
5. $m_{n+k} \leftarrow \text{cluster}(m_i, m_j)$;
6. $q_k \leftarrow \{m_i, m_j\}$;
7. Randomly decide the relation of $m_i$ and $m_j$ and construct the corresponding B*-subtree;
8. $\mathcal{M} \leftarrow \mathcal{M} \cup \{m_{n+k}\}\backslash\{m_i, m_j\}$;
9. Update $\Phi$ whose entries are associated with $m_{n+k}$;
10. $Q \leftarrow q_{n-1}$;
11. return $Q$;

Figure 5: The 2-way clustering algorithm.

Figure 6: The relation of two modules and their clustering. (a) Two candidate modules $m_i$ and $m_j$. (b) The clustering and the corresponding B*-subtree for the case where $m_i$ is horizontally related to $m_j$. (c) The clustering and the corresponding B*-subtree for the case where $m_i$ is vertically related to $m_j$.

### 4.2 Declustering

We shall first introduce the metric used in simulated annealing for refining floorplan/placement solutions. The declustering metric is defined by the two criteria: area utilization (dead space) and the wirelength among modules.

- Dead space: Same as that defined in Section 4.1.

- Wire length: The wirelength of a net is measured by half the bounding box of all the pins of the net, or by the length of the center-to-center interconnections between the modules if no pin positions are specified. The wirelength for clustering two modules $m_i$ and $m_j$, $w_{ij}$, is measured by the total wirelength interconnecting the two modules. The total wirelength in the final floorplan $P$, $w_{total}$, is the summation of the length of the wires interconnecting all modules.
Obviously, the cost function of dead space is for area optimization while that of wirelength is for timing and wiring area optimization. Therefore, the metric for refining a floorplan solution during declustering, \( \psi_{ij} : \{m_i, m_j\} \rightarrow \mathbb{R}^+ \cup \{0\} \), is then given by

\[
\psi_{ij} = \gamma \delta_{ij} + \delta \hat{w}_{ij},
\]

where \( \delta_{ij} \) and \( \hat{w}_{ij} \) are respective normalized costs for \( s_{ij} \) and \( w_{ij} \), and \( \gamma \) and \( \delta \) are user-specified parameters.

---

**Algorithm: Declustering\( (m_k, m_i, m_j) \)**

**Input:** \( m_k \)—the cluster module; 
\( m_i, m_j \)—two modules with \( m_i \) right to or below \( m_j \);

1. \( \text{parent}(n_i) \leftarrow \text{parent}(n_k); \)
2. if \( (n_k = \text{left}(\text{parent}(n_k))) \) then
   3. \( \text{left}(\text{parent}(n_k)) \leftarrow n_i; \)
4. else
5. \( \text{right}(\text{parent}(n_k)) \leftarrow n_i; \)
6. if \( (m_i \rightarrow m_j) \) then
7. \( \text{left}(n_i) \leftarrow n_j; \text{parent}(n_j) \leftarrow n_i; \text{right}(n_j) \leftarrow \text{NIL}; \)
8. \( \text{right}(n_i) \leftarrow \text{right}(n_k); \)
9. if \( (\text{right}(n_k) \neq \text{NIL}) \) then
10. \( \text{parent}(\text{right}(n_k)) \leftarrow n_i; \)
11. \( \text{left}(n_j) \leftarrow \text{left}(n_k); \)
12. if \( (\text{left}(n_k) \neq \text{NIL}) \) then
13. \( \text{parent}(\text{left}(n_k)) \leftarrow n_j; \)
14. if \( (m_i \uparrow m_j) \) then
15. \( \text{right}(n_i) \leftarrow n_j; \text{parent}(n_j) \leftarrow n_i; \)
16. \( \text{right}(n_j) \leftarrow \text{right}(n_k); \)
17. if \( (\text{right}(n_k) \neq \text{NIL}) \) then
18. \( \text{parent}(\text{right}(n_k)) \leftarrow n_j; \)
19. let \( n_a \in \{m_i, m_j\} \) and \( a \neq b \) such that \( h_a \geq h_b; \)
20. \( \text{left}(n_a) \leftarrow \text{left}(n_k); \)
21. if \( (\text{left}(n_k) \neq \text{NIL}) \) then
22. \( \text{parent}(\text{left}(n_k)) \leftarrow n_a; \)
23. \( \text{left}(n_k) \leftarrow \text{NIL}; \)

Figure 7: The declustering algorithm.

The declustering stage iteratively ungroups a set of previously clustered modules (i.e., expand a node into a subtree according to the B*-tree constructed at the clustering stage) and then refines the floorplan solution based on simulated annealing.

Figure 7 shows the algorithm for declustering a cluster module \( m_k \) into two modules \( m_i \) and \( m_j \) that are clustered into \( m_k \) at the clustering stage. Without loss of generality, we make \( m_i \) right to or below \( m_j \). In Algorithm Declustering (see Figure 7), \( \text{parent}(n_i), \text{right}(n_i), \) and \( \text{left}(n_i) \) denote the parent, the right child, and the left child of node \( n_i \) in a B*-tree, respectively. Line 1 updates the parent of \( n_k \) as that of \( n_i \). Lines 2-5 make \( n_i \) a left (right) child if \( n_k \) is a left (right) child. Lines 6-13 deal with the case where \( m_i \) is horizontally related to \( m_j \). If \( m_i \rightarrow m_j \), then \( n_j \) is the left child of \( n_i \) and thus we update the corresponding links in Line 7. Lines 8–10 (11–13) update the links associated the right (left) child of \( n_k \). Similarly, lines 14–23 cope with the case where \( m_i \) is vertically related to \( m_j \).

Figure 8 gives an illustration of this algorithm. Figure 8(a) shows an instance of clustering and its corresponding B*-tree, for which we are preparing to decluster \( m_3 \) into \( m_6 \) and \( m_7 \) (i.e., the clustering scheme for \( m_3 \) is \( \{m_6, m_7\} \)). Figure 8(b) shows four
cases to decluster \( m_3 \), and their corresponding resulting B*-trees are illustrated in Figure 8(c). Cases 1 and 2 correspond to Line 6–13 of Figure 7, and Cases 3 and 4 correspond to Line 14–23.

Figure 8: Declustering \( m_3 \) into \( m_6 \) and \( m_7 \). (a) The configuration before the declustering. (b) Four cases to decluster \( m_6 \) and \( m_7 \). (c) The placement and corresponding B*-tree topology after the declustering.

**Theorem 1** Each declustering operation takes \( O(1) \) time, and the overall declustering stage takes \( O(M) \) time, where \( |M| \) is the number of input primitive modules.

**Proof:** As listed in Algorithm Declustering (see Figure 7), each declustering operation requires updating only local links associated with the three involved modules \((m_1, m_j, \text{ and } m_k)\). Since there are only a constant number of such links, performing a declustering operation takes \( O(1) \) time. Further, it is trivial that we perform \(|M| - 1\) declustering operations to ungroup all modules, and the overall declustering complexity thus follows.

### 4.3 Simulated Annealing

We proposed a simulated annealing based algorithm to refine the solution at each level of declustering. We apply the following three operations to perturb a multilevel B*-tree (a feasible solution) to another.

- **Op1:** Rotate a module.
- **Op2:** Move a module to another place.
- **Op3:** Swap two modules.

Op1 exchanges the width and height of a module. Op2 deletes a node of a B*-tree and inserts it into another position. Op3 deletes two nodes and inserts them into the corresponding positions in the B*-tree. Obviously, Op2 and Op3 need to perform the deletion and insertion operations on a B*-tree, which takes \( O(h) \) time, where \( h \) is the height of the B*-tree.

The annealing procedure uses a parameter, temperature \( t \), to control the probability of accepting an uphill move (an inferior solution). The initial temperature \( t_0 = \Delta_{\text{avg}}/\ln(P) \), where \( \Delta_{\text{avg}} \) is the average cost change for a set of randomly generated uphill moves, and \( P \) is the initial probability of accepting uphill moves. The temperature \( t \) is then decreased by a factor \( r < 1 \) (i.e., the temperature for the next iteration is \( rt \)). We terminate the annealing process when the temperature cools down to a user defined value \( \varepsilon \).
Figure 9: (a) Given seven modules, $m_i$'s, $1 \leq i \leq 7$. (b) Cluster $m_5$, $m_6$, and $m_7$ into $m_8$. (c) Cluster $m_1$, $m_2$, and $m_4$ into $m_9$. (d) Cluster $m_3$, $m_8$, and $m_9$ into $m_{10}$. (e) Decluster $m_{10}$ to $m_5$, $m_8$, and $m_9$. (f) Perform Op2 for $m_8$. (g) Decluster $m_9$ to $m_1$, $m_2$, and $m_4$. (h) Perform Op1 and Op2 for $m_2$ and $m_3$, respectively. (i) Decluster $m_8$ to $m_5$, $m_6$, and $m_7$. (j) Perform Op2 for $m_4$. 
The simulated annealing algorithm starts by a B*-tree produced during declustering. Then it perturbs a B*-tree (a feasible solution) to another B*-tree by Op1, Op2, and/or Op3 until a predefined “frozen” state is reached. At last, we transform the resulting B*-tree to the corresponding final admissible placement.

### 4.4 The Overall MB*-tree Algorithm

The MB*-tree algorithm integrates the aforementioned three algorithms and is summarized in Figure 10. In Line 1, We first perform clustering to reduce the problem size level by level and then enter the declustering stage. In the declustering stage, we perform floorplanning for the modules at each level using the simulated annealing based algorithm B*-tree $\text{SA}$.

Figure 9 illustrates an execution of the MB*-tree algorithm. For explanation, we cluster three modules each time in Figure 9. Figure 9(a) lists seven modules to be packed, $m_i$'s, $1 \leq i \leq 7$. Figures 9(b)–(d) illustrate the execution of the clustering algorithm. Figures 9(b) shows the resulting configuration after clustering $m_5$, $m_6$, and $m_7$ into a new cluster module $m_8$ (i.e., the clustering scheme of $m_8$ is $\{m_5, m_6, m_7\}$). Similarly, we cluster $m_1$, $m_2$, and $m_4$ into $m_9$ by using the clustering scheme $\{m_2, m_4, m_1\}$. Finally, we cluster $m_3$, $m_8$, and $m_9$ into $m_{10}$ by using the clustering scheme $\{m_3, m_8, m_9\}$. The clustering stage is thus done, and the declustering stage begins, in which simulated annealing is applied to do the floorplanning. In Figure 9(e), we first decluster $m_{10}$ into $m_3$, $m_8$, and $m_9$ (i.e., expand the node $n_{10}$ into the B*-subtree illustrated in Figure 9(e)). We then move $m_8$ to the top of $m_9$ (perform Op2 for $m_8$) during simulated annealing (see Figure 9(f)). As shown in Figure 9(g), we further decluster $m_9$ into $m_1$, $m_2$, and $m_4$, and then rotate $m_2$ and move $m_3$ on top of $m_2$ (perform Op1 on $m_2$ and Op2 on $m_3$), resulting in the configuration shown in Figure 9(h). Finally, we decluster $m_8$ shown in Figure 9(i) to $m_5$, $m_6$, and $m_7$, and move $m_4$ to the right of $m_3$ (perform Op2 for $m_4$), which results in the optimum placement shown in Figure 9(j).

### 5 Handling Soft Modules

In this section, we present our approach for handling soft modules. We first apply Lagrangian relaxation [24] to cluster soft modules at the clustering stage while keeping declustering the same as before. We then propose a network-flow based algorithm for projecting Lagrange multipliers to satisfy their optimality conditions.

#### 5.1 Formulation

Let $M = \{m_1, m_2, \ldots, m_n\}$ be a set of $n$ primitive soft modules. Each primitive soft module $m_i \in M$ is associated with a three tuple $(h_i, w_i, a_i)$, where $h_i$, $w_i$, and $a_i$ denote the width, height, and aspect ratio of $m_i$, respectively. The area $A_i$ of $m_i$ is given

```
Algorithm: MB*-tree($M, N$)
Input: $M$—the primitive modules;
       $N$—Nets;
Stage I: Clustering
1   Perform the clustering algorithm described in Section 4.1;
Stage II: Declustering
2   $i \leftarrow 1$; $k \leftarrow 0$; $n \leftarrow |M|$;
3   while $i \leq \lceil \log n \rceil$ do
4     $j \leftarrow 1$;
5       while $j \leq i^2$ and $k \leq n - 1$ do
6         Perform the declustering algorithm described in Section 4.2;
7         $j \leftarrow j + 1$; $k \leftarrow k + 1$;
8       end
9   end
10  $i \leftarrow i + 1$;
11  return
```

Figure 10: The MB*-tree algorithm.
by $h_iw_i$, and the aspect ratio $a_i$ of $m_i$ is given by $h_i/w_i \in [r_{i,\text{min}}, r_{i,\text{max}}]$. Let $L_i = \sqrt{A_i/r_{i,\text{min}}}$ and $U_i = \sqrt{A_i/r_{i,\text{max}}}$ denote the minimum and the maximum width of $m_i$, respectively. We have $h_i = A_i/w_i$ and $L_i \leq w_i \leq U_i$.

A cluster module $m_c$ is composed of a set of primitive soft modules $M_p$. $m_c$ can be reshaped via reshaping the modules in $M_p$ without violating the relations of the modules in $M_p$. We create two dummy modules $m_s$ and $m_t$ and set $x_s = 0$, $y_s = 0$, $w_s = 0$, and $h_s = 0$. Then we construct a horizontal and a vertical constraint subgraphs of $m_c$, denoted by $G_{hc}$ and $G_{vc}$, respectively. $G_{hc}$ and $G_{vc}$ are constructed as follows:

- For $m_s$ and $m_t$, create two vertices $v_s$ and $v_t$ in both $G_{hc}$ and $G_{vc}$.
- For each $m_p \in M_p$, create a vertex $v_{p}$ in $G_{hc}$ and $G_{vc}$.
- For each $m_p, m_q \in M_p$, if $m_p$ is left to (below) $m_q$, create an edge $e(p, q)$ from $v_{p}$ to $v_{q}$ in $G_{hc}$ ($G_{vc}$).
- For each $m_p$ which is placed at the left boundary (bottom boundary), create an edge $e(v_s, v_{p})$ from $v_s$ to $v_{p}$ in $G_{hc}$ ($G_{vc}$).
- For each $m_p$ which is placed at the right boundary (top boundary), create an edge $e(v_{p}, v_t)$ from $v_{p}$ to $v_t$ in $G_{hc}$ ($G_{vc}$).

If $x_{p} + w_{p} \leq x_{q}, \forall e(p, q) \in G_{hc}$ and $y_{p} + w_{p} \leq y_{q}, \forall e(p, q) \in G_{vc}$ are satisfied, the relations of the modules in $M_p$ will not be violated. Figure 11 illustrates how to construct $G_{hc}$ and $G_{vc}$ and what corresponding constraints must be satisfied. Figure 11(a) gives a cluster modules $m_c$ with the cluster scheme $\{m_1, m_2, m_3\}$. Figure 11(b) shows the corresponding constraint subgraphs $G_{hc}$ and $G_{vc}$ of $m_c$. Figure 11(c) shows the constraints to ensure that no relation of modules is violated. Thus, it implies that $w_{c} \geq x_{t}$, and $h_{c} \geq y_{t}$.

![Diagram](image_url)

At level $i$, Let $M^i = \{m_1^i, m_2^i, ..., m_{n_i}^i\}$ denote the set of cluster modules. For each $m_j^i \in M^i$, $(x_j^i, y_j^i)$ denote the coordinate of its bottom-left corner, and $h_j^i$ and $w_j^i$ denote the height and width of $m_j^i$, respectively. For convenience, we additionally create two variables, $x_{n_i+1}$ and $y_{n_i+1}$, which denote the estimated height and width of the chip at level $i$, respectively. Thus, the estimated area of the chip at level $i$ equals $x_{n_i+1}y_{n_i+1}$. To estimate wirelength, we adopt the quadratic of the length of the complete graph of pins in a net, and take the center of a module as the location of a pin, if the pins are not assigned during floorplanning. Let $E^i$
denote the set of nets at level $i$. For a net $e^i_j \in E^i$, $e^i_j$ can be represented as a set of the modules $\{m^i_k | e^i_j \}$. Thus, the estimated wirelength $\overline{w}^i_j$ of a net $e^i_j \in E^i$ is defined by

$$
\overline{w}^i_j = \sum_{m^i_k, m^i_l \in e^i_j} (((x^i_j + w^i_j / 2) - (x^i_k + w^i_k / 2))^2 + ((y^i_j + h^i_j / 2) - (y^i_k + h^i_k / 2))^2).
$$

We use the cost function $\phi'$ to guide the clustering of soft modules:

$$
\phi'(x, y) = \alpha x^i_{n+1} y^i_{n+1} + \beta \sum_{e^i_j \in E^i} \overline{w}^i_j,
$$

where $\alpha$ and $\beta$ are nonnegative user-defined parameters, and $\overline{w}^i_j$ denotes the estimated wirelength of a net $e^i_j$. In the formulation of clustering for soft modules, we have the constraints that all modules are not overlapped and must be laid in the chip (i.e. $x^i_j + w^i_j \leq x^i_{n+1}$ and $y^i_j + h^i_j \leq y^i_{n+1}$). Therefore, we can formulate the problem of clustering for soft modules, called $CS$, as follows:

Minimize

$$
\alpha x^i_{n+1} y^i_{n+1} + \beta \sum_{e^i_j \in E^i} \overline{w}^i_j
$$

subject to

$$
x^i_j + w^i_j \leq x^i_{n+1}, \forall 1 \leq j \leq n_i,
$$

$$
y^i_j + h^i_j \leq y^i_{n+1}, \forall 1 \leq j \leq n_i,
$$

$$
x^i_j \leq w^i_j, \forall 1 \leq j \leq n_i,
$$

$$
y^i_j + \frac{A_p}{w_p} \leq y^i_{n+1}, \forall (p, q) \in G_{hj} \forall 1 \leq j \leq n_i,
$$

$$
L_i \leq w_i \leq U_i, \forall 1 \leq i \leq n,
$$

where $\alpha$ and $\beta$ are nonnegative user-defined parameters.

### 5.2 Lagrangian Relaxation

Then, the Lagrangian relaxation subproblem associated with the multiplier $\overline{\lambda} = (\overline{\lambda}, \overline{\eta}, \overline{\lambda}, \overline{\mu}, \overline{\pi}, \overline{\sigma})$, denoted by $LRS/\overline{\lambda}$, can be defined as follows:

Minimize

$$
\alpha x^i_{n+1} y^i_{n+1} + \beta \sum_{e^i_j \in E^i} \overline{w}^i_j
$$

$$
+ \sum_{j=1}^{n_i} \alpha_j (x^i_j + w^i_j - x^i_{n+1}) + \sum_{j=1}^{n_i} \eta_j(y^i_j + h^i_j - y^i_{n+1})
$$

$$
+ \sum_{j=1}^{n_i} \sum_{(p, q) \in G_{hj}} \lambda_{j p q} (x^i_p + w^i_p - x^i_q)
$$

$$
+ \sum_{j=1}^{n_i} \sum_{(p, q) \in G_{vj}} \mu_{j p q} \left( y^i_p + \frac{A_p}{w_p} - y^i_q \right)
$$

$$
+ \sum_{j=1}^{n_i} r_j (x^i_j + w^i_j) + s_j (y^i_j + h^i_j)
$$

subject to

$$
L_i \leq w_i \leq U_i, \forall 1 \leq i \leq n.
$$

Let $Q(\overline{\lambda})$ denote the optimal value of $LRS/\overline{\lambda}$. The Lagrangian dual problem $\mathit{LD\overline{\lambda}}$ of $CS$ can be defined as follows:

Maximize

$$
Q(\overline{\lambda})
$$

subject to

$$
\overline{\lambda} \geq 0.
$$
Since CS can be transformed into a convex problem, we can apply Theorem 6.2.4 of [3]. This implies that if $\vec{P}$ is an optimal solution to $LDP$, the optimal solution of $LRS/(\vec{P})$ will also optimize $CS$.

Consider the Lagrangian $\zeta$ of $CS$ defined as follows:

$$
\zeta = \alpha x_{n_{i+1}}^i + \beta \sum_{e_j \in E^i} \varphi_j + \sum_{j=1}^{n_i} \kappa_j (x_j + w_j - x_{n_{i+1}}^i)
$$

$$
+ \sum_{j=1}^{n_i} \eta_j (y_j^i + h_j^i - y_{n_{i+1}}^i) + \sum_{j=1}^{n_i} \sum_{e_{(p,q)} \in G_{s_j}} \lambda_{jpq} (x_p + w_p - x_q)
$$

$$
+ \sum_{j=1}^{n_i} \sum_{e_{(p,q)} \in G_{s_j}} \mu_{jpq} \left( y_p + \frac{A_p}{w_p} - y_q \right) + \sum_{j=1}^{n_i} \mu_j \left( x_j - w_j \right)
$$

$$
+ s_j (y_j - h_j^i) + \sum_{i=1}^{n_i} u_i (L_i - w_i) + \sum_{i=1}^{n_i} v_i (w_i - U_i).
$$

The Kuhn-Tucker conditions imply that the optimal solution of $CS$ must be at $\partial \zeta / \partial x_p = 0$, $\partial \zeta / \partial y_p = 0$, $\partial \zeta / \partial x_{n_{i+1}}^i = 0$, and $\partial \zeta / \partial y_{n_{i+1}}^i = 0$. Thus, we only need to consider the multipliers $\vec{P}$ which satisfy these conditions. Therefore, for $1 \leq p \leq n$,

$$
\partial \zeta / \partial x_p = \sum_{j=1}^{n_i} \left( \sum_{e_{(p,q)} \in G_{s_j}} \lambda_{jpq} - \sum_{e_{(q,p)} \in G_{s_j}} \lambda_{jpq} \right) = 0, \quad (6)
$$

and

$$
\partial \zeta / \partial y_p = \sum_{j=1}^{n_i} \left( \sum_{e_{(p,q)} \in G_{s_j}} \eta_{jpq} - \sum_{e_{(q,p)} \in G_{s_j}} \eta_{jpq} \right) = 0, \quad (7)
$$

and

$$
\partial \zeta / \partial x_{n_{i+1}}^i = \alpha y_{n_{i+1}}^i - \sum_{j=1}^{n_i} \kappa_j = 0,
$$

$$
\partial \zeta / \partial y_{n_{i+1}}^i = \alpha x_{n_{i+1}}^i - \sum_{j=1}^{n_i} \eta_j = 0.
$$

Thus, we have $y_{n_{i+1}}^i = \frac{1}{\alpha} \sum_{j=1}^{n_i} \kappa_j$, and $x_{n_{i+1}}^i = \frac{1}{\alpha} \sum_{j=1}^{n_i} \eta_j$.

5.3 Solving $LRS/(\vec{P})$ and $LDP$

Let $\Omega$ denote the set of multipliers $\vec{P}$ satisfying Equations (6) and (7). We now consider solving the Lagrangian relaxation subproblem $LRS/(\vec{P})$ for a given $\vec{P} \in \Omega$, i.e. computing the dimension and coordinate of each module. First, we partially differentiate $\zeta$ with respect to $w_i$ to get an optimal value of $w_i$ such that $\zeta$ is minimized.

$$
\partial \zeta / \partial w_i = (v_p - u_p) + \sum_{j=1}^{n_i} \left( \sum_{e_{(p,q)} \in G_{s_j}} \lambda_{jpq} \right)
$$

$$
- \sum_{j=1}^{n_i} \left( \sum_{e_{(p,q)} \in G_{s_j}} \mu_{jpq} \frac{A_p}{w_p} \right) = 0.
$$

Thus, we have

$$
w_p = \sqrt{\frac{\sum_{j=1}^{n_i} \sum_{e_{(p,q)} \in G_{s_j}} \mu_{jpq} A_p}{(v_p - u_p) + \sum_{j=1}^{n_i} \sum_{e_{(p,q)} \in G_{s_j}} \lambda_{jpq}}},
$$

where $out_G(v) = \{u | e(v, u) \in E(G)\}$. Recall that $L_p \leq w_p \leq U_p, 1 \leq p \leq n$. Thus, the optimal $w_p^*$ can be computed by $w_p^* = \min \{U_p, \max \{L_p, w_p\}\}$. 

13
Since the dimension of each primitive module \((w_p, h_p)\) has been determined, the dimension of each cluster module \((w_j^i, h_j^i)\) can be computed by applying a longest path algorithm in \(G_{h,j}\) and \(G_{w,j}\). Then, we consider partial differentiation of \(\zeta\) with respect to \(x_j^i\) and \(y_j^i\), giving the optimality conditions of \(CS\). Therefore, for \(1 \leq j \leq n_i\),

\[
\frac{\partial \zeta}{\partial x_j^i} = \beta \left( \sum_{e_k^i \supset \{m_j^i\}} \left( 2 \left| e_k^i \right| - 1 \right) x_j^i - \sum_{e_k^i \supset \{m_j^i\}} \sum_{m_l^i \subset e_k^i \setminus \{m_j^i\}} x_l^i \right) + \sum_{e_k^i \supset \{m_j^i\}} \sum_{m_l^i \subset e_k^i \setminus \{m_j^i\}} \left( w_j^i - w_l^i \right) = 0 \tag{8}
\]

\[
\frac{\partial \zeta}{\partial y_j^i} = \beta \left( \sum_{e_k^i \supset \{m_j^i\}} \left( 2 \left| e_k^i \right| - 1 \right) y_j^i - \sum_{e_k^i \supset \{m_j^i\}} \sum_{m_l^i \subset e_k^i \setminus \{m_j^i\}} y_l^i \right) + \sum_{e_k^i \supset \{m_j^i\}} \sum_{m_l^i \subset e_k^i \setminus \{m_j^i\}} \left( h_j^i - h_l^i \right) = 0, \tag{9}
\]

where \(|e_k^i|\) denotes the number of the pins of \(e_k^i\).

In Equation (8), there are \(n_i\) equations with \(n_i\) variables. Thus, we can apply the Gaussian elimination to solve these \(n_i\) equations with \(n_i\) variables to get the optimal value of \(x_j^i\). In these \(n_i\) equations, all coefficients of variables depend only on the net information (i.e., \(e_k^i\)). Since the net information is the same through the entire process, each variable can be solved by the same process. Hence, we can record the process of solving each variable during the first iteration (which takes cubic time), and then each subsequent computation will take only quadratic time by applying the same process. Similarly, we can compute the optimal value of \(y_j^i\).

Next, we use a subgradient optimization method to search for the optimal \(\vec{P}\). Let \(\vec{P}\) be a multiplier at step \(k\). We move \(\vec{P}\) to a new multiplier \(\vec{P}^t\) based on the subgradient direction:

\[
\begin{align*}
v_j^i &= [\kappa_j + \rho_k (x_j^i + w_j^i - x_{n_i+1}^i)]^+ \\
n_j^i &= [\eta_j + \rho_k (y_j^i + h_j^i - y_{n_i+1}^i)]^+ \\
\lambda_{jpq}^i &= [\lambda_{jpq} + \rho_k (x_p + w_p - x_q)]^+ \\
\mu_{jpq}^i &= [\mu_{jpq} + \rho_k (y_p + \frac{A_p}{w_p} - y_q)]^+,
\end{align*}
\]

where \([x]^+ = \max\{x, 0\}\) and \(\rho_k\) is a step size such that \(\lim_{k \to \infty} \rho_k = 0\) and \(\sum_{k=1}^{\infty} \rho_k = \infty\).

After updating \(\vec{P}\), we need to project \(\vec{P}^t\) to \(\vec{P}^* \in \Omega\), and then solve the Lagrangian relaxation subproblem \(LRS/\{\vec{P}^*\}\) by the above algorithm until the solution converges.

### 5.4 Projecting Lagrange Multipliers

We present a network flow based algorithm to check whether \(\vec{P}\) belongs to \(\Omega\) and to project \(\vec{P}\) to \(\vec{P}^* \in \Omega\), if \(\vec{P} \notin \Omega\). Further, an incrementual update technique is employed to make the maximum flow computation more efficient. For each cluster module \(m_c\), we first create two networks \(N_{hc}\) (for \(G_{hc}\)) and \(N_{wc}\) (for \(G_{wc}\)) as follows:

- For each \(v_i \in V(G_{hc})\) (\(V(G_{wc})\)), create a vertex \(v_i^t\) in \(N_{hc}\) (\(N_{wc}\)), and make \(v_i^t\) and \(v_i^t\) as the source and sink, respectively.
- For each \(e(p, q) \in E(G_{hc})\) (\(E(G_{wc})\)), create a corresponding edge \(e(p^t, q^t)\) with capacity \(\lambda_{cpq} \lambda_{cpq}\) in \(N_{hc}\) (\(N_{wc}\)).

We apply the maximum flow computation on the networks to check whether \(\vec{P}\) belongs to \(\Omega\). The maximum flow computation finds an augmenting path from \(v_i^t\) to \(v_i^t\) and then pushes flow on it until no argument path can be found. Let \(\text{cap}(v, u)\) and \(\text{flow}(v, u)\) denote the capacity and flow on the edge \(e(v, u)\). An edge \(e(v, u)\) is saturated if its capacity equals the flow (i.e., \(\text{cap}(v, u) = \text{flow}(v, u)\)).
Theorem 2 If all edges in the networks are saturated, \( \mathcal{P} \in \Omega \).

Proof: After the maximum flow computation, for each \( v'_p \) in a network except the source and sink, the sum of the flows of \( v'_p \)'s incoming edges equals the sum of its outgoing ones (i.e., \( \sum_{e(p',q') \in N_{ve}} \text{flow}(p',q') = \sum_{e(q',p') \in N_{ve}} \text{flow}(q',p') \) for each \( N_{ve} \) and \( \sum_{e(p',q') \in N_{ve}} \text{flow}(p',q') = \sum_{e(q',p') \in N_{ve}} \text{flow}(q',p') \) for each \( N_{ve} \)). Besides, \( \text{cap}(p',q') = \text{flow}(p',q') \) for all edges \( e(p',q') \) (all edges are saturated), and \( \text{cap}(p',q') \) of each edge \( e(p',q') \) in \( N_{ve} \) equals \( \lambda_{cpq} (\mu_{cpq}) \). Hence, \( \sum_{e(p,q) \in G_{ve}} \lambda_{cpq} (\mu_{cpq}) \) for each cluster module \( m_c, \mathcal{P} \) belongs to \( \Omega \).

If \( \mathcal{P} \) does not belong to \( \Omega \), we project \( \mathcal{P} \) to \( \mathcal{P}^* \) by restoring the flow \( \text{flow}(p',q') \) of each edge \( e(p',q') \) in \( N_{ve} \) to \( \lambda_{cpq} (\mu_{cpq}) \) for each \( m_c \).

Theorem 3 \( \mathcal{P}^* \in \Omega \).

The projection process greatly affects the efficiency of the entire optimization, since there may be \( O(n^2) \) edges in the worst case. Thus, we employ an incremental flow update technique to speed up the max-flow computation after updating \( \mathcal{P} \) and its corresponding capacity. Figure 12 gives an algorithm for the incremental network update. Lines 1–2 check whether each edge \( e(p',q') \) violates the capacity constraint (i.e., \( 0 \leq \text{flow}(p',q') \leq \text{cap}(p',q') \)). Lines 3–9 fix the overflow on \( e(p',q') \), if an edge \( e(p',q') \) violates its capacity constraint. Finally, Line 10 computes a maximum flow again.

Note that, for efficiency consideration, we may perform Lagrangian relaxation only at the higher levels of the multilevel framework (when the number of modules become small enough for Lagrangian relaxation). To do so, however, we still need to pass the information of the aspect ratio for each soft module level by level.

Algorithm: IncrementalUpdate\( (N, s, t) \)
Input: \( N \)—the flow network; \( s \)—the source of \( N \); \( t \)—the sink of \( N \);
1. for each edge \( e(p',q') \in E(N) \) do
2. if \( \text{flow}(p',q') > \text{cap}(p',q') \) then
3. \( f_{\text{over}} \leftarrow \text{flow}(p',q') - \text{cap}(p',q') \);
4. while \( f_{\text{over}} > 0 \) do
5. find a path \( p \) from \( s \) to \( t \), passing through \( e(p',q') \), and \( \min \{ \text{flow}(u,v) | e(u,v) \in p \} > 0 \);
6. \( f_{\text{reduced}} \leftarrow \min \{ \min \{ \text{flow}(u,v) | e(u,v) \in p \}, f_{\text{over}} \} \);
7. for each edge \( e(u,v) \in p \)
8. \( \text{flow}(u,v) \leftarrow \text{flow}(u,v) - f_{\text{reduced}} \);
9. \( f_{\text{over}} \leftarrow f_{\text{over}} - f_{\text{reduced}} \);
10. compute maximum flow on \( N \);

Figure 12: The incremental update algorithm.

6 Experimental Results

We implemented the MB*-tree algorithm in the C++ language on a 450 MHz SUN Ultra 60 workstation with 2 GB memory. The package is available at http://cc.ee.ntu.edu.tw/~ywchang/research.html.

Columns 1, 2, and 3 of Table 1 lists the names of the benchmark circuits, the number of modules, and the number of nets, respectively. \text{ami49} is the largest MCNC benchmark circuit used in the previous works [5, 9] for comparative study. To test the capability of existing methods, we created ten synthetic circuits, named \text{ex ami49 x}, by duplicating the modules and nets of \text{ami49} by \( x \) times. The largest circuit \text{ex ami49 200} contains 9,800 modules and 81,600 nets (specified by pin-to-pin interconnections).

Table 2 shows the results for \text{ex ami49 x} by optimizing area alone (\( \gamma = 1.0 \) and \( \delta = 0.0 \)). Columns 2, 3, 4, 5, and 6 give the total area of modules in the circuit, the resulting area, the dead space, the runtime, and the memory requirement for our
Table 1: The benchmark circuits used in our experiment.

<table>
<thead>
<tr>
<th>Circuit</th>
<th>#modules</th>
<th>#net</th>
</tr>
</thead>
<tbody>
<tr>
<td>ami49</td>
<td>49</td>
<td>408</td>
</tr>
<tr>
<td>ex_ami49_2</td>
<td>98</td>
<td>816</td>
</tr>
<tr>
<td>ex_ami49_4</td>
<td>196</td>
<td>1632</td>
</tr>
<tr>
<td>ex_ami49_10</td>
<td>490</td>
<td>4080</td>
</tr>
<tr>
<td>ex_ami49_20</td>
<td>980</td>
<td>9160</td>
</tr>
<tr>
<td>ex_ami49_40</td>
<td>1960</td>
<td>16320</td>
</tr>
<tr>
<td>ex_ami49_60</td>
<td>2940</td>
<td>24480</td>
</tr>
<tr>
<td>ex_ami49_80</td>
<td>3920</td>
<td>32640</td>
</tr>
<tr>
<td>ex_ami49_100</td>
<td>4900</td>
<td>40800</td>
</tr>
<tr>
<td>ex_ami49_150</td>
<td>7350</td>
<td>61200</td>
</tr>
<tr>
<td>ex_ami49_200</td>
<td>9800</td>
<td>81600</td>
</tr>
<tr>
<td>industry</td>
<td>189</td>
<td>9777</td>
</tr>
</tbody>
</table>

Figure 13: (a) Comparison for the dead space vs. circuit size (# of modules). (b) Comparison for the CPU time vs. circuit size (# of modules).

MB*-tree, respectively. The remaining columns list the results for the famous previous works, sequence pair [19], O-tree [9], and B*-tree [5]. Note that the B*-tree package we used here is the September 2000 version, B*-tree-v1.0, available also at http://cc.ee.ntu.edu.tw/~ywchang/research.html. It runs 50X–100X faster and achieves better area utilization than the B*-tree package reported in [5]. As shown in the table, our MB*-tree algorithm obtained a dead space of only 2.78% for ami49 in only 0.4 min runtime and 1.3 MB memory while B*-tree-v1.0 reported a dead space of 3.53% using 0.25 min runtime and 3.2 MB memory. Further, the experimental results for larger circuits show that the MB*-tree scales very well as the circuit size increases while the previous works, sequence pair, O-tree, and B*-tree, do not. For circuit sizes ranging from 49 to 9,800 modules and from 408 to 81,600 nets, the MB*-tree consistently obtains high-quality floorplans with dead spaces of less than 3.72% in empirically linear runtime, while sequence pair, O-tree, and B*-tree can handle only up to 196, 98, and 1,960 modules in the same amount of time and result in dead spaces of as large as 13.00% (@ 196 modules), 12.29% (@ 98 modules), and 27.33% (@ 1960 modules), respectively. In Figures 13, the resulting dead space, and runtime are plotted as functions of the circuit size (in the number of modules), respectively. As shown in Table 2 and Figures 13(a), the resulting dead spaces for the MB*-tree is almost independent of the circuit sizes, which proves the high scalability of the MB*-tree. In contrast, the dead spaces for the non-hierarchical previous
Table 2: Comparisons for area, dead space, runtime, and memory among MB*-tree, Sequence pair, O-tree, and B*-tree. **NR:** No result obtained within 300-minute CPU time on SUN Sparc Ultra 60. Note that MB*-tree, sequence pair, and B*-tree finished their memory allocation in the very early stage of execution. Therefore, their memory consumption for the listed circuit sizes can be measured. O-tree performs memory allocation and de-allocation during execution; therefore, only the memory requirements for the small cases that finished execution are available.
works all grow dramatically as the circuit size increases. Figure 13(b) shows the empirical runtime for the four algorithms. This figure reveals that the empirical runtime of the MB*-tree In particular, the empirical runtime of the MB*-tree approaches linear in the circuit size while the other previous works cannot handle large-scale designs. Figure 14 shows the layout for the largest circuit ex_ami49_200 obtained by MB*-tree in 256 min CPU time. It has a dead space of only 3.44%. Note that this circuit is not feasible to the previous works [5, 9, 19].

Table 3 shows the comparisons for area optimization alone (γ = 1.0, δ = 0.0), wirelength optimization alone (γ = 0.0, δ = 1.0), and simultaneous area and wirelength optimization (γ = 0.5, δ = 0.5) among sequence pair (SP), B*-tree, and MB*-tree based on the circuit industry (whose total area = 658.04 mm²). The circuit industry is a 0.18 µm, 1 GHz industrial design with 189 modules, 20 million gates, and 9,777 center-to-center interconnections. It is a large chip design and consists of three “tough” modules with aspect ratios greater than 19 (and as large as 36). (Note that we do not have the results for O-tree for this experiment because the data industry cannot be fed into the O-tree package.) In each entry of the table, we list the best/average values obtained in ten runs of simulated annealing, using a random seed for each run. For the column “Time,” we report the runtime for obtaining the best value and the average runtime of the ten runs. As shown in the table, our MB*-tree algorithm obtained significantly better silicon area and wirelength than sequence pair and B*-tree in all tests. For area optimization, MB*-tree can obtain a dead space of only 2.11% while sequence pair (B*-tree) results in a dead space of at least 28.1% (12.9%). For wirelength optimization, MB*-tree can obtain a total wirelength of only 56631 mm while sequence pair (B*-tree) requires a total wirelength of at least 81344 mm (113216 mm). For simultaneous area and wirelength optimization, MB*-tree also obtains the best area and wirelength. The results show the effectiveness of our MB*-tree algorithm. For the runtimes, MB*-tree is larger than B*-tree and SP for wirelength optimization. (For area optimization, MB*-tree runs faster than SP.) This is reasonable because it took much longer to obtain significantly better results and the multilevel process incurred some overhead. Nevertheless, as shown in Table 2, both SP and B*-tree do not scale well to the instances with a large number of modules (and thus their runtimes increase dramatically when the number of modules grows into hundreds). The resulting layout of industry for simultaneous area and wirelength optimization using MB*-tree is shown in Figure 15.

<table>
<thead>
<tr>
<th>Package</th>
<th>Area optimization (γ = 1.0, δ = 0.0)</th>
<th>Wirelength optimization (γ = 0.0, δ = 1.0)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Area (mm²)</td>
<td>Dead space (%)</td>
</tr>
<tr>
<td>SP</td>
<td>914.5/988.0</td>
<td>28.1/33.2</td>
</tr>
<tr>
<td>B*-tree</td>
<td>755.7/876.6</td>
<td>12.9/24.7</td>
</tr>
<tr>
<td>MB*-tree</td>
<td>671.9/740.8</td>
<td>11.5/6.6</td>
</tr>
<tr>
<td>SP : MB*-tree</td>
<td>1.36/1.34</td>
<td>13.38/10.71</td>
</tr>
<tr>
<td>B*-tree : MB*-tree</td>
<td>1.13/1.18</td>
<td>6.14/7.97</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Package</th>
<th>Simultaneous area and wirelength optimization (γ = 0.5, δ = 0.5)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Area (mm²)</td>
</tr>
<tr>
<td>SP</td>
<td>1104.2/1182.5</td>
</tr>
<tr>
<td>B*-tree</td>
<td>834.7/982.8</td>
</tr>
<tr>
<td>MB*-tree</td>
<td>716.3/740.8</td>
</tr>
<tr>
<td>SP : MB*-tree</td>
<td>1.54/1.60</td>
</tr>
<tr>
<td>B*-tree : MB*-tree</td>
<td>1.17/1.33</td>
</tr>
</tbody>
</table>

Table 3: Comparisons for area optimization alone, wirelength optimization alone, and simultaneous area and wirelength optimization among sequence pair (SP), B*-tree, and MB*-tree based on the circuit industry. In each entry, both the best/average values obtained in ten runs of simulated annealing are reported. The last two rows give the ratios of the results (SP to MB*-tree and B*-tree to MB*-tree).
7 Concluding Remarks

We have presented the MB*-tree based multilevel framework to handle the floorplanning and packing for large-scale modules. Experimental results have shown that the MB*-tree scales very well as the circuit size increases. The capability of the MB*-tree shows its promise in handling large-scale designs with complex constraints. We propose to explore the floorplanning/placement problem with large-scale rectilinear and mixed sized modules/cells as well as buffer-block planning for interconnect-driven floorplanning in the future.

References


Figure 15: The layout of industry by simultaneously optimizing area and wirelength ($\gamma = 0.5, \delta = 0.5$). CPU time = 5234 sec, Area = 716263680 $\mu m^2$, Total wirelength = 67786296 $\mu m$, Dead space = 8.14%.


