CHAPTER

10 Floorplanning

Tung-Chieh Chen
National Taiwan University, Taipei, Taiwan

Yao-Wen Chang
National Taiwan University, Taipei, Taiwan

ABOUT THIS CHAPTER

Floorplanning is an essential design step for hierarchical, building-module design methodology. Floorplanning provides early feedback that evaluates architectural decisions, estimates chip areas, and estimates delay and congestion caused by wiring. As technology advances, design complexity is increasing and the circuit size is getting larger. To cope with the increasing design complexity, hierarchical design and ***intellectual property*** (IP) modules are widely used. This trend makes floorplanning much more critical to the quality of a ***very large-scale integration*** (VLSI) design than ever.

This chapter starts with the formulation of the floorplanning problem. After the problem formulation, the two most popular approaches to floorplanning, simulated annealing and analytical formulations, are discussed. On the basis of simulated annealing, three popular floorplan representations, normalized Polished expression, B*-tree, and sequence pair, are further covered and compared. Some modern floorplanning issues such as soft modules, fixed-outline constraints, and large-scale designs are also addressed.

10.1 INTRODUCTION

In Chapter 1, we introduced the electronic design automation flow. Floorplanning is the first major step in physical design; it is particularly important because the resulting floorplan affects all the subsequent steps in physical design, such as placement and routing that are discussed in Chapters 11 and 12, respectively.

10.1.1 Floorplanning basics

Two popular approaches to floorplanning, **simulated annealing** and **analytical formulation**, are typically used to solve the floorplanning problem [Sait 1999;

Sherwani 1999]. Basically, simulated annealing-based floorplanning relies on the representation of the geometric relationship among modules, whereas an analytical approach usually captures the absolute relationship directly. The topological representation profoundly affects the operations of modules and the complexity of a simulated annealing-based floorplan design process. In this chapter, three popular floorplan representations, **normalized Polish expression** [Wong 1986], **B*-tree** [Chang 2000], and **Sequence Pair** [Murata 1995], are introduced. In general, these representations are efficient, flexible, and effective in modeling **geometric relationships** (*e.g.,* left, right, above, and below relationships) among modules for floorplan designs. The simulated annealing-based floorplanning is concluded with the comparisons of popular floorplan representations in the recent literature.

The analytical approach applies **mathematical programming** that is composed of an objective function and a set of constraints. The objective function models the cost metric (*e.g.,* area and wirelength) for floorplan optimization, whereas the constraints capture the geometric and dimensional restrictions among modules (*e.g.,* the nonoverlapping and aspect ratio constraints). Specifically, this chapter introduces ***mixed integer linear programming*** (ILP) for the floorplanning problem [Sutanthavibul 1990]. For the mixed ILP formulation, an approximated area is modeled by a linear cost function, whereas the nonoverlapping and aspect ratio constraints are modeled by a set of linear equations. To handle the expensive time complexity of mixed ILPs, a **successive augmentation** method that solves a partial problem at each step to reduce the floorplanning complexity is also introduced.

In addition to chip area minimization, modern VLSI floorplanning also needs to handle some important issues such as **soft modules** and **fixed-outline constraints**. Unlike a **hard module** that has a fixed dimension (width and height), the shape of a soft module is to be decided during floorplanning, although its area is fixed. Therefore, a floorplanner needs to find a desired aspect ratio for each soft module to optimize the floorplan cost. As pointed out by [Kahng 2000], modern VLSI design is based on a fixed-die (fixed-outline) floorplan, rather than a variable-die one. An area-optimized floorplan without considering the fixed-outline constraint may be useless, because it might not fit into the given outline. Therefore, modern floorplanning should address the fixed-outline consideration.

As the transistor feature size scales down, design complexity is increasing drastically. To cope with the increasing design complexity, ***intellectual property*** (IP) modules are widely reused for large-scale designs. Consequently, a modern VLSI design often consists of large-scale functional modules, and designs with billions of transistors are already in production. Therefore, efficient and effective design methods and tools capable of placing and optimizing large-scale modules are essential for modern chip designs. In addition to the enhancement in floorplanning tools, the floorplanning frameworks are evolving to tackle the challenges in design complexity. This chapter also addresses the **multi-level frameworks** for large-scale building module designs.

10.1.2 **Problem statement**

The floorplanning problem can be stated as follows: Let $B = \{b_1, b_2, \ldots, b_m\}$ be a set of m rectangular modules whose respective width, height, and area are denoted by w_i, h_i, and a_i, $1 \leq i \leq m$. Each module is free to rotate. Let (x_i, y_i) denote the coordinate of the bottom-left corner of module b_i, $1 \leq i \leq m$, on a chip. A floorplan F is an assignment of (x_i, y_i) for each b_i, $1 \leq i \leq m$, such that no two modules overlap with each other. The goal of floorplanning is to optimize a predefined cost metric such as a combination of the area (*i.e.*, the minimum bounding rectangle of F) and wirelength (*i.e.*, the sum of all interconnection lengths) induced by a floorplan. For modern floorplan designs, other costs such as routability, power, and thermal might also need to be considered.

10.1.3 **Floorplanning model**

We can classify floorplans into two categories for discussions: (1) **slicing floorplans** and (2) **non-slicing floorplans**. A slicing floorplan can be obtained by repetitively cutting the floorplan horizontally or vertically, whereas a non-slicing floorplan cannot. The given dimension of each hard module must be kept. All modules are free of rotation; if a module is rotated, its width and height are exchanged.

Example 10.1 Two example floorplans are shown in Figure 10.1. Consider the non-slicing floorplan with five modules shown in Figure 10.1a first. The five modules can be rearranged to form a slicing floorplan given in Figure 10.1b, in which each module can be extracted by repetitively cutting the floorplan horizontally or vertically.

10.1.3.1 ***Slicing floorplans***

On the basis of the slicing property of a slicing floorplan, we can use a binary tree to represent a slicing floorplan [Otten 1982]. A **slicing tree** is a binary tree with modules at the leaves and cut types at the internal nodes. There are two cut types, H and V. The H cut divides the floorplan horizontally, and the left (right) child represents the bottom (top) sub-floorplan. Similarly, the V cut divides the floorplan vertically, and the left (right) child represents the left (right) sub-floorplan.

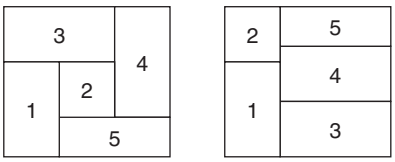

FIGURE 10.1

Examples of (a) a non-slicing floorplan. (b) a slicing floorplan.

Note that a slicing floorplan may correspond to more than one slicing tree, because the order of the cut-line selections may be different. This representation duplication might incur a larger solution space and complicate the optimization process. Therefore, it is desirable to prune such redundancies to facilitate floorplan design. As such, we refer to a slicing tree as a **skewed slicing tree** if it does not contain a node of the same cut type as its right child.

Example 10.2 Figure 10.2 shows the slicing floorplan from Figure 10.1b and its corresponding slicing tree. The tree root, V, represents the vertical cut-line that divides the floorplan into the left sub-floorplan (modules 1 and 2) and the right sub-floorplan (modules 3, 4, and 5). The left child of the root is a node H, which horizontally divides the left sub-floorplan into the bottom sub-floorplan (module 1) and the top sub-floorplan (module 2). Similarly, the right child of the root horizontally cuts the sub-floorplan into the bottom (module 3) and the top (modules 4 and 5) sub-floorplans, and the top sub-floorplan is further divided into the bottom (module 4) and the top (module 5) sub-floorplans.

There are two slicing trees corresponding to the floorplan in Figure 10.2a. Figure 10.3a and Figure 10.3b show the two slicing trees. The slicing tree in Figure 10.3a is a non-skewed slicing tree, because there is a node H as the right child of anther node H (see the dashed box). The slicing tree in Figure 10.3b gives a skewed one.

10.1.3.2 *Non-slicing floorplans*

The non-slicing floorplan is more general than the slicing floorplan. However, because of its non-slicing structure, we cannot use a slicing tree to model it. Instead, we can use a ***horizontal constraint graph*** (HCG) and a ***vertical constraint graph*** (VCG) to model a non-slicing floorplan. The horizontal constraint graph defines the horizontal relations of modules, and the vertical constraint graph defines the vertical ones. In a constraint graph, a node represents a module. If there is an edge from node A to node B in the HCG (VCG), then module A is at the left (bottom) of module B.

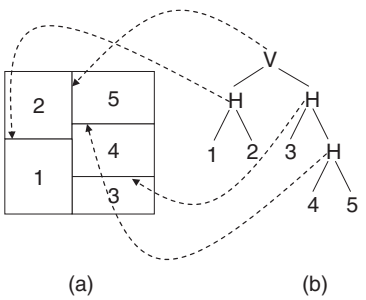

FIGURE 10.2

An example of (a) a slicing floorplan. (b) a slicing tree modeling.

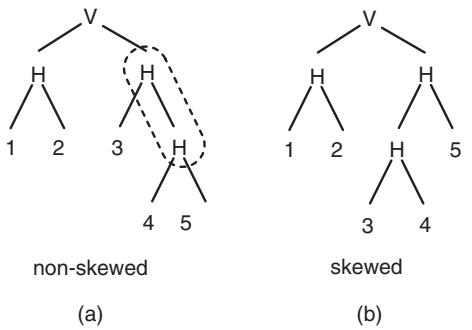

FIGURE 10.3

An example of (a) a non-skewed slicing tree. (b) a skewed slicing tree. Both slicing trees represent the same floorplan shown in Figure 10.2a.

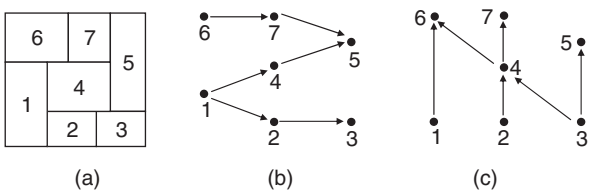

FIGURE 10.4

An example of (a) a non-slicing floorplan. (b) The horizontal constraint graph. (c) the vertical constraint graph.

Example 10.3 Consider the example non-slicing floorplan given in Figure 10.4a. Figure 10.4b and Figure 10.4c show the corresponding horizontal and vertical constraint graphs of the floorplan, respectively. Because module 1 is at the left of module 4, we add a directed edge from node 1 to node 4 in the horizontal constraint graph. Similarly, module 2 is below module 4, and thus we add a directed edge from node 2 to node 4 in the vertical constraint graph. On the basis of the left/right and above/below relationships, we can construct a horizontal and a vertical constraint graph corresponding to a floorplan.

10.1.4 Floorplanning cost

The goal of floorplanning is to optimize a predefined cost function, such as the area of a resulting floorplan given by the minimum bounding rectangle of the floorplan region. The floorplan area directly correlates to the chip silicon cost. The larger the area, the higher the silicon cost. The space in the floorplan bounding rectangle uncovered by any module is called **white space** or **dead space**.

Other floorplanning cost, such as wirelength, will also be considered. Shorter wirelength not only can reduce signal delay but also can facilitate wire interconnection at the routing stage. The floorplanning objective can also be a combined cost, such as area plus wirelength.

Example 10.4 Consider the two different floorplans for the same seven modules given in Figure 10.5. The left figure (see Figure 10.5a) is an optimal floorplan in terms of area, because there is no wasted area among modules. The right figure (see Figure 10.5b) illustrates a non-optimal floorplan. It is clear that the area of the right floorplan is larger than that of the left one, because there are white spaces in the floorplan shown in Figure 10.5b.

There are two popular approaches to find a desired floorplan: simulated annealing and the analytical approach. The two approaches are discussed in the following two sections.

10.2 SIMULATED ANNEALING APPROACH

Simulated annealing (SA) is probably the most popular method for floorplan optimization [Kirpatrick 1983]. It has the significant advantage of easily incorporating an optimizing goal into the objective function. To apply simulated annealing for floorplan design, it needs to first encode a floorplan as a solution, called a **floorplan representation**, which models the geometric relation of modules in a **floorplan**. A floorplan representation not only induces a solution space that contains all feasible solutions defined by the representation but also induces a unique solution structure that guides the search of simulated annealing to find a desired floorplan in the solution space. In this section, we detail three popular floorplan representations, **Normalized Polish Expression** [Wong 1986], **B*-tree** [Chang 2000], and **Sequence Pair** [Murata 1995], and summarize the properties of some popular recent representations in the literature.

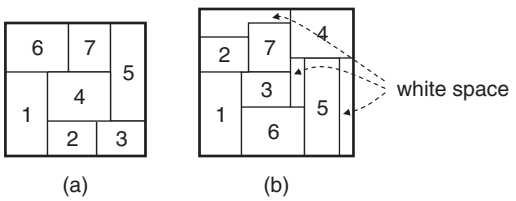

FIGURE 10.5

An example of (a) an optimal floorplan in terms of area. (b) a non-optimal floorplan.

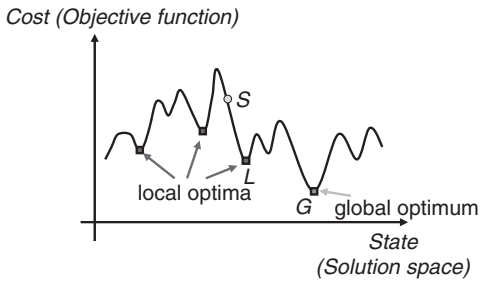

FIGURE 10.6

Illustration of simulated annealing.

10.2.1 **Simulated annealing basics**

Figure 10.6 illustrates the simulated annealing process. In Figure 10.6, the **cost** of a floorplan (represented by the vertical axis) is plotted as a function of the **state** of a floorplan configuration (represented by the horizontal axis). Given an initial solution S, it tries to search for a globally optimal floorplan solution with the lowest cost. For a greedy approach, it might iteratively search for a neighboring solution with a lower cost than that of the current solution until such a neighboring solution cannot be found. For this greedy mechanism, it is very likely that we get stuck in a locally optimal solution, such as L in Figure 10.6. Once it gets stuck at L, it is impossible to escape from this locally optimal solution, because all neighboring solutions have higher costs than L. Unlike the greedy approach, simulated annealing adopts a **hill-climbing** technique to escape from the locally optimal solution L. Given a solution, simulated annealing provides a non-zero probability to move from the current solution to a neighboring one even with a higher cost. With this **uphill move** capability, it is possible to reach the globally optimal solution G no matter where the initial solution starts. The probability of accepting a neighboring solution depends on two factors: (1) the magnitude of the uphill move and (2) the search time. To implement this idea, the probability of accepting a new solution S' is defined by:

$$\text{Prob}(S \rightarrow S') = \begin{cases} 1 & \text{if } \Delta C \leq 0 \quad \text{(down-hill move)} \\ e^{\Delta C/T} & \Delta C > 0 \quad \text{(up-hill move)} \end{cases}$$

where $\Delta C = \text{cost}(S') - \text{cost}(S)$, and T is the current temperature. Every down-hill move is accepted, and the probability of accepting an uphill move depends on the magnitude of the move (cost difference) and the search time (annealing temperature). Initially, we are assigned a high temperature. As the annealing process goes by, the temperature is typically decreased by a fixed ratio, say 0.9. For example, a simple annealing schedule is given by $T = T_0, T_1, T_2, \ldots$, and $T_i = r_i T_{i-1}$, $r < 1$. At each temperature, we perturb the current solutions to search for a number of neighboring solutions for k times, where k is a user-defined value, and keep the best solution found so far. This process continues until the temperature is reduced to a "frozen" state or a predefined termination condition is reached. Then, the

best-found solution is reported. It is clear that the probability of accepting an "inferior" solution is higher if the cost increase is smaller and/or the current temperature T is higher. In practice, simulated annealing is often quite effective in searching for a desired solution. The whole simulated annealing process is an analogy of annealing an iron to produce a craft work, and it is where the name simulated annealing comes from. At a high temperature, the atoms of iron get more energy and thus have more freedom to move around. As the temperature reduces gradually, the atoms reach an equilibrium state step by step. As a result, the iron forms a desired shape. Algorithm 10.1 summarizes the generic algorithm of simulated annealing.

In addition to simulated annealing, we could also apply the **iterative method** (or **greedy search)** for floorplan designs. For this method, all uphill moves are rejected. The implementation is easier, and it typically can converge to a solution faster. However, it is very likely that this method gets stuck at some local optimum (see Figure 10.6 for an illustration). Consequently, its solution quality highly depends on the selected initial solution, and thus this approach is not as popular as simulated annealing for floorplan designs.

Algorithm 10.1 Simulated Annealing Algorithm for Floorplanning

```
1. Get an initial floorplan S;   S_Best = S;
2. Get an initial temperature T > 0;
3. while not "frozen" do
4.     for i = 1 to k do
5.       Perturb the floorplan to get a neighboring S' from S;
6.       ΔC = cost(S') – cost(S);
7.       if ΔC ≤ 0 then    // down-hill move
8          S = S';
9.       else    // uphill move
10.        S = S' with the probability e^(–ΔC/T);
11.      end if
12.      if cost(S_Best) > cost(S) then
13.        (S_Best) = S;
14.      end if
15.   end for
16.   T = rT;    // reduce temperature
17. end while
18. return S_Best;
```

There are four basic ingredients for simulated annealing: (1) **solution space**, (2) **neighborhood structure**, (3) **cost function (objective function)**, and (4) **annealing schedule**. A floorplan representation defines the solution space and the neighborhood structure, the cost function is defined by the

optimization goal, and the annealing schedule captures the temperature change during the annealing process.

10.2.2 **Normalized Polish expression for slicing floorplans**

In Section 10.1.3, a binary tree is used to model a slicing floorplan. We can record the binary tree by use of a **Polish expression** $E = e_1\, e_2 \ldots e_{2n-1}$ where $e_i \in \{1, 2, \ldots, n, \text{H}, \text{V}\}$. Here, each number denotes a module and H (V) represents a horizontal (vertical) cut in the slicing floorplan. The Polish expression is the **postfix ordering** of a binary tree, which can be obtained from the post-order traversal on a binary tree given in Algorithm 10.2. The length of E is $2n-1$, where n is the number of modules.

Because the Polish expression is the postfix order of a slicing tree, it has the **balloting property:** for every sub-expression $E_i = e_1 \ldots e_i$, $1 \leq i \leq 2n-1$, the number of operands is always larger than the number of operators.

As shown in Section 10.1.3, a slicing floorplan might induce multiple slicing trees, resulting in significant redundancies. Such redundancies will enlarge the solution space and complicate the search for a desired solution. Therefore, it is desired to prune such redundant solutions. As such, the **normalized Polish expression** E corresponding to the skewed slicing tree T is defined [Wong 1986]; a skewed slicing tree does not contain any node of the same cut type as its right child, and so does a normalized Polished expression, which does not contain consecutive operators of the same type, which is "HH" or "VV," in E. This results in a 1-1 correspondence between a set of skewed slicing trees with n modules and the corresponding set of normalized Polish expressions of length $2n-1$.

Algorithm 10.2 PostOrderTraversal(T)

```
1. /* Post-order traversal of a binary tree */
2. if root(T) ! = NULL then
3.    PostOrderTraversal(LeftSubtree(T));
4.    PostOrderTraversal(RightSubtree(T));
5.    Visit(root(T));
6. end if
7. return;
```

To transform a normalized Polish expression E to its corresponding floorplan F, we can use a bottom-up method to recursively combine the slicing sub-floorplans on the basis of E. There are two binary operators, H and V. If a and b are two modules or sub-floorplans, the expression abH implies to place a below b, and abV implies to place a to the left of b, as illustrated in Figure 10.7.The packing is performed by a post-order procedure. Each time, we combine two slicing sub-floorplans according to the operator type. For example, $E = 12\text{H}$ implies that module 1 is placed

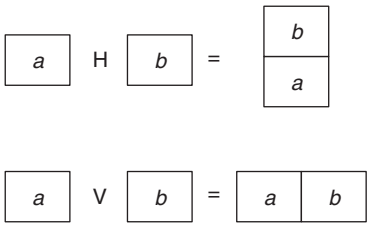

FIGURE 10.7

Binary operators, H and V, for slicing floorplans.

below module 2, and $E = 34\text{V}$ implies that module 3 is to the left of module 4. For $E = 123\text{H}\ldots$, because E is in the postfix expression, the operator H takes the operands 2 and 3, and module 2 is to the left of module 3.

Because E is in the postfix form, we can use a stack to facilitate the packing procedure (see Algorithm 10.3). Each time, we check an operand or an operator from E. If it is an operand, we push it into the stack; if it is an operator, we pop two operands from the stack and derive the new sub-floorplan based on the two operands and the operator. Then, the resulting sub-floorplan is treated as an operand and is pushed into the stack. This procedure continues until all operands/operators in E are processed, and the final floorplan is popped out from the stack.

Algorithm 10.3 Polish Expression Evaluation (E)

```
1. stack s;
2. for i = 1 to 2n-1 do
3.       if e_i is an operand then s.push(e_i);
4.       if e_i is an operator then
5.          a = s.pop( ); b = s.pop( ); c = a e_i b;
6.          s.push(c);
7.       end if
8. end for
9. return s.pop( );
```

Example 10.5 Given a binary tree shown in Figure 10.8, we can construct a Polish expression $E =$ 12H34H5HV based on the post-order traversal.

The balloting property is verified in the following table. For each column, the number of operands is always larger than that of operators. Further, $E = 12\text{H}34\text{H}5\text{HV}$ is a normalized Polish expression because there are no consecutive operators of the same type.

	1	*2*	*H*	*3*	*4*	*H*	*5*	*H*	*V*
No. of operands	1	2	2	3	4	4	5	5	5
No. of operators	0	0	1	1	1	2	2	3	4

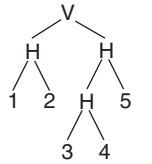

FIGURE 10.8

A binary tree and its Polish expression.

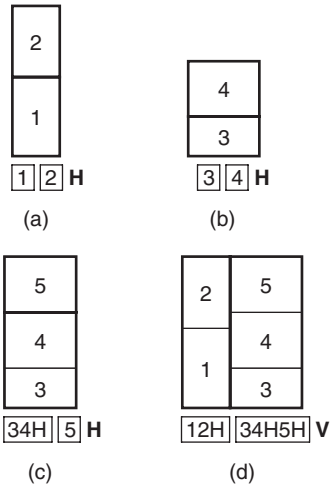

FIGURE 10.9

The packing process of the normalized Polish expression $E = 12H34H5HV$. (a), (b), (c), (d).

After obtaining the normalized Polish expression, $E = 12H34H5HV$ from the given slicing tree, Figure 10.9 gives steps to construct the corresponding floorplan from E. In step (a), we place module 1 below module 2 to obtain a slicing floorplan 12H. In step (b), we place module 3 below module 4 to obtain a slicing floorplan 34H. In step (c), we place the slicing floorplan 34H below module 5 to obtain the slicing floorplan 34H5H. In the final step, we place the sub-floorplan 12H to the left of the sub-floorplan 34H5H to obtain the final floorplan 12H34H5HV in Figure 10.9d.

10.2.2.1 *Solution space*

The set of all normalized Polish expressions forms the solution space. Given a normalized Polish expression with n operands (modules) and n–1 operators, the total number of combinations can be computed by the number of unlabeled binary trees with $2n$–1 nodes and the permutation of n labels. The permutation

of n labels is $n!$ From [Hilton 1991], the counting of an unlabeled p-ary tree with n node is given by

$$\frac{1}{(p-1)n+1}\binom{pn}{n}$$

Applying **Stirling's approximation**

$$n! = \Theta\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right)$$

and setting p to 2, we have the following asymptotic form

$$\begin{aligned}
&O\left(\frac{1}{(2-1)n+1}\binom{2n}{n}\right)\\
&= O\left(\frac{(2n)!}{(n+1)n!(2n-n)!}\right)\\
&= O\left(\frac{\sqrt{4\pi n}(2n/e)^{2n}}{(n+1)2\pi n(n/e)^{2n}}\right)\\
&= O\left(\frac{2^{2n}}{n^{1.5}}\right)
\end{aligned}$$

With the H/V label on internal nodes (2^{n-1}) and the permutation on external nodes ($n!$), the total number of possible skewed slicing floorplans (normalized Polish expressions) with n modules is

$$\begin{aligned}
&O\left(n!2^{n-1}\frac{2^{2n}}{n^{1.5}}\right)\\
&= O\left(n!\frac{2^{3n}}{n^{1.5}}\right)
\end{aligned}$$

Note that the upper bound is not tight. The reader can refer to [Shen 2003] for the derivation of the tighter bound of $\Theta(n!\ 2^{2.543n}/n^{1.5})$ for the total number of skewed slicing floorplans.

10.2.2.2 *Neighborhood structure*

Given a solution, we can perturb it to obtain a "neighboring" solution. The perturbation plays an important role in the search for a desired solution. For a normalized Polish expression, two operands are said to be **adjacent** if there is no operand between them. Two operators are said to be adjacent if there is no operand or operator between them. An operand and an operator are said to be adjacent if they are next to each other in E. A **chain** is a sequence of adjacent operators. For a normalized Polish expression, no consecutive operators of the same type are allowed, and thus there are only two types of chains, HVHVH... or VHVHV.... In other words, no chain can be HH... or VV....

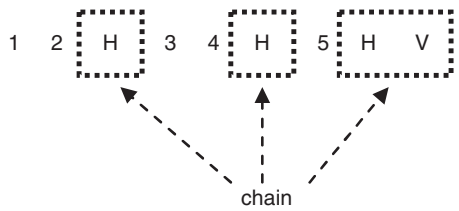

FIGURE 10.10

An example of the chains in a normalized Polish expression.

Example 10.6 In Figure 10.10, the operands 1 and 2 are adjacent, and so are the operands 3 and 4; the operand 3 and the operator H are also adjacent. There are three chains H, H, and HV in Figure 10.10.

We define three types of operations to perturb one normalized Polish expression to another.

Op1: Swap two adjacent operands.
Op2: Invert a chain by changing V to H and H to V.
Op3: Swap two adjacent operands and operators.

Performing Op1 and Op2 on a normalized Polish expression always produces a legal normalized Polish expression. However, Op3 could make the number of operands not greater than that of operators, which violates the balloting property, or it could generate two identical consecutive operators, which violates the property of a normalized Polish expression. As a result, we will only accept those Op3 operations that result in legal normalized Polish expressions. It turns out not to be difficult to check the legality of Op3. Assume that Op3 swaps the operand e_i and the operator e_{i+1}, $1 \leq i \leq k-1$. Then, the swap will not violate the balloting property if and only if $2N_{i+1} < i$, where N_k is the number of operators in the expression $E = e_1\, e_2 \ldots\, e_k$, $1 \leq k \leq 2n-1$.

Two normalized Polish expressions are said to be **neighboring** if one can be perturbed to another by use of one of the three operations. Furthermore, these three operations are sufficient to generate any normalized Polish expression from a given normalized Polish expression by a sequence of the preceding operations.

Example 10.7 Figure 10.11 gives an example of applying the three types of operations on the normalized Polish expression E to obtain its corresponding slicing floorplans. Given $E =$ 12H4H35VH and the modules' dimensions, the initial floorplan is shown in Figure 10.11a. Applying Op1 to swap the two adjacent operands 4 and 3, we obtain $E' =$ 12H3H45VH and the resulting floorplan as shown in Figure 10.11b. We then apply Op2 to invert the last chain VH to obtain $E'' =$ 12H3H45HV and its resulting floorplan

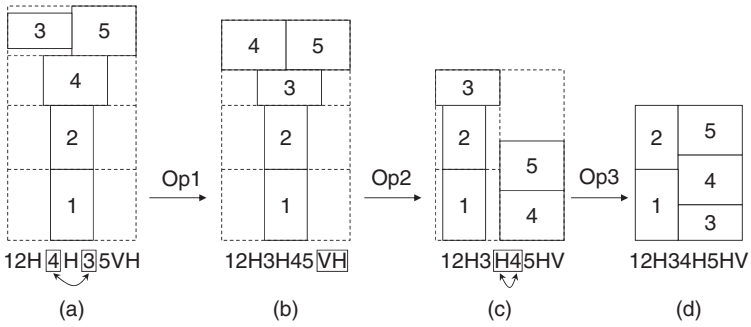

FIGURE 10.11

Illustrations of the perturbations in Example 10.7.

as shown in Figure 10.11c. We further apply Op3 to swap the adjacent operand and operator H4 to obtain $E''' = 12H34H5HV$. Consequently, we obtain the final floorplan as shown in Figure 10.11d, which has zero dead space.

10.2.2.3 *Cost function*

Area and wirelength are perhaps the two most commonly used costs for floorplan design. We can adopt the following cost function for simulated annealing:

$$Cost = \alpha \frac{A}{A_{norm}} + (1 - \alpha) \frac{W}{W_{norm}}$$

where A is the floorplan area, A_{norm} is the average area, W is the total wirelength, W_{norm} is the average wirelength, and α controls the weight between area and wirelength. To compute A_{norm} and W_{norm}, we can perturb the initial normalized Polish expression by use of the three operations for m times to obtain m floorplans and compute the average area A_{norm} and the average wirelength W_{norm} of these floorplans. The value m is proportional to the problem size.

To illustrate the area evaluation for a normalized Polish expression, we first construct the corresponding skewed slicing tree. All feasible floorplan implementations with the minimum areas are recorded in the corresponding nodes. Because module rotation is allowed, we might have two possible floorplan implementations, (w, h) and (h, w), for a module of the dimension (w, h). Of course, we have only one possible implementation for a square module. The floorplan size can be obtained in a bottom-up manner. Consider a non-leaf node with its left child of the dimension (w_1, h_1) and its right child of the dimension (w_2, h_2). If the cut type is H, the resulting implementation is $(\max(w_1, w_2), h_1 + h_2)$; if the cut type is V, the resulting implementation is $(w_1 + w_2, \max(h_1, h_2))$. For a node with two possible implementations in its left child and its right child each, it may generate four resulting implementations. For any two implementations $m_i = (w_i, h_i)$ and $m_j = (w_j, h_j)$, m_i **dominates** m_j if $w_i \leq w_j$ and $h_i \leq h_j$. In other words, the implementation m_j is **redundant,** because the implementation m_i gives a

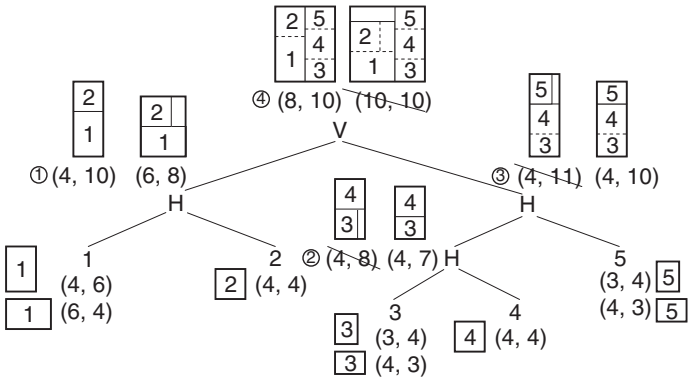

FIGURE 10.12

Dimension computation based on the slicing tree.

Table 10.1 The Dimensions of Modules in Example 10.8.

Module No.	Width	Height
1	4	6
2	4	4
3	3	4
4	4	4
5	3	4

floorplan solution of smaller area. By doing so, we can prune redundant implementations. Stockmeyer shows that the number of resulting irredundant slicing floorplan implementations after combining two nodes grows only linearly [Stockmeyer 1983]. (See Exercise 10.5.) Consequently, the area cost for a normalized Polish expression can be computed efficiently in polynomial time.

To compute the wirelength, we can only resort to approximation, because actual wiring is not performed yet at the floorplan stage. A popular wirelength approximation for a net is to measure its ***half-perimeter wirelength*** (HPWL). The HPWL is the half-perimeter length of the smallest bounding box that encloses all pins. If pin positions are not given, we can compute HPWL by use of the centers of modules.

Example 10.8 Find a floorplan implementation with the minimum area for the normalized Polish expression E = 12H34H5HV. The module dimensions are listed in Table 10.1.

First, we construct the corresponding skewed slicing tree and record the dimension candidates for each module with its corresponding leaf node. Because modules 2 and 4 are square, they have only one dimension candidate. The dimension candidates

Table 10.2 Area Evaluation for Example 10.8.

Step	Operator	Left child	Right child	Results
1	H	(4, 6) (6, 4)	(4, 4)	(4, 10) (6, 8)
2	H	(3, 4) (4, 3)	(4, 4)	~~(4, 8)~~ (4, 7)
3	H	(4, 7)	(3, 4) (4, 3)	~~(4, 11)~~ (4, 10)
4	V	(4, 10) (6, 8)	(4, 10)	(8, 10) ~~(10, 10)~~

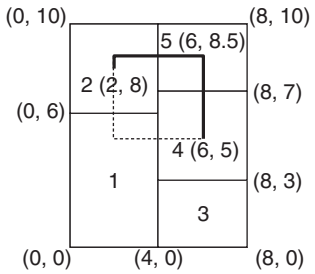

FIGURE 10.13

An example of the HPWL computation.

(irredundant implementations) for each internal node are updated in a post-order traversal order. See the steps in Table 10.2 and Figure 10.12. The resulting implementations that are dominated by others are crossed out.

Finally, the only irredundant implementation for the root is (8, 10), implying that the resulting floorplan has width = 8, height = 10, and area = 8 * 10 = 80. This floorplan has the minimum area for the normalized Polish expression E = 12H34H5HV with the given module dimensions. In case that we have more than one irredundant implementation for the root, we would pick the one with the minimum area.

Example 10.9 Given the floorplan in Figure 10.13, compute the HPWL of a net connecting modules 2, 4, and 5. The center coordinates of modules 2, 4, and 5 are (2, 8), (6, 5), and (6, 8.5), respectively. The height of the minimum bounding box is 3.5 and the width is 4. So the HPWL of this net is 3.5 + 4 = 7.5.

10.2.2.4 *Annealing schedule*

We can apply the **classical simulated annealing** algorithm for floorplanning described in Algorithm 10.4. The annealing schedule is $T = T_0, T_1, T_2, \ldots$, and $T_i = r_i T_{i-1}$, $r < 1$. The initial temperature is set to a high value so that the probability of accepting all perturbation is close to 1.0. We can compute the initial

temperature as follows. Before the simulated annealing process starts, we perturb the initial normalized Polish expression for a certain time to compute the average of all positive (uphill) cost change Δ_{avg}. Then, T_0 is initialized as $T_0 = -\Delta_{avg}/\ln P$, where P is the initial probability of accepting an uphill solution. We can set P very close to 1.0 (but certainly not 1.0). The temperature is reduced by a fixed ratio r at each iteration of annealing. The value 0.85 is recommended by most previous works. The larger the r, the longer the annealing time; however, a larger r often results in a better floorplan solution.

Algorithm 10.4 Wong-Liu Floorplanning (P, ε, r, k)

```
1. E = 12V3V4V ... nV;   // initial solution
2. E_Best = E; T = -Δ_avg / lnP;
3. do
4.    reject = 0;
5.    for ite = 0 to k do
6.       SelectOperation(Op);
7.       Case Op
8.       Op1: Select two adjacent operands e_i and e_j; E' = Swap(E, e_i, e_j);
9.       Op2: Select a nonzero length chain C; E' = Complement(E, C);
10.      Op3: done = FALSE
11.          while not (done) do
12.             Choice 1: Select two adjacent operand e_i and operator e_{i+1};
13.                if (e_{i-1} ≠ e_{i+1}) and (2N_{i+1} < i) then done = TRUE;
14.             Choice 2: Select two adjacent operator e_i and operand e_{i+1};
15.                if (e_i ≠ e_{i+2}) then done = TRUE;
16.         end while
17.         E' = Swap(E, e_i, e_{i+1});
18.      end case
19.      ΔCost = cost(E') – cost(E);
20.      if (ΔCost ≤ 0) or (Random < e-ΔCost/T)    then
21.         E = E';
22.         if cost(E) < cost(E_Best)   then   E_Best = E;
23.      else
24.         reject = reject + 1;
25.      end if
26.   end for
27.   T = rT;   // reduce temperature
28. until (reject/k > 0.95) or (T < ε) or (OutOfTime)
```

The excessive running time, however, is a significant drawback of the classical SA algorithm. To improve the efficiency of SA for searching for desired solutions, several annealing schemes of controlling the temperature changes during the annealing process have been proposed in the literature. The annealing

schedule used in TimberWolf [Sechen 1986, 1988] is one of the most popular schemes. It increases r gradually from its lowest value (0.8) to its highest value (approximately 0.95) and then gradually decreases r back to its lowest value.

Recently, a ***fast-simulated annealing*** (Fast-SA) scheme was proposed in [Chen 2005b]. The motivation is to reduce the number of uphill moves in the beginning, because most of the uphill moves at this stage lead to inferior solutions. Because it is not efficient and effective to accept too many uphill moves in the beginning, a greedy search can be applied to find a local optimum faster.

Starting with the local optimum, we then switch to normal SA. By doing so, it can save time for searching for desired solutions. To implement the preceding scheme, Fast-SA consists of three stages: (1) the high-temperature random search stage, (2) the pseudo-greedy local search stage, and (3) the hill-climbing search stage. At the first stage, the temperature is set to a very large value, and thus the probability of accepting an inferior solution approaches 1. This can avoid getting trapped in a local optimum in the very beginning. At the second stage, the temperature is set approaching zero to accept only a small number of inferior solutions. At the third stage, the temperature is raised to facilitate hill climbing to search for better solutions. The temperature reduces gradually, and very likely it finally converges to a desired solution. See Figure 10.14a and Figure 10.14b for the respective temperature changes over the search time with classical SA and Fast-SA.

Because the Fast-SA scheme saves significant iterations to explore the solution space, it could devote more time to finding better solutions in the hill-climbing stage. This makes the annealing much more efficient and effective. To implement this annealing scheme, we derive the temperature updating function T of Fast-SA by the following equations:

$$T_r = \begin{cases} \dfrac{-\Delta_{avg}}{\ln P} & r = 1 \\ \dfrac{T_1 \langle \Delta_{cost} \rangle}{rc} & 2 \leq r \leq k \\ \dfrac{T_1 \langle \Delta_{cost} \rangle}{r} & r > k \end{cases}$$

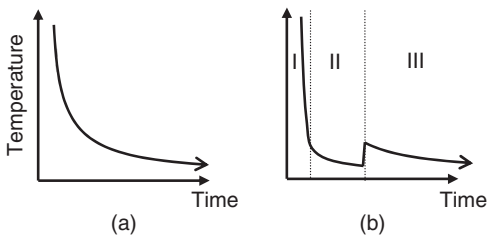

FIGURE 10.14

Temperature versus search time for (a) classical SA. (b) Fast-SA.

Here, r is the number of iterations, Δ_{avg} is the average uphill cost, P is the initial probability for accepting uphill solutions, $<\Delta_{cost}>$ is the average cost change (new cost − old cost) for the current temperature, and c and k are user-specified parameters. At the first iteration, the temperature is set according to the given initial probability P and the average uphill cost Δ_{avg}. Because P is usually set close to 1, it performs a random search to find a good solution. Then, it enters the pseudo-greedy local search stage until the kth iteration. Here, c is a user-defined parameter to control how low the temperature is in the second stage. We usually choose a large c to make $T \rightarrow 0$ such that it only accepts good solutions to perform pseudo-greedy searches. After k iterations, the temperature jumps up to further improve the solution quality. The value of $<\Delta_{cost}>$ affects the reduction rate of the temperature. If the cost of a neighboring solution changes significantly, $<\Delta_{cost}>$ is larger and thus the temperature reduces slower. In contrast, if $<\Delta_{cost}>$ is smaller, it implies that the cost of the neighboring solution only changes a little; in this case, we reduce the temperature more to reduce the number of iterations. Because the cost function is normalized to 1, this implies that $<\Delta_{cost}>$ is less than 1, and it ensures that the temperature is decreased. The number of iterations in the second stage can be determined by the problem size. The smaller the problem size, the smaller the k value. We can set $c = 100$ and $k = 7$ for typical floorplanning problems. Note that the initial temperature for Fast-SA is the same as that for the classical SA (*i.e.*, $T_1 = -\Delta_{avg}/\ln P$). The initial temperature T_1 needs to be kept high to avoid getting trapped into a local optimum in the very beginning.

10.2.3 B*-tree for compacted floorplans

B*-trees are based on ordered binary trees to model **compacted floorplans**. In a compacted floorplan, no modules can be moved toward left or bottom in the floorplan [Chang 2000]. Consequently, an area-optimal floorplan always corresponds to some B*-tree. Inheriting from the nice properties of ordered binary trees, B*-trees are very easy for implementation and can perform the three primitive tree operations **search**, **insertion**, and **deletion** in constant, constant, and linear time, respectively.

Unlike the slicing floorplan and its corresponding slicing tree(s), there exists a unique correspondence between a compacted floorplan and its induced B*-tree. In other words, given a compacted floorplan F, we can construct a unique B*-tree corresponding to F, and the packing corresponding to the B*-tree is the same as F. The nice property of the unique correspondence between a compacted floorplan and its induced B*-tree prevents the search space from being enlarged with redundant solutions and guarantees that an area-optimal placement can be found by searching on B*-trees. In the following, we describe the procedures for the transformation between a floorplan and a B*-tree.

10.2.3.1 *From a floorplan to its B*-tree*

First, we compact all modules to the left and bottom to obtain a compacted floorplan, because the B*-tree can only model compacted floorplans. A B*-tree is an ordered binary tree whose root corresponds to the module on the bottom-left corner. Similar to the ***depth-first-search*** (DFS) procedure, we construct a B*-tree T for a compacted floorplan in a recursive fashion: starting from the root, we first recursively construct the left sub-tree and then the right sub-tree. Let R_i be the set of modules located on the right-hand side and adjacent to module i. The left child of the node n_i corresponds to the lowest, unvisited module in R_i. The right child of n_i represents the lowest module located above and with the same x-coordinate as that of module i.

Example 10.10 Figure 10.15 shows an example of constructing a B*-tree from the floorplan given in Figure 10.15a. First, we compact all modules to the left and bottom to obtain a compacted floorplan as shown in Figure 10.15b. Module 1 is the root of the B*-tree, because it locates at the bottom-left corner. Module 2 is the lowest unvisited adjacent module on the right of module 1, so we make node n_2 the left child of node n_1. Module 3 has the same x-coordinate as that of module 1, and it is the lowest unvisited module above module 1, so we make node n_3 the right child of node n_1. Similarly, node n_4 is the left child of node n_3, node n_6 is the right child of node n_3, and node n_5 is the left child of node n_4.

10.2.3.2 *From a B*-tree to its floorplan*

Given a B*-tree T, its root represents the module on the bottom-left corner, and thus the coordinate of the module is $(x_{root}, y_{root}) = (0, 0)$. If node n_j is the left child of node n_i, module j is placed on the right-hand side and adjacent to module i, *i.e.*, $x_j = x_i + w_i$. Otherwise, if node n_j is the right child of n_i, module j is placed above module i, with the same x-coordinate as that of module j, *i.e.*, $x_j = x_i$. Therefore, given a B*-tree, the x-coordinates of all modules can be determined by traversing the tree once in linear time.

To efficiently compute the y-coordinates from a B*-tree, a **contour data structure** [Guo 2001] is used to facilitate the operations on modules. The

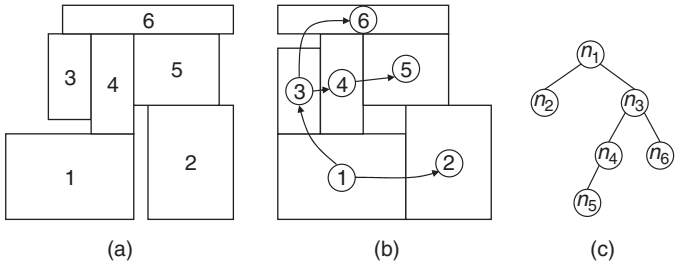

FIGURE 10.15

An example of (a) a given floorplan. (b) A compacted floorplan. (c) the corresponding B*-tree.

contour structure is a double-linked list storing the coordinates of the contour curve in the current compaction direction. A horizontal contour can reduce the running time for computing the y-coordinate of a newly inserted module. Without the contour, the running time for determining the y-coordinate of a newly inserted module would be linear to the number of modules. By maintaining the contour structure, however, the y-coordinate of a module can be computed in amortized $O(1)$ time.

Example 10.11 Find the floorplan corresponding to the B*-tree given in Figure 10.15c. The module dimensions are given in Table 10.3.

At first, there is no module. Therefore, we initialize the contour data structure $C = <(0,0)\ (\infty,0)>$. On the basis of the **depth-first order**, we pack modules one by one in six steps. The detailed processing is explained below, summarized in Table 10.4, and illustrated in Figure 10.16.

Step (a): Because n_1 is the **root**, the coordinate of module 1 is (0, 0). Inserting module 1 introduces three more contour points $C_{new} = <(0, 6), (9, 6), (9, 0)>$. To generate the new contour list, we need to find two sub-contour lists that are before and after the x-span [0, 9] of module 1: $C_{before} = <(0, 0)>$ and $C_{after} = <(\infty, 0)>$. The resulting contour $C = <C_{before}, C_{new}, C_{after}> = <(0, 0), (0, 6), (9, 6), (9, 0), (\infty, 0)>$.

Step (b): n_2 is the **left** child of n_1. Therefore, the x-coordinate of module 2 is $x_2 = x_1 + w_1 = 9$. To determine the y-coordinate of module 2, we search the contour to find the maximum y-coordinate between the x-span $[x_2, x_2 + w_2] = [9, 15]$. The maximum y-coordinate is 0, so we have $y_2 = 0$. Inserting module 2 introduces three more contour points $C_{new} = <(9, 8), (15, 8), (15, 0)>$. Again, we need to find two sub-contour lists that are before and after the x-span of module 1, [9, 15], to generate the new contour list: $C_{before} = <(0, 0), (0, 6), (9, 6)>$ and $C_{after} = <(\infty, 0)>$. The resulting contour $C = <C_{before}, C_{new}, C_{after}> = <(0, 0), (0, 6), (9, 6), (9, 8), (15, 8), (15, 0), (\infty, 0)>$.

Step (c): n_3 is the **right** child of n_1. Therefore, module 3 has the same x-coordinate as module 1. To determine the y-coordinate of module 3, we search the contour to

Table 10.3 The Dimensions of Modules in Example 10.11

Module No.	Width	Height
1	9	6
2	6	8
3	3	6
4	3	7
5	6	5
6	12	2

find the maximum y-coordinate in the x-span $[x_3, x_3 + w_3] = [0, 3]$. Because the maximum y-coordinate is 6, we have $y_3 = 6$. Inserting module 3 introduces three more contour points $C_{new} = <(0, 12), (3, 12), (3, 6)>$. We have the two sub-contour lists that are before and after the x-span of module 3, [0, 3]: $C_{before} = <(0, 0)>$ and $C_{after} = <(9, 6), (9, 8), (15, 8), (15, 0), (\infty, 0)>$. So the resulting contour $C = <C_{before}, C_{new}, C_{after}> = <(0, 0), (0, 12), (3, 12), (3, 6), (9, 6), (9, 8), (15, 8), (15, 0), (\infty, 0)>$.

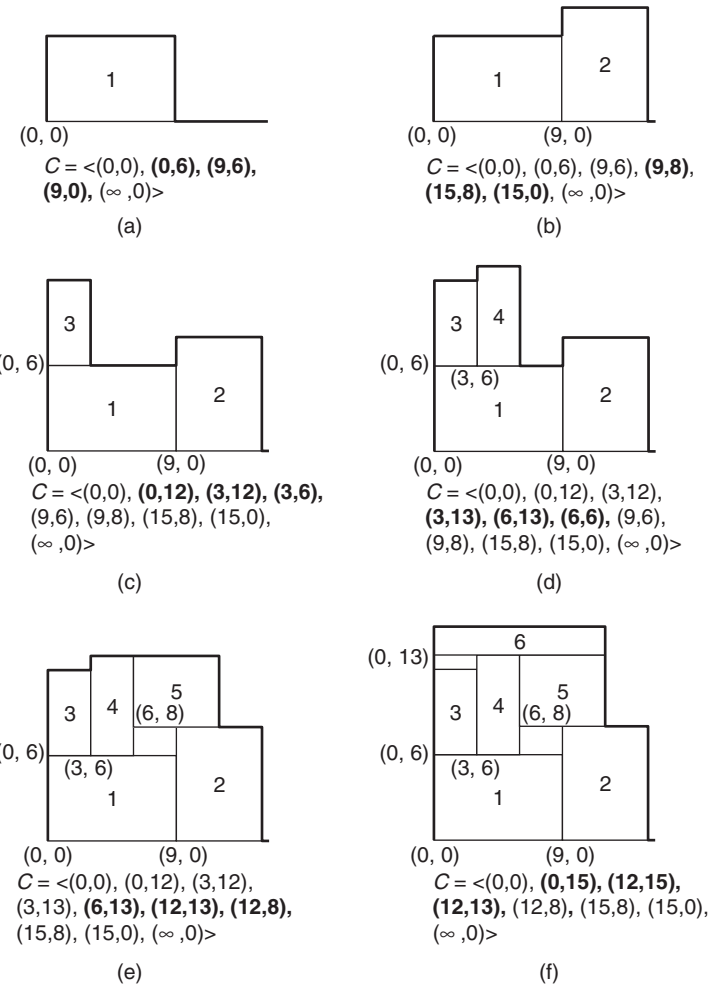

FIGURE 10.16

The B*-tree packing process. The double linked list C of the contour is shown below each figure. The horizontal contour lines are in bold.

Table 10.4 B*-Tree Packing for Example 10.11.

Step	Mod.	x-Coordinate	y-Coordinate	Contour C
Contour initialization				$<(0, 0), (\infty, 0)>$
(a)	1	$x1 = 0$	$y1 = 0$	$<(0, 0), (0, 6), (9, 6),(9, 0), (\infty, 0)>$
(b)	2	$x2 = x1 + w1 = 9$	$y2 = 0$	$<(0, 0), (0, 6), (9, 6), (9, 8), (15, 8), (15, 0), (\infty, 0)>$
(c)	3	$x3 = x1 = 0$	$y3 = \max(y1 + h1) = 6$	$<(0, 0), (0, 12), (3, 12), (3, 6), (9, 6), (9, 8), (15, 8), (15, 0), (\infty, 0)>$
(d)	4	$x4 = x3 + w3 = 3$	$y4 = \max(y1 + h1) = 6$	$<(0, 0), (0, 12), (3, 12), (3, 13), (6, 13), (6, 6), (9, 6), (9, 8), (15, 8), (15, 0), (\infty, 0)>$
(e)	5	$x5 = x4 + w4 = 6$	$y5 = \max(y1 + h1, y2 + h2) = 8$	$<(0, 0), (0, 12), (3, 12), (3, 13), (6, 13), (12, 13), (12, 8), (15, 8), (15, 0), (\infty, 0)>$
(f)	6	$x6 = x3 = 0$	$y6 = \max(y3 + h3, y4 + h4, y5 + h5) = 13$	$<(0, 0), (0, 15), (12, 15), (12, 13), (12, 8), (15, 8), (15, 0), (\infty, 0)>$

Step (d): n_4 is the **left** child of n_3. Therefore, $x_4 = x_3 + w_3 = 3$. To determine the y-coordinate of module 4, we search the contour to find the maximum y-coordinate between the x-span $[x_4, x_4 + w_4] = [3, 6]$. Because the maximum y-coordinate is 6, we have $y_4 = 6$. Inserting module 4 introduces three more contour points $C_{new} = <(3, 13), (6, 13), (6, 6)>$ and two sub-contour lists that are before and after the x-span of module 4, [3, 6]: $C_{before} = <(0, 0), (0, 12), (3, 12)>$ and $C_{after} = <(9, 6), (9, 8), (15, 8), (15, 0), (\infty, 0)>$. Consequently, the resulting contour $C = <C_{before}, C_{new}, C_{after}> = <(0, 0), (0, 12), (3, 12), (3, 13), (6, 13), (6, 6), (9, 6), (9, 8), (15, 8), (15, 0), (\infty, 0)>$.

Step (e): n_5 is the **left** child of n_4. Therefore, $x_5 = x_4 + w_4 = 6$. The maximum y-coordinate in the x-span $[x_5, x_5 + w_5] = [6, 12]$ is 8, and so is $y_5 = 8$. Inserting module 5 introduces the three contour points $C_{new} = <(6, 13), (12, 13), (12, 8)>$ and the two sub-contour lists before and after the x-span of module 5, [6, 12]: $C_{before} = <(0, 0), (0, 12), (3, 12), (3, 13)>$ and $C_{after} = <(15, 8), (15, 0), (\infty, 0)>$. The resulting contour $C = <C_{before}, C_{new}, C_{after}> = <(0, 0), (0, 12), (3, 12), (3, 13), (6, 13), (12, 13), (12, 8), (15, 8), (15, 0), (\infty, 0)>$.

Step (f): n_6 is the **right** child of n_3. Therefore, $x_6 = x_3 = 0$. The maximum y-coordinate in the x-span $[x_6, x_6 + w_6] = [0, 12]$ is 13, and so is $y_6 = 13$. Inserting module 6 introduces the three contour points $C_{new} = <(0, 15), (12, 15), (12, 13)>$ and the two sub-contour lists before and after the x-span of module 6, [0, 12]: $C_{before} = <(0, 0)>$ and $C_{after} = <(12, 8), (15, 8), (15, 0), (\infty, 0)>$. The resulting contour $C = <C_{before}, C_{new}, C_{after}> = <(0, 0), (0, 15), (12, 15), (12, 13), (12, 8), (15, 8), (15, 0), (\infty, 0)>$.

10.2.3.3 *Solution space*

The total number of B*-trees can be computed by the number of unlabeled binary trees and the permutation of n labels. The permutation of n labels is $n!$. From [Hilton 1991], the counting of an unlabeled p-ary tree with n node is

$$\frac{1}{(p-1)n+1}\binom{pn}{n}$$

Applying **Stirling's approximation**, we have

$$n! = \Theta\left(\sqrt{2\pi n}\left(\frac{n}{e}\right)^n\right)$$

Setting p to 2, we have the following asymptotic form:

$$O\left(\frac{2^{2n}}{n^{1.5}}\right)$$

Thus, the total number of possible floorplans for a B*-tree with n nodes is

$$O\left(n!\frac{2^{2n}}{n^{1.5}}\right)$$

10.2.3.4 *Neighborhood structure*

Each B*-tree corresponds to a floorplan. Therefore, the solution space consists of all B*-trees with the given nodes (modules). To find a neighboring solution, we can perturb a B*-tree to get another B*-tree by the following operations:

Op1: Rotate a module.
Op2: Move a node to another place.
Op3: Swap two nodes.

For Op1, we rotate a module for a B*-tree node, which does not affect the B*-tree structure. For Op2, we move a node to another place in the B*-tree. Op2 consists of two steps, **deletion** and **insertion**, which will be explained later. For Op3, we swap two nodes in the B*-tree. After packing for a B*-tree, we obtain a resulting floorplan.

There are three cases for the deletion operation. Note that in Cases 2 and 3, the relative positions of the modules might be changed after the operation, and thus we might need to reconstruct a corresponding floorplan for further processing.

- Case 1: A leaf node. We can simply delete the target leaf node.
- Case 2: A node with one child. We remove the target node and then place its only child at the position of the removed node. The tree update can be performed in $O(1)$ time.
- Case 3: A node with two children. We replace the target node n_t by either its right child or its left child n_c. Then we move a child of n_c, if any, to the original position of n_c. This process proceeds until the corresponding leaf node is handled. It is obvious that such a deletion operation requires $O(h)$ time, where h is the height of the B*-tree.

We can insert a new node into either an internal or an external position as follows.

- **Internal position:** A position between two nodes in a B*-tree.
- **External position:** A position pointed by a NULL pointer. Each node has two pointers, the left child and the right child. When the node has no left or right child, it points NULL.

Example 10.12 Figure 10.17 gives an example of the three types of operations on the B*-tree and its corresponding floorplans. (a) Rotate module 3 (Op1). It does not affect the B*-tree structure. (b) Move n_4 to the left child of n_6 (Op2). First, we delete n_4. Because n_4 has the only left child n_5, we attach n_5 to the left child of n_3. Then, we insert n_4 to the left child of n_6. (c) Swap n_1 and n_2 (Op3). Finally, we obtain the B*-tree and the corresponding floorplan in (d).

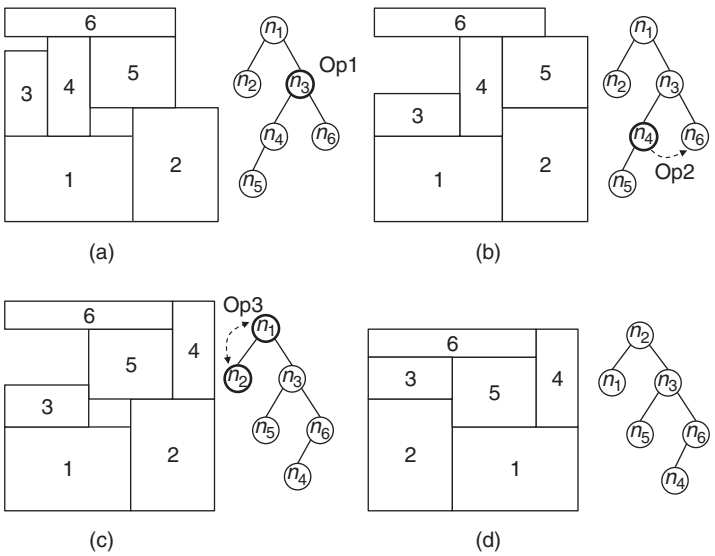

FIGURE 10.17

Illustration of the B*-tree perturbations.

10.2.3.5 *Cost function*

Similar to the normalized Polish expression, we can use the floorplan area and the total wirelength as the cost function of the simulated annealing:

$$Cost = \alpha \frac{A}{A_{norm}} + (1 - \alpha) \frac{W}{W_{norm}}$$

where A is the floorplan area, A_{norm} is the average area, W is the total wirelength, W_{norm} is the average wirelength, and α controls the weight between area and wirelength. (See Section 10.2.2 for the computation of A_{norm} and W_{norm}.) The area for a B*-tree can be computed by its width/height of the resulting floorplan, and HPWL can be used to evaluate the wirelength.

10.2.3.6 *Annealing schedule*

We can apply either classical SA or Fast-SA to B*-trees to find a desired floorplan. See Section 10.2.2 for more information on classical SA and Fast-SA.

10.2.4 Sequence pair for general floorplans

Sequence pair (SP) is a flexible representation to model a general floorplan [Murata 1995]. A sequence pair consists of an ordered pair of module name sequences. For example, (124536, 326145) can represent a floorplan of the six modules 1, 2, . . . , 6. In the following, we describe the procedures for the transformation between a floorplan and a sequence pair.

10.2.4.1 *From a floorplan to its sequence pair*

Given six modules shown in Figure 10.18a, we first stretch modules one by one to obtain **rooms**, each room containing only one module. Figure 10.18b shows the floorplan F with rooms derived from Figure 10.18a.

The following procedure encodes F by a pair of the module name sequences. For each module i, we draw two rectilinear curves, **right-up locus** and **left-down locus**. The right-up locus of module i is initially located at the center of module i and starts to move rightward. It turns its direction up and right alternately when it hits (1) the sides of rooms, (2) previously drawn lines, or (3) the boundary of the chip. The locus goes until it reaches the upper-right corner. The union of the two loci of module i forms the **positive locus** of module i. Figure 10.19a shows an example of positive loci. With the construction of positive loci, no two positive loci cross each other. Therefore, these positive loci can be linearly ordered, as well as the corresponding modules. Here we order the positive loci from left to right. Let Γ_+ be the module name sequence in this order. In Figure 10.19a, the first sequence $\Gamma_+ = 124536$ is obtained.

Negative loci can be obtained similar to the positive loci. The difference is that a negative locus is the union of the **up-left locus and down-right locus.** Let Γ_- be the module name sequence in the order of the negative loci from left

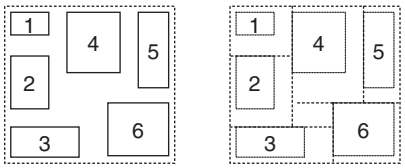

FIGURE 10.18

(a) Given modules. (b) A floorplan of the "rooms."

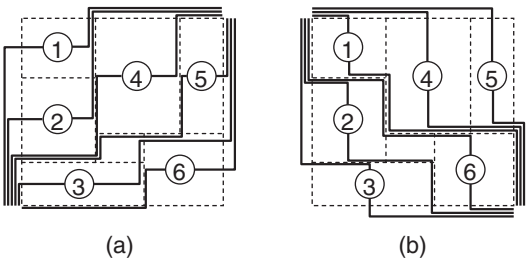

FIGURE 10.19

(a) Positive loci. (b) Negative loci.

to right. An example of negative loci is shown in Figure 10.19b. As a result, we have the negative loci from left to right and obtain $\Gamma_- = 326145$. Finally, the sequence pair $(\Gamma_+, \Gamma_-) = (124536, 326145)$ is obtained.

10.2.4.2 *From a sequence pair to its floorplan*

Given a sequence pair (Γ_+, Γ_-), the geometric relation of modules can be derived from the sequence pair as follows:

Rule 1 (horizontal constraint): module i is left to module j if i appears before j in both Γ_+ and Γ_- $(\ldots i \ldots j \ldots, \ \ldots i \ldots j \ldots)$.

Rule 2 (vertical constraint): module i is below module j if i appears after j in Γ_+ and i appears before j in Γ_- $(\ldots i \ldots j \ldots, \ \ldots i \ldots j \ldots)$.

The following steps describe a procedure to transform a sequence pair to its floorplan. Consider an $n \times n$ grid, where n is the number of modules. Label the horizontal grid lines and the vertical grid lines with module names along Γ_+ and Γ_- from top and from left, respectively. A cross point of the horizontal grid line of label i and the vertical grid line of label j is referred to by (i, j). Then, rotate the resulting grid counterclockwise by 45 degrees to get an oblique grid. (See Figure 10.20.) Place each module i with its center being on (i, i). See Figure 10.20 for an example.

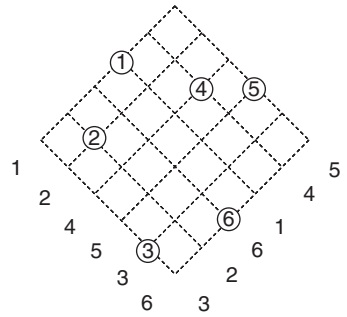

FIGURE 10.20

An oblique grid for (Γ+, Γ–) = (124536, 326145).

On the basis of the preceding constraints, we can create a **horizontal-constraint graph** with a source and a sink, and a node-weighted directed acyclic graph $G_H(V, E)$, where V is the set of nodes, and E is the set of edges as follows:

- V: source s, sink t, and n nodes labeled with module names.
- E: (s, i) and (i, t) for each module i, and (i, j) if and only if module i is on the left of module j (horizontal constraint).
- Nodes weight: zero for s and t, width of module i for node i.

Similarly, a **vertical-constraint graph** $G_V(V, E)$ can be constructed on the basis of the vertical constraints and the height of each module. Note that both the horizontal and the vertical constraint graphs are acyclic. If two modules i and j are in horizontal relation, then there is an edge between nodes i and j in G_H, and thus they do not overlap horizontally in the resulting floorplan. Similarly, if modules i and j are in vertical relation, they do not overlap vertically. Because any pair of modules is either in horizontal or vertical relation, no modules overlap with each other in the resulting floorplan.

The module locations can be obtained from the constraint graphs. The x-coordinate of module i is given by the longest path length from the source s to node i in G_H. Similarly, the y-coordinate of module i can be computed on G_V. Consequently, the width and the height of the resulting floorplan can be computed by the longest path length between the source and the sink in G_H and G_V, respectively. The longest path length computation on each node-weighted directed acyclic graph, G_H or G_V, can be performed in $O(n^2)$ time by applying the well-known **longest path algorithm** [Lawler 1976], where n is the number of modules. In other words, given a sequence pair (Γ_+, Γ_-), the area-optimal packing can be obtained in quadratic time.

For the example sequence pair $(\Gamma_+, \Gamma_-) = (124536, 326145)$, we can construct the corresponding G_H and G_V shown in Figure 10.21. Table 10.5 lists the module dimensions. The weight and the width (height) of each module are indicated in each node of G_H (G_V). As mentioned earlier, the x-coordinates

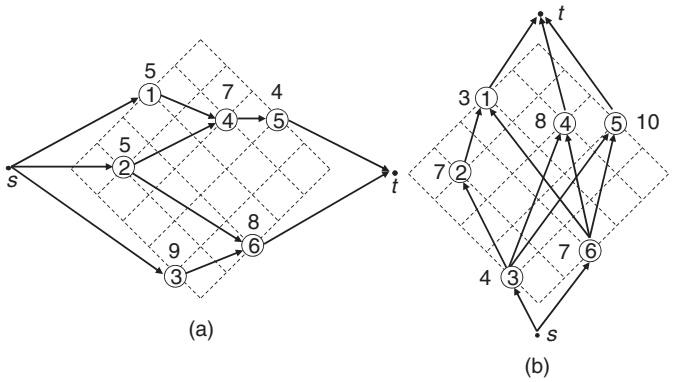

FIGURE 10.21

The constraint graphs with the source *s* and sink *t* induced from the sequence pair (124536, 326145): (a) The horizontal constraint graph G_H. (b) The vertical constraint graph G_V. (Note that the existence of the edges (*a, b)* and (*b, c*) implies that the transitive edge (*a, c*) is also in the constraint graph. Transitive edges are not shown in both graphs for simplicity.)

Table 10.5 The Dimensions of the Modules in Figure 10.21

Module No.	Width	Height
1	5	3
2	5	7
3	9	4
4	7	8
5	4	10
6	8	7

can be computed by the longest path length from the source in $G_{H,}$ and thus we have

Module 1: $x_1 = 0$
Module 2: $x_2 = 0$
Module 3: $x_3 = 0$
Module 4: $x_4 = \max(x_1 + w_1, x_2 + w_2) = \max(0 + 5, 0 + 5) = 5$
Module 5: $x_5 = x_4 + w_4 = 5 + 7 = 12$
Module 6: $x_6 = \max(x_2 + w_2, x_3 + w_3) = \max(0 + 5, 0 + 9) = 9$
Floorplan width $= \max(x_5 + w_5, x_6 + x_6) = \max(12 + 4, 9 + 8) = 17$.

The *y*-coordinates can be computed from G_V similarly as follows:

Module 1: $y_1 = \max(y_2 + h_2, y_6 + h_6) = \max(4 + 7, 0 + 7) = 11$

Module 2: $y_2 = y_3 + h_3 = 4$
Module 3: $y_3 = 0$
Module 4: $y_4 = \max(y_3 + h_3, y_6 + h_6) = \max(0 + 4, 0 + 7) = 7$
Module 5: $y_5 = \max(y_3 + h_3, y_6 + h_6) = \max(0 + 4, 0 + 7) = 7$
Module 6: $y_6 = 0$
Floorplan height $= \max(y_1 + h_1, y_4 + h_4, y_5 + h_5) = \max(11 + 3, 7 + 8, 7 + 10) = 17$.

The resulting floorplan is shown in Figure 10.22, and the coordinate of each module and the resulting floorplan dimension are as follows:

Module 1 = (0, 11)
Module 2 = (0, 4)
Module 3 = (0, 0)
Module 4 = (5, 7)
Module 5 = (12, 7)
Module 6 = (9, 17)
Floorplan dimension = (17, 17)

10.2.4.3 *Solution space*

Each permutation of Γ_+ and Γ_- gives a floorplan solution. For n modules, the lengths of Γ_+ and Γ_- are both n, and thus each of Γ_+ and Γ_- have $n!$ permutations. Consequently, there are $(n!)^2$ total permutations for a sequence pair with n modules.

10.2.4.4 *Neighborhood structure*

To search for a desired floorplan solution, we can use the following three types of operations to perturb a sequence pair to another:

Op1: Rotate a module.
Op2: Swap two module names in only one sequence.
Op3: Swap two module names in both sequences.

Each of the three operations results in a legal sequence pair and floorplan solution. Furthermore, these three operations are sufficient to generate any sequence pair from a given sequence pair by a sequence of operations.

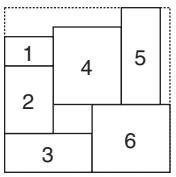

FIGURE 10.22

The minimal area packing result of (Γ+, Γ–) = (124536, 326145).

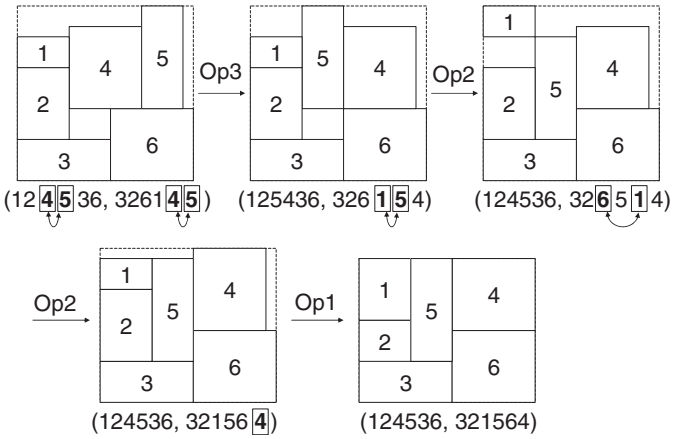

FIGURE 10.23

Effects of the perturbations in sequence pairs.

Figure 10.23 gives an example of these types of operations on the sequence pair and its resulting floorplans.

10.2.4.5 *Cost function*

Again, we can use the floorplan area and the total wirelength as the cost function of the simulated annealing:

$$Cost = \alpha \frac{A}{A_{norm}} + (1 - \alpha) \frac{W}{W_{norm}}$$

where A is the floorplan area, A_{norm} is the average area, W is the total wirelength, W_{norm} is the average wirelength, and α controls the weight between area and wirelength. (See Section 10.2.2 for the computation of A_{norm} and W_{norm}.) The area for a sequence pair can be computed by its width/height (longest path length in G_H/G_V) of the corresponding floorplan, and HPWL can be used to evaluate the wirelength.

10.2.4.6 *Annealing schedule*

Again, we can apply either classical SA or Fast-SA based on sequence pair to find a desired floorplan solution. See Section 10.2.2 for classical SA or Fast-SA.

10.2.5 Floorplan representation comparison

In addition to normalized Polish expression, B*-tree, and sequence pair, there are quite a few popular floorplan representations, such as ***BSG*** [Nakata 1996], ***O-tree*** [Guo 1999], ***Corner Block List*** (CBL) [Hong 2000], ***Transitive Closure Graph*** (TCG) [Lin 2001], ***TCG-S*** [Lin 2002], ***Corner Sequence*** (CS) [Lin 2003], ***Twin Binary Sequences*** (TBS) [Young 2003], ***Adjacent Constraint Graph*** (ACG) [Zhou 2004], etc. Some representations are closely related. For example, B*-tree is equivalent to O-tree, yet with faster operations,

Table 10.6 Comparison among Floorplan Representations

Representation	Solution Space	Packing Time	Flexibility
Normalized Polish Expression	$O(n!2^{3n}/n^{1.5})$	$O(n)$	Slicing
Corner Block List	$O(n!2^{3n})$	$O(n)$	Mosaic
Twin Binary Sequence	$O(n!2^{3n}/n^{1.5})$	$O(n)$	Mosaic
O-tree	$O(n!2^{2n}/n^{1.5})$	$O(n)$	Compacted
B*-tree	$O(n!2^{2n}/n^{1.5})$	$O(n)$	Compacted
Corner Sequence	$\leq (n!)^2$	$O(n)$	Compacted
Sequence Pair	$(n!)^2$	$O(n^2)$	General
BSG	$O(n!C(n^2, n))$	$O(n^2)$	General
Transitive Closure Graph	$(n!)^2$	$O(n^2)$	General
TCG-S	$(n!)^2$	$O(n \lg n)$	General
Adjacent Constraint Graph	$O((n!)^2)$	$O(n^2)$	General

simpler data structures, and higher flexibility in handling various placement constraints. TCG and sequence pair are also equivalent but their induced operations are significantly different.

Table 10.6 summarizes the sizes of the solution spaces, packing times, and flexibility of the popular floorplan representations. Among the representations, sequence pair, TCG, TCG-S, and ACG can represent general floorplans; O-tree, B*-tree, and corner sequence can model only compacted floorplans; CBL and TBS model the floorplan with each room containing exactly one module, called the mosaic floorplan; and normalized Polish is restricted to slicing floorplans. The general floorplan has the highest flexibility, followed by the compacted floorplan, then followed by the mosaic floorplan, and the slicing floorplan has the least flexibility. (For tighter bounds of slicing, mosaic, and general floorplans, please see [Shen 2003].)

For the packing time, sequence pair, TCG, and ACG require $O(n^2)$ time to generate a floorplan, where n is the number of modules. Note that sequence pair can reduce its packing time to $O(n \lg \lg n)$ time based on the longest common subsequence technique [Tang 2001]. TCG-S needs $O(n \lg n)$ time for packing. For O-tree, B*-tree, corner sequence, and the normalized Polish expression, the packing time is only linear time mainly because they keep relatively simpler information in their data structures.

As a remark for floorplan representations, the evaluation of a floorplan representation should be made based on at least the following three criteria: (1) the definition/properties of the representation, (2) its induced solution structure (not merely the size of its solution space), and (3) its induced operations. We shall avoid the pitfall that judges a floorplan representation by only one of the aforementioned three criteria alone; for example, claiming a floorplan

representation A is superior to another floorplan representation B simply because A has a smaller solution space and a faster packing time. Here is an analogy: the representation itself is like the body of an automobile, the induced operations are like the wheels of the automobile, and the solution structure is like the highway network. An automobile with its body alone can go nowhere. For a comprehensive study of floorplan representations, similarly, we shall evaluate them from at least all the aforementioned three criteria.

10.3 ANALYTICAL APPROACH

In addition to simulated annealing, we can resort to the **analytical approach** to floorplan designs [Sutanthavibul 1991]. The analytical approach is a mathematical programming formulation that includes an objective function and a set of constraints. For the floorplanning problem, we need to consider two sets of basic constraints: (1) the **module nonoverlapping constraint** and (2) the **dimension constraint**.

Two modules, i and j are said to be **nonoverlapping**, if at least one of the following cases (linear constraints) is satisfied:

		p_{ij}	q_{ij}
Case 1: i to the left of *j*	$x_i + w_i \le x_j$	0	0
Case 2: i below *j*	$y_i + h_i \le y_j$	0	1
Case 3: i to the right of *j*	$x_i - w_j \ge x_j$	1	0
Case 4: i above *j*	$y_i - h_j \ge y_j$	1	1

where two binary variables, p_{ij} and q_{ij}, are introduced to denote that one of the above inequalities is enforced. For example, when $p_{ij} = 0$ and $q_{ij} = 1$, the inequality equation $y_i + h_i \le y_j$ is enforced.

Let W and H be upper bounds of the width and height of the floorplan, respectively. We have the following linear constraints for module nonoverlap:

$$x_i + w_i \le x_j + W(p_{ij} + q_{ij})$$

$$y_i + h_i \le y_j + H(1 + p_{ij} - q_{ij})$$

$$x_i - w_j \ge x_j - W(1 - p_{ij} + q_{ij})$$

$$y_i - h_j \ge y_j - H(2 - p_{ij} - q_{ij})$$

where

$$1 \le i \le j \le n$$

For the dimension constraint, each module must be enclosed within a rectangle of the width W and the height H of the floorplan. Specifically, we have

$$x_i + w_i \leq W$$

$$y_i + h_i \leq H$$

where

$$1 \leq i \leq j \leq n$$

Our objective is to minimize the floorplan area, xy, where x and y are the width and height of the resulting floorplan, respectively. Notice that the area xy is non-linear, and it is much harder to solve a non-linear system than a linear one. To transform the original non-linear objective into a linear one, we can approximate the problem by fixing the floorplan width W and minimizing the height y. As a result, we need to modify the dimension constraints to $x_i + w_i \leq W$ and $y_i + h_i \leq y$, where $1 \leq i \leq n$ and y is the height of the current floorplan. In summary, we have the following four types of constraints:

1. There is no overlap between any two modules ($\forall i, j : 1 \leq i < j \leq n$).
2. Each module is enclosed within a rectangle of width W and height H ($x_i + w_i \leq W$, $y_i + h_i \leq y$, $1 \leq i \leq n$). Here, w_i and h_i are known.
3. $x_i \geq 0,\ y_i \geq 0,\ 1 \leq i \leq n$
4. $p_{ij}, q_{ij} \in \{0, 1\}$

On the basis of the above discussions, we can formulate the floorplan designs as the following ***mixed integer linear program*** (**MILP**). Note that both the objective function and all constraints are linear. Our floorplan design problem is to minimize the height y for a given bound of the floorplan width W, subject to the following system of inequality constraints:

Minimize	y	
subject to	$x_i + w_i \leq W$	$1 \leq i \leq n$
	$y_i + h_i \leq y$	$1 \leq i \leq n$
	$x_i + w_i \leq x_j + W(p_{ij} + q_{ij})$	$1 \leq i < j \leq n$
	$y_i + h_i \leq y_j + H(1 + p_{ij} - q_{ij})$	$1 \leq i < j \leq n$
	$xi - w_j \geq x_j - W(1 - p_{ij} + q_{ij})$	$1 \leq i < j \leq n$
	$y_i - h_j \geq y_j - H(2 - p_{ij} - q_{ij})$	$1 \leq i < j \leq n$
	$x_i, y_i \geq 0$	$1 \leq i \leq n$
	$p_{ij}, q_{ij} \in \{0, 1\}$	$1 \leq i < j \leq n$

For the size of the mixed ILP for n modules, the number of continuous variables is $O(n)$, the number of integer variables is $O(n^2)$, and the number of linear constraints is $O(n^2)$. There are a few popular mixed ILP solvers, such as ***GLPK***

[GLPK 2008], ***CPLEX*** [ILOG 2008], ***LINDO*** [LINDO 2008], ***lp_solve*** [lp_solve 2008], etc. ILP has the exponential time complexity in the worst case, and thus it is time-consuming for problems of large sizes. To cope with problems of large sizes, methods such as the divide-and-conquer and the progressive approaches are often used. We will elaborate on this issue later.

The preceding formulation does not consider the rotation of modules. We can extend the aforementioned MILP formulation by introducing a new binary variable r_i to consider the rotation of the module i. When $r_i = 0$, module i is not rotated (*i.e.,* rotated by 0 degree); when $r_i = 1$, module i is rotated by 90 degrees. The system of inequality constraints now becomes

$x_i + r_i h_i + (1 - r_i)w_i \leq W$	$1 \leq i \leq n$
$y_i + r_i w_i + (1 - r_i)h_i \leq y$	$1 \leq i \leq n$
$x_i + r_i h_i + (1 - r_i)w_i \leq x_j + M(p_{ij} + q_{ij})$	$1 \leq i < j \leq n$
$y_i + r_i w_i + (1 - r_i)h_i \leq y_j + M(1 + p_{ij} - q_{ij})$	$1 \leq i < j \leq n$
$x_i - r_j h_j - (1 - r_j)w_j \geq x_j - M(1 - p_{ij} + q_{ij})$	$1 \leq i < j \leq n$
$y_i - r_j w_j - (1 - r_j)h_j \geq y_j - M(2 - p_{ij} - q_{ij})$	$1 \leq i < j \leq n$
$x_i, y_i \geq 0$	$1 \leq i \leq n$
$r_i, p_{ij}, q_{ij} \in \{0, 1\}$	$1 \leq i < j \leq n$

where $M = \max\{W, H\}$. The following gives an example of the MILP formulation for floorplan design. (The preceding formulation considers only the area optimization. If wirelength also needs to be considered, we need to modify the objective function to minimize the total wirelength.)

Example 10.13 Given the dimensions of modules listed in Table 10.7. The total module area is 8 * 6 + 8 * 5 + 11 * 2 = 110. Because a square floorplan is often desired, the square root of 110 is approximately 10. Therefore, we set $W = 10$ and $M = \max\{8, 6\} + \max\{8, 5\} + \max\{11, 2\} = 27$ to find a floorplan with width less than 10. We can use the publicly available lp_solve program to solve this problem. The Figure 10.24 shows the input file in the lp-format. The objective is to minimize the floorplan height (y). The constraints c_1 to c_6 define the bounding box of the floorplan, and the constraints c_7 to c_{10}, c_{11} to c_{14},

Table 10.7 The Dimensions of Modules in Example 10.13

Module No.	Width (w_i)	Height (h_i)
1	8	6
2	8	5
3	11	2

```
min: y;

c1: x1 +  6 r1 +  8 -  8 r1 <= 10;
c2: x2 +  5 r2 +  8 -  8 r2 <= 10;
c3: x3 +  2 r3 + 11 - 11 r3 <= 10;
c4: y1 +  8 r1 +  6 -  6 r1 <= y;
c5: y2 +  8 r2 +  5 -  5 r2 <= y;
c6: y3 + 11 r3 +  2 -  2 r3 <= y;

c7:  x1 + 6 r1 + 8 - 8 r1 <= x2 + 27 p12 + 27 q12;
c8:  y1 + 8 r1 + 6 - 6 r1 <= y2 + 27 + 27 p12 - 27 q12;
c9:  x1 - 5 r2 - 8 + 8 r2 >= x2 - 27 + 27 p12 + 27 q12;
c10: y1 - 8 r2 - 5 + 5 r2 >= y2 - 54 + 27 p12 + 27 q12;

c11: x1 +  6 r1 +  8 -  8 r1 <= x3 + 27 p13 + 27 q13;
c12: y1 +  8 r1 +  6 -  6 r1 <= y3 + 27 + 27 p13 - 27 q13;
c13: x1 -  2 r3 - 11 + 11 r3 >= x3 - 27 + 27 p13 + 27 q13;
c14: y1 - 11 r3 -  2 +  2 r3 >= y3 - 54 + 27 p13 + 27 q13;

c15: x2 +  5 r2 +  8 -  8 r2 <= x3 + 27 p23 + 27 q23;
c16: y2 +  8 r2 +  5 -  5 r2 <= y3 + 27 + p23 - 27 q23;
c17: x2 -  2 r3 - 11 + 11 r3 >= x3 - 27 + 27 p23 + 27 q23;
c18: y2 - 11 r3 -  2 +  2 r3 >= y3 - 54 + 27 p23 + 27 q23;

r1 <= 1;
r2 <= 1;
r3 <= 1;
p12 <= 1;
q12 <= 1;
p13 <= 1;
q13 <= 1;
p23 <= 1;
q23 <= 1;

int r1, r2, r3, p12, q12, p13, q13, p23, q23;
```

FIGURE 10.24

The input file for lp_solve to minimize y in Example 10.13.

and c_{15} to c_{18} define the nonoverlapping relationship between modules 1 and 2, modules 1 and 3, and modules 2 and 3, respectively. The remaining constraints define the 0-1 integer variables.

Applying lp_solve to solve the preceding MILP program, we can obtain the outputs shown in Figure 10.25, which gives the co-ordinates of modules: Module 1 $(x_1, y_1) = (0, 5)$; module 2 $(x_2, y_2) = (0, 0)$; module 3 $(x_3, y_3) = (8, 0)$. Only module 3 is rotated ($r_3 = 1$). The resulting floorplan height is 11 ($y = 11$), and the final floorplan in shown in Figure 10.26.

As mentioned earlier, the time complexity of MILP is exponential, and thus it is prohibitively expensive for problems of large sizes. To cope with problems of large sizes, methods such as the progressive and the divide-and-conquer approaches are often used to reduce the problem sizes. We will examine a **progressive augmentation method** that solves a partial problem at each step to reduce the floorplanning complexity. Each time, we select a set of modules and place them into the current partial floorplan, as illustrated in Figure 10.27.

```
Value of objective function: 11

Actual values of the variables:
y                              11
x1                              0
r1                              0
x2                              0
r2                              0
x3                              8
r3                              1
y1                              5
y2                              0
y3                              0
p12                             1
q12                             1
p13                             0
q13                             0
p23                             0
q23                             0
```

FIGURE 10.25

The outputs from lp_solve in Example 10.13.

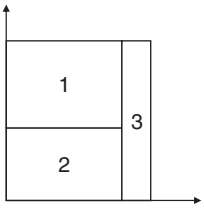

FIGURE 10.26

The resulting floorplan in Example 10.13.

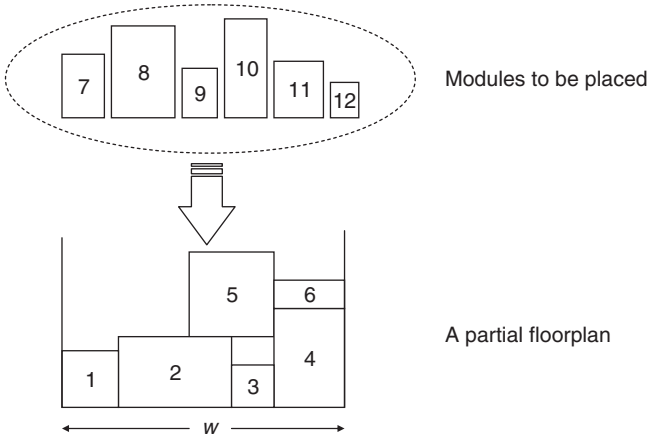

FIGURE 10.27

Floorplanning with an existing partial floorplan.

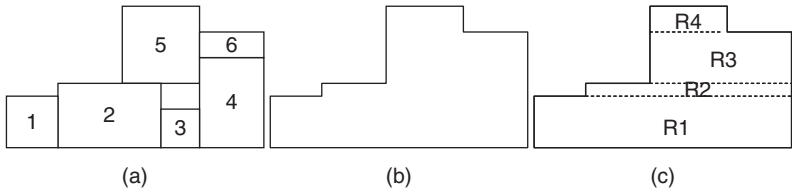

FIGURE 10.28

Reducing the problem size by a set of covering rectangles: (a) The original partial floorplan. (b) The outline of the partial floorplan. (c) A set of rectangles covering the partial floorplan.

To reduce the problem size, we limit the number of the modules to be placed at each step and also minimize the problem size of the current partial floorplan. We can replace the already placed modules by a set of covering rectangles. Figure 10.28 illustrates the procedure for obtaining these rectangles. First, we find the outline of the six placed modules, as shown in Figure 10.28b. The dead spaces among the placed modules are also enclosed in the outline, because it is impossible for the newly added modules to use them. Then, we horizontally dissect the outline into rectangles, R1, R2, R3, and R4. By doing so, the number of rectangles is usually much smaller than that of placed modules, and so are the number of variables and constraint in the MILP formulation.

Besides the preceding MILP-based floorplanning, a sophisticated analytical floorplanning method was proposed [Zhan 2006]. It first roughly determines the module positions by uniformly distributing modules. Then, the overlaps are gradually removed in the second stage to obtain the final floorplan. This approach has much better scalability to handle large-scale designs. However, this approach cannot guarantee a nonoverlap floorplan solution.

10.4 MODERN FLOORPLANNING CONSIDERATIONS

Increasing design complexity and new circuit properties and requirements have reshaped the modern floorplanning problem. The new considerations and challenges make the problem much more difficult. In this section, we will discuss such crucial considerations. Specifically, we will focus on (1) soft modules, (2) fixed-outline constraints, and (3) large-scale floorplanning, and then highlight other important issues for modern floorplanning.

10.4.1 Soft modules

Unlike hard modules with fixed heights and widths, **soft modules** can change their heights and widths while keeping the same module area. The aspect ratio bounds are given as inputs for each module. There are many techniques for the adjustment of

soft-module dimensions. In the following, we introduce an effective and efficient heuristic that adjusts soft-module dimensions to optimize the chip area. The underlying concept of this sizing method is to align the module width/height to its adjacent module to reduce the dead space [Chang 2000; Chi 2003; Chen 2005a, 2006].

Given a set B of modules, we assume that module i's bottom-left coordinate is (x_i, y_i) and its top-right coordinate is $(x_i + w_i, y_i + h_i)$. Each soft module has four candidates for the dimensions (*i.e.*, shapes). The candidates are defined as follows:

- $R_i = x_a + w_a - x_i$, where $x_a + w_a = \min \{x_k + w_k \mid x_k + w_k > x_i + w_i, k \in B\}$
- $L_i = x_b + w_b - x_i$, where $x_b + w_b = \max \{x_k + w_k \mid x_k + w_k < x_i + w_i, k \in B\}$
- $T_i = x_c + h_c - y_i$, where $x_c + h_c = \min \{x_k + h_k \mid x_k + h_k > x_i + h_i, k \in B\}$
- $B_i = x_d + h_d - y_i$, where $x_d + h_d = \max \{x_k + h_k \mid x_k + h_k < x_i + h_i, k \in B\}$

Define the **aspect ratio** of a module as the ratio of the height over width of the module. After determining the candidates of the module shapes, we may change the shape of a soft module i by choosing one of the following five choices during simulated annealing:

1. Change the width of module i to R_i.
2. Change the width of module i to L_i.
3. Change the height of module i to T_i.
4. Change the height of module i to B_i.
5. Change the aspect ratio of module i to a random value in the range of the given soft aspect ratio constraint.

We can add the module resizing as one floorplan perturbation operation during simulated annealing so that the module shapes could be changed to obtain a more desired floorplan.

Example 10.14 Consider the example of soft-module resizing given in Figure 10.29. Module 4 has four shape candidates, R_4, L_4, T_4, and B_4, with four candidate lines being shown in Figure 10.29a. If we stretch the right boundary of module 4 to R_4 (the height is also changed correspondingly to maintain a fixed area), it can generate a more compacted floorplan as shown in Figure 10.29b.

The preceding soft-module sizing technique can be applied to any floorplan representations based on simulated annealing or iterative improvement. For the normalized Polish expression (slicing tree), we can use a more sophisticated method, **shape curve,** to handle soft modules. Because the area of a soft module is fixed, the shape function of a module is a hyperbola: $wh = A$, or $h = A/w$, where w is the width, h is the height, and A is the area of the

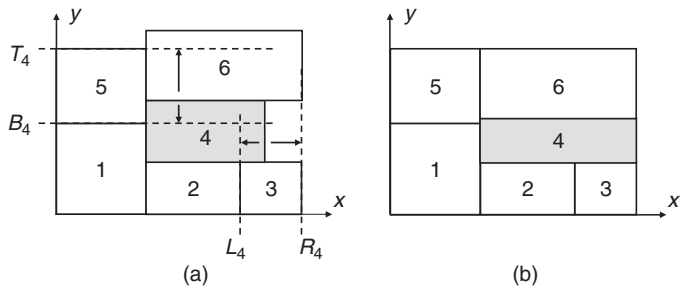

FIGURE 10.29

A soft-module resizing example: (a) the original floorplan with four shape candidates for resizing module 4. (b) a compacted floorplan by stretching the right boundary of module 4 to R_4.

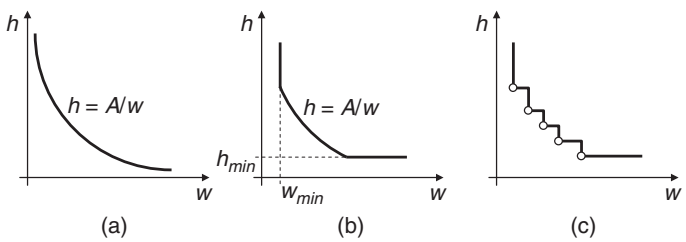

FIGURE 10.30

The shape curve of a module: (a) The shape curve in a hyperbola function. (b) The shape curve with the minimum width/height constraint. (c) The piecewise linear shape curve.

module. See Figure 10.30a for an example of the shape curve. Because module width and height are usually constrained to avoid very thin modules, $h \geq h_{\min}$ and $w \geq w_{\min}$; see Figure 10.30b for the resulting shape curve. In practice, we can use piecewise linear functions to record the shape curve for easier implementation. We only need to record the **corner points** of the shape curve, as shown in Figure 10.30c.

The shape curves can record not only the shapes of a basic soft module, but also that of a composite module formed by a set of basic modules (*i.e.,* a sub-floorplan). In a slicing tree, we first generate a shape curve for each module and record this shape curve with the corresponding leaf node, as shown in Figure 10.31. Then, the shape curve of a composite module can be derived from its children nodes and recorded in the corresponding internal node. By use of the bottom-up procedure, we can find the shape curve of the root node, which gives all possible shapes of the resulting floorplans.

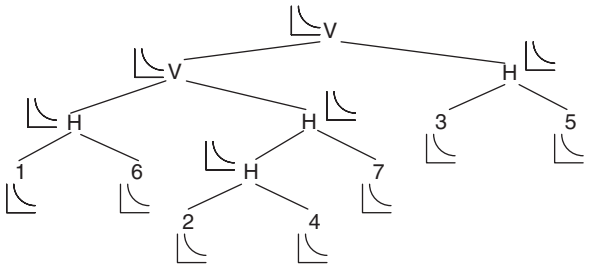

FIGURE 10.31

Shape curves in a slicing tree.

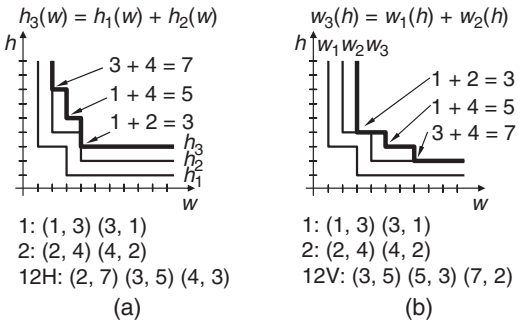

FIGURE 10.32

Examples of updating the shape curve: (a) The H operator. (b) The V operator.

Example 10.15 Given two piecewise linear shape curves for modules 1 and 2, derive the shape curves of the composite modules 12H and 12V. Figure 10.32a illustrates the derivation of the shape curve for the composite module 12H. For the H operator, two modules are merged vertically; we have $h_3(w) = h_1(w) + h_2(w)$, and the minimum width of the resulting floorplan cannot be smaller than max(min w_1, min w_2). See the bold lines in Figure 10.32a for the shape curve of 12H. Similarly, for the V operator, we have $w_3(h) = w_1(h) + w_2(h)$ and the height of the resulting floorplan cannot be smaller than max(min h_1, min h_2). The shape curve of 12V is represented by the bold lines in Figure 10.32b.

10.4.2 **Fixed-outline constraint**

Modern VLSI design is typically based on a **fixed-die (fixed-outline) floorplan** [Kahng 2000], rather than a **variable-die floorplan**. A floorplan with pure area minimization without any fixed-outline constraints may be useless, because it cannot fit into the given outline. Unlike classical floorplanning that usually handles only module packing to minimize silicon area, modern floorplanning should be formulated as **fixed-outline floorplanning**.

The fixed-outline constraint is given as follows. We first construct a fixed outline with the aspect ratio R^* (*i.e.*, height/width). For a collection of modules with the given total area A and the **maximum percentage of dead space** Γ, we have the chip area $= H^*W^* = (1 + \Gamma)A$ and the chip aspect ratio $= H^*/W^* = R^*$. Therefore, the new height H^* and width W^* of the outline are defined by the following equations [Adya 2001]:

$$H^* = \sqrt{(1+\Gamma)AR^*}$$

$$W^* = \sqrt{(1+\Gamma)A/R^*}$$

Example 10.16 Figure 10.33 gives three floorplan examples with different R^*'s and 's. The three floorplans contain the same modules. Figure 10.33a and Figure 10.33b have the same maximum percentage of dead spaces, 15%, yet with different outline ratios, 2.0 and 1.0, respectively. Figure 10.33b and Figure 10.33c have the same outline ratio, 1.0, yet with different maximum percentages of dead spaces, 15% and 50%, respectively.

To handle the fixed-outline constraint, we will modify the cost function for simulated annealing. In addition to the wirelength/area objective, we may add an aspect ratio penalty to the cost function [Chen 2005a, 2006]. The rationale is that if the aspect ratio of the floorplan is similar to that of the outline, and the dead space of the floorplan is smaller than the maximum percentage of the dead space Γ, then the floorplan can fit into the outline. Suppose that the current aspect ratio of the floorplan is R. We define the cost function Φ for a floorplan solution F by the following equation:

$$\Phi(F) = \alpha A + \beta W + (1 - \alpha - \beta)(R^* - R)^2$$

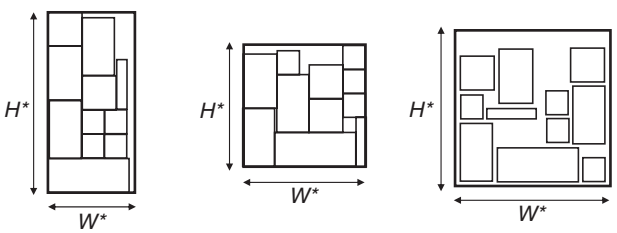

FIGURE 10.33

Three floorplans with different outline ratios (R*) and the maximum percentages of dead spaces (Γ) based on the same modules.

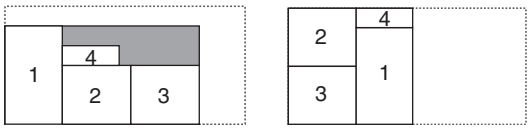

FIGURE 10.34

Examples of: (a) a floorplan with the aspect ratio the same as the one of the outline. (b) the optimal floorplan with a different aspect ratio from the aspect ratio of the outline.

where A is the floorplan area, W is the wirelength, R is the floorplan aspect ratio, R^* is the desired floorplan aspect ratio, and α and β are user-defined parameters.

The best aspect ratio of the floorplan in the fixed outline may not be the same as that of the outline, as shown in Figure 10.34. In this case, we will decrease the weight of the aspect ratio penalty to concentrate more on the wirelength/area optimization. We can use an adaptive method to control the weights in the cost function according to the most recent floorplans found in [Chen 2005a]. If there are more feasible floorplans in most recent floorplans found during simulated annealing, it implies that this instance is easier to be fit into the floorplan outline, and thus we will reduce the weight of the aspect ratio penalty to focus more on the wirelength/area optimization. Figure 10.35 shows the resulting floorplans for the MCNC circuit ami49 with various aspect ratios. There are 49 modules in this circuit [Chen 2005a].

In addition to the objective function adjustment, new perturbations can also be applied to better guide a local search for fixed-outline floorplanning on the basis of sequence pair or normalized Polish expression [Adya 2003; Lin 2004a]. However, unlike the objective function adjustment that can be applied to all floorplan representations, the new perturbations are specific to the target floorplan representation. On the basis of the generalized slicing tree, DeFer is proposed to handle fixed-outline floorplanning efficiently and effectively [Yan 2008]. DeFer generates a collection of possible floorplan solutions and chooses the best one that can fit into the fixed outline with the smallest wirelength at the last stage.

10.4.3 **Floorplanning for large-scale circuits**

As technology advances, the number of modules in a chip becomes larger. Simulated annealing alone cannot handle large-scale floorplanning instances effectively and efficiently. To cope with the **scalability** problem, **hierarchical** floorplanning is proposed. The hierarchical approach recursively divides a floorplan region into a set of sub-regions and solves those sub-problems independently. Patoma is a fast hierarchical floorplanner based on recursive bipartitioning [Cong 2005]. It partitions a floorplan and uses ***row-oriented block*** (ROB) packing and

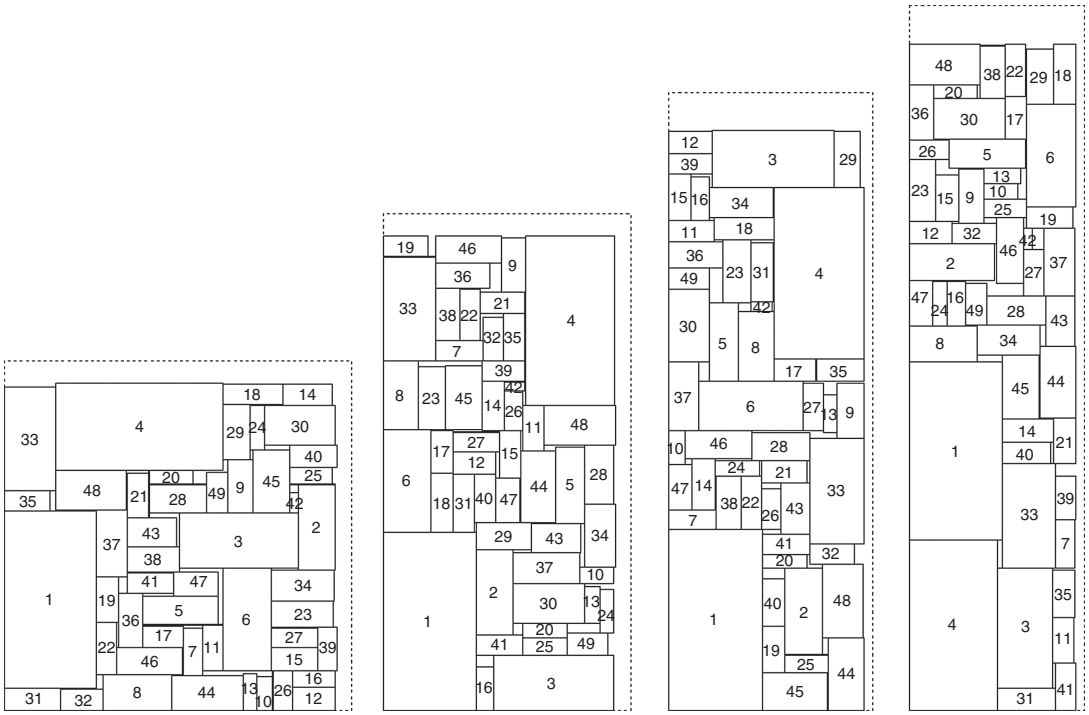

FIGURE 10.35

The floorplans of the MCNC circuit ami49 with fixed-outline ratios 1, 2, 3, and 4.

zero-dead space (ZDS) floorplanning to find legal sub-floorplans. The top-down, hierarchical technique is efficient in handling large-scale problems. Nevertheless, a significant drawback of the hierarchical approach is that it might lack the global information for the floorplanning interactions among different sub-regions, because each sub-region is processed independently. As a result, the hierarchical approach might not find desired solutions.

To remedy the deficiency, **multi-level floorplanning** is proposed to find a better trade-off between the scalability and solution quality. The multi-level framework applies a two-stage technique, **bottom-up coarsening** and **top-down uncoarsening**. We take the MB*-tree [Lee 2003] as an example to explain the concept of multi-level floorplanning. Figure 10.36 shows the MB*-tree multi-level framework based on a two-stage technique of bottom-up **clustering** (coarsening) followed by top-down **declustering** (uncoarsening). It should be noted that although we use the MB*-tree as an example to explain the multi-level floorplanning framework, this framework itself is general to all floorplan representations.

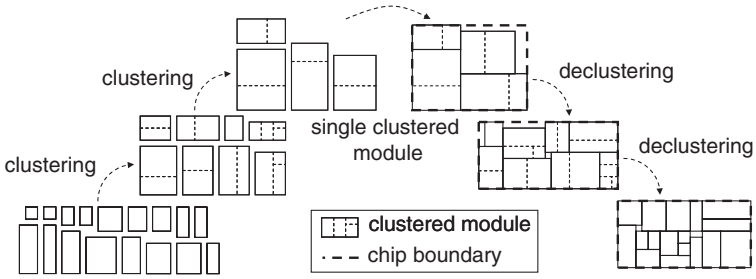

FIGURE 10.36

Multilevel floorplanning that uses recursive clustering and declustering.

The **clustering** stage iteratively groups a set of (primitive or composite) modules (say, two modules) on the basis of 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 that can be handled by a classical floorplanner), denoted by a one-node B*-tree that records the entire clustering scheme. During clustering, we will record how two modules i and j are clustered into a new composite module k. 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.

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 on the basis of a simulated annealing scheme. The refinement should 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 multi-level B*-tree–based floorplanner preserves the geometric relations among modules during declustering (*i.*e., the tree expansion), which makes the B*-tree an ideal data structure for the multi-level floorplanning framework.

The MB*-tree algorithm is summarized in Algorithm 10.5. 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 with simulated annealing.

Algorithm 10.5 MB*-tree Floorplanning

Input: A set of modules and a set of nets.
Output: A final area-optimized floorplan.
Stage I: Clustering

1. **while** the number of modules/clusters is still large
2. Cluster modules according to their dimensions and connectivity;
3. **end while**;

Stage II: Declustering
4. **while** still having clusters
5. Decluster a set of clusters;
6. Perform simulated annealing to refine the floorplan;
7. **end while**;
8. **return** the final floorplan;

Figure 10.37 illustrates the MB*-tree algorithm. For easier explanation, we cluster three modules each time in Figure 10.37. Figure 10.37a lists seven modules to be packed, i's, $1 \leq i \leq 7$. Figure 10.37b to Figure 10.37d illustrate the clustering process. Figure 10.37b shows the resulting configuration after clustering modules 5, 6, and 7 into a new cluster module 8 (*i.e.*, the clustering scheme of 8 is {{5, 6}, 7}); note that the B*-tree for the packing of modules 5, 6, and 7 is recorded with module 8. Similarly, we cluster modules 1, 2, and 4 into module 9 with the clustering scheme {{2, 4}, 1} and record the B*-tree with module 9 for packing modules 1, 2, and 4. Finally, we cluster modules 3, 8, and 9 into module 10 by use of the clustering scheme {{3, 8}, 9} and record a one node B*-tree for module 10. The clustering stage is thus done, and the declustering stage begins, in which simulated annealing is applied to the floorplanning. In Figure 10.37e, we first decluster module 10 into modules 3, 8, and 9 (*i.e.*, expand the node n_{10} into the B*-subtree illustrated in Figure 10.37e). We then refine the solution by moving module 8 to the top of module 9 (perform Op2 on n_8) during simulated annealing (see Figure 10.37f). As shown in Figure 10.37g, we further decluster module 9 into modules 1, 2, and 4, and then rotate module 2 and move module 3 on top of module 2 (perform Op1 on n_2 and Op2 on n_3), resulting in the configuration shown in Figure 10.37h. Finally, we decluster module 8 shown in Figure 10.37i to modules 5, 6, and 7, and move module 4 to the right of module 3 (perform Op2 on n_4), which results in the area optimal floorplan shown in Figure 10.37j.

Figure 10.38 shows the layout for the circuit ami49_200 with 9800 modules and 81,600 nets (not shown in the layout) [Lee 2003] obtained by MB*-tree. It has a dead space of only 3.44%. Without the use of the multi-level approach,

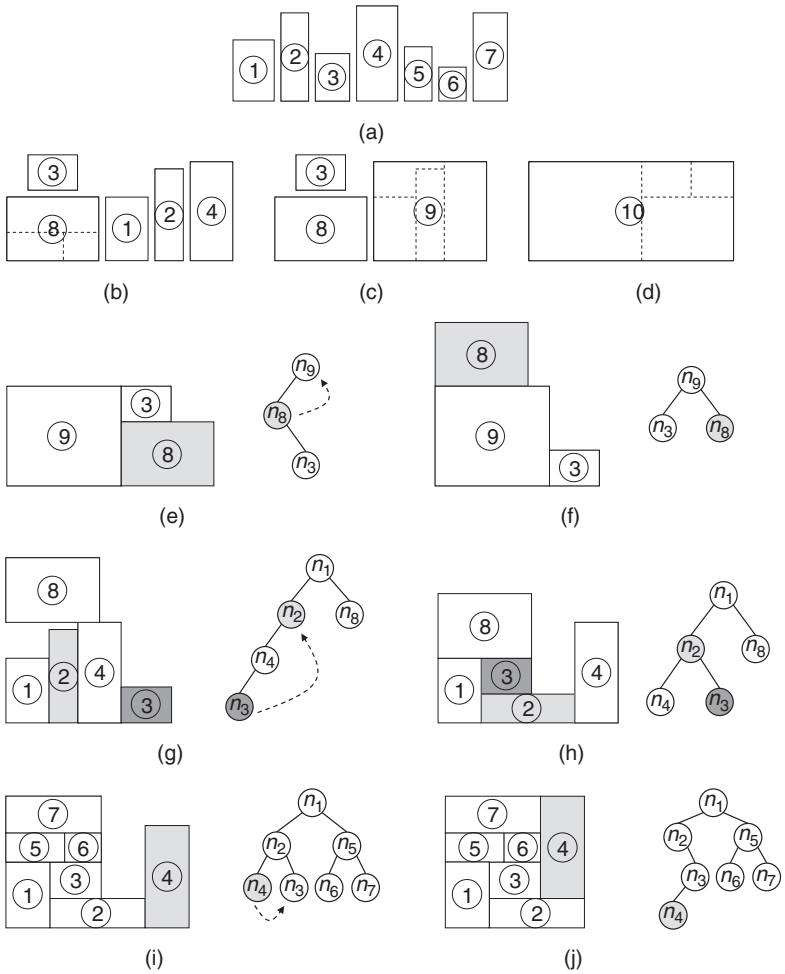

FIGURE 10.37

Clustering and declustering: (a) given seven *i*s, $1 \leq i \leq 7$. (b) Cluster modules 5, 6, and 7 into 8. (c) Cluster modules 1, 2, and 4 into 9. (d) Cluster modules 3, 8, and 9 into 10. (e) Decluster module 10 into modules 3, 8, and 9. (f) Perform Op2 on module 8. (g) Decluster module 9 into modules 1, 2, and 4. (h) Perform Op1 and Op2 on modules 2 and 3, respectively. (i) Decluster module 8 into module 5, 6, and 7. (j) Perform Op2 on module 4 to obtain the final floorplan.

the flat floorplanning method could not handle large circuits of this magnitude effectively.

The MB*-tree approach is referred to the **∧-shaped multi-level framework,** because it starts with bottom-up coarsening (clustering) followed by

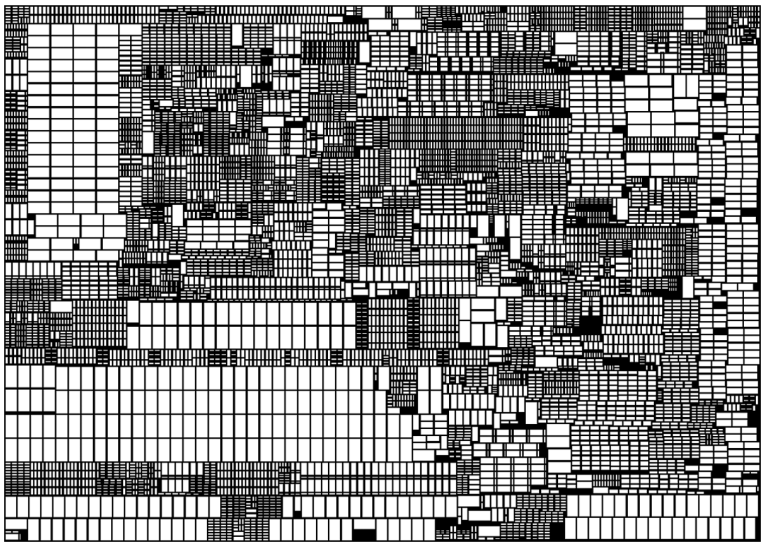

FIGURE 10.38

The resulting floorplan for the circuit ami49_200 with 9800 modules and 81,600 nets; the resulting dead space (the dark regions) is only 3.44%.

top-down uncoarsening (declustering). In contrast, the **V-shaped multi-level framework** works from top-down uncoarsening (partitioning) followed by bottom-up coarsening (merging) [Chen 2005b]. The V-shaped multi-level framework often outperforms the ∧-shaped one in the optimization of global circuit effects, such as interconnection optimization, because the V-shaped framework considers the global configuration first and then processes down to local ones level by level and thus the global effects can be handled at earlier stages.

10.4.4 Other considerations and topics

In addition to the aforementioned modern floorplanning considerations, there are many other issues that might need to be considered. In the following, we briefly describe these issues.

Modern circuit designs often need to integrate analog and digital circuits on a single chip and thus may suffer from **substrate noise coupling**. A pioneering work along this direction was proposed in [Cho 2006]. With the continued increase in system frequency and design complexity, existing techniques for reducing substrate noise may need to be enhanced substantially. Considering substrate noise in early floorplanning is now desirable.

For nanometer VLSI designs, **interconnect** dominates overall circuit performance. However, the conventional design flow often deals with interconnect

optimization at the routing or post-routing stages. When the interconnect complexity grows drastically, it is often too late to perform aggressive interconnect optimization during or after routing, because most silicon and routing resources are occupied. Therefore, it is desirable to optimize interconnect as early as possible. Many techniques have been proposed for interconnect optimization. Some examples are wiring topology construction, buffer/repeater insertion and sizing, and wire sizing and spacing [Cong 1997]. Among these interconnect optimization techniques, **buffer insertion** is generally considered the most effective and popular technique to interconnect delay reduction, especially for global signals [Alpert 1997]. With so many buffers being added, the buffer positions should be planned as early as possible to ensure timing closure and design convergence; in particular, current VLSI designs often do not allow buffers to be inserted inside a circuit module, because they consume silicon resources and require connections to the power/ground network. Consequently, buffers are placed in channels and dead spaces of the current floorplan and are often clustered to form buffer blocks between existing circuit modules of the floorplan, which inevitably increases the chip area [Cong 1999]. It is thus desirable to carefully plan for the buffers during/after floorplanning to minimize the area overhead and facilitate routing, which is referred to as the **buffer block planning**. Furthermore, long interconnects affect microarchitecture designs very much, because multiple cycles are necessary to communicate global signals across the chip. As a result, it is desirable to handle **microarchitecture aware floorplanning** considering interconnect pipelining to improve the performance of microarchitecture designs [Jagannathan 2005; Ma 2007].

Because interconnection on the chip becomes more congested as technology advances, bus routing becomes a challenging task. Because buses have different widths and go through multiple modules, the positions of the modules greatly affect the bus routing. To make the bus routing easier, bus planning should be considered at the floorplanning stage, which is called **bus-driven floorplanning**. The feasibility conditions of bus-driven floorplanning for sequence pair [Xiang 2003] and B*-tree [Chen 2005a] are studied to reduce their solution spaces and find the desired floorplans efficiently. When the number of modules through which a bus goes is large, multi-bend bus structure can be used to find better solutions [Law 2005].

As technology advances, the metal width decreases, whereas the global wirelength increases. This trend makes the resistance of the power wire increase substantially. Therefore, floorplanning considering **voltage (IR) drop** in the ***power/ground*** **(P/G) network** becomes important. Because of IR-drop, supply voltage in logic may not be an ideal reference. An important problem of P/G network synthesis is to use the minimum amount of wiring area for a P/G network under the power integrity constraints such as IR drop and electromigration. As the design complexity increases dramatically, it is necessary to handle the IR-drop problem earlier in the design cycle for better design convergence. Most existing commercial tools deal with the IR-drop problem at

the post-layout stage, when entire chip design is completed and detailed layout and current information are known. It is, however, often very difficult and computationally expensive to fix the P/G network synthesis at the post-layout stage. Therefore, researchers started to consider the P/G network analysis at an earlier design stage [Yim 1999; Wu 2004; Lin 2007].

Recently, **3-D floorplanning** was developed to handle dynamically reconfigurable ***field programmable gate arrays*** (FPGAs) to improve logic capacity by time-sharing. We may use the 3-D space (x, y, t) to model a dynamically reconfigurable system. The x and y coordinates represent the 2-D plane of FPGA resources (spatial dimension), whereas the t coordinate represents the time axis (temporal dimension). Each "task" [***Reconfigurable Functional Unit Operation*** (RFUOP), the execution unit in a reconfigurable FPGA] is modeled by a rectangular box (module). We may denote each module as a 3-D box with the spatial dimensions x and y and the temporal dimension t. Figure 10.39a shows a program with four parts of codes to be mapped into RFUOPs. Because of the capacity constraint, we may not load all modules into the device at the same time. Therefore, it is desirable to consider the 3-D floorplanning problem of placing these modules into the ***Reconfigurable Functional Unit*** (RFU) (see Figure 10.39b). The objective is to allocate modules to optimize the area and execution time and to satisfy specified constraints. To deal with the 3-D floorplanning problem, a few 3-D floorplan representations extending the 2-D floorplan ones are proposed. For example, **Sequence Triple** [Yamazaki 2003] and **Sequence Quintuple** [Yamazaki 2003] are extensions of sequence pair for 2-D packing. **K-tree** [Kawai 2005], **T-tree** [Yuh 2004a], and **3D-subTCG**

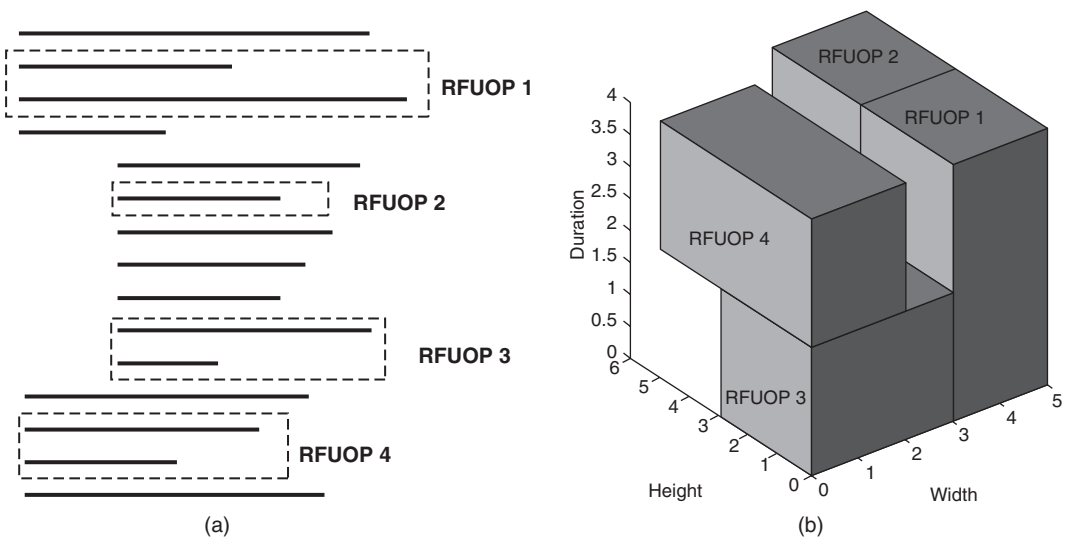

FIGURE 10.39

(a) A running program. (b) A 3-D floorplan of the running program.

[Yuh 2004b] are extensions of O-tree [Guo 1999], B*-tree [Chang 2000], and ***Transitive Closure Graph*** (TCG) [Lin 2001] for 2-D packing, respectively. Furthermore, heat dissipation is the most critical challenge of system-in-package design, sometimes called 2.5-D IC's (discrete layers are added into the traditional x and y spatial dimensions). Layer partitioning followed by 2-D floorplanning is often adopted to handle the thermal constraints for the 2.5-D IC designs [Cong 2004].

In addition to the floorplanning for VLSI modules, the floorplanning techniques can also be applied to other problems, such as system-on-chip test scheduling [Wu 2005] and digital microfluidic biochip placement [Yuh 2006].

10.5 CONCLUDING REMARKS

Floorplanning is an essential design step for hierarchical, building-module design methodology. It provides valuable insights into the hardware decisions and estimation of various costs. The most popular floorplanning method resorts to the modeling of the floorplan structure and then optimizes the floorplan solutions with simulated annealing. There exist many floorplan representations in the literature. Yet, normalized Polish expression, B*-tree, and sequence pair have been recognized as the most valuable representations because of their superior simplicity, effectiveness, efficiency, and flexibility.

In additional to simulated annealing, analytical floorplanning approaches have shown their advantage in the effective wirelength optimization [Zhan 2006]; however, it is harder to handle the module overlaps and the fixed-outline constraint for such an approach. Floorplanning considering both hard and soft modules is also more challenging for the analytical approach.

After floorplanning, all hard modules are fixed. For each soft module, we might need to further place standard cells inside the module. The placement problem will be introduced in Chapter 11. Once the positions of all hard modules and standard cells are decided, we need to route all signal and power/ground nets, which will be introduced in Chapters 12 and 13, respectively.

10.6 EXERCISES

10.1. (Polish Expression and B*-tree) Given the following Polish expression, $E = 12V34HVH5$,

(a) Does the above expression have the balloting property? Justify your answer.

(b) Is E a normalized Polish expression? If not, exchange an operator and an operand to transform E into a normalized Polish expression E'.

(c) Give the slicing tree that corresponds to the Polish expression E. Also, give the slicing tree corresponding to the "resulting" normalized Polish expression E', if E is not a normalized Polish expression.
(d) Assume that modules 1, 2, 3, 4 and 5 have the sizes and shapes listed in Table 10.8. If all modules are rigid (hard) and rotation is not allowed, what will be the size of the smallest bounding rectangle corresponding to the "resulting" normalized Polish expression E'? Show all steps that lead to your answer.
(e) Give a B*-tree for the floorplan derived in (d).
(f) Show all steps for computing the coordinates of the modules from the resulting B*-tree of (e).

10.2. (Polish Expression and B*-tree) Given the following Polish expression, $E = 12V3H4V$,

(a) Give a slicing tree corresponding to the expression E.
(b) Assume modules 1, 2, 3, and 4 have the sizes and shapes indicated in Table 10.9. If all modules are rigid and no rotation is allowed, what will be the size of the smallest bounding rectangle corresponding to the Polish expression E? Show all steps that lead to your answer.
(c) Give a B*-tree for the floorplan derived in (b).
(d) Show all steps for computing the coordinates of the modules from the resulting B*-tree of (c).

Table 10.8. The Dimensions of Modules in Exercise 10.1

Module No.	Width	Height
1	2	3
2	2	2
3	5	3
4	3	3
5	1	3

Table 10.9. The Dimensions of Modules in Exercise 10.2

Module No.	Width	Height
1	2	2
2	3	2
3	2	4
4	5	3

10.3. (Sequence Pair and B*-tree) Consider the floorplan of five modules 1, 2, 3, 4, and 5 and their dimensions shown in Table 10.10 and Figure 10.40.

(a) Derive the sequence pair $S = (\Gamma_+, \Gamma_-)$ for the floorplan. Show your procedure.
(b) Show all steps on the sequence pair to evaluate the area cost for the $S = (\Gamma_+, \Gamma_-)$-packing. What is the area cost?
(c) Derive the B*-tree for the floorplan shown in the figure.
(d) Show all steps on the B*-tree for evaluating the cost efficiently. What is the area cost?

10.4. (Rectilinear Modules) In this chapter, we assume all modules are rectangular. However, in real-world application, some modules may be of a rectilinear shape. Show how to extend B*-tree and sequence pair to handle rectilinear modules. (*Hint: Dissect a rectilinear module into rectangular submodules.*)

10.5. (Shape Curve Candidates) Consider two lists, $A = \{(p_1, q_1), \ldots, (p_m, q_m)\}$ and $B = \{(x_1, y_1), \ldots, (x_n, y_n)\}$, with $p_i \leq p_{i+1}$, $q_i \geq q_{i+1}$, $x_i \leq x_{i+1}$, and $y_i \geq y_{i+1}$. Combine A and B by considering each element (p_i, q_i) of A and each element (x_j, y_j) of B to produce an element of a list C: $(p_i + x_j, \max\{q_i, y_j\})$. Thus C has $m \times n$ elements. If there are two elements (c_i, d_i) and (c_j, d_j) in C with $c_i \leq c_j$ and $d_i \leq d_j$, then delete

Table 10.10 The Dimensions of Modules in Exercise 10.3

Module No.	Width	Height
1	2	2
2	2	3
3	3	3
4	3	3
5	4	2

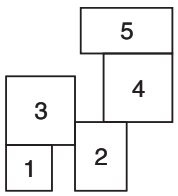

FIGURE 10.40

The floorplan for Exercise 10.3.

(c_j, d_j) from C. Prove that the resulting list C has at most a linear function of m and n elements. Find the linear function.

10.6. (Multi-level Framework) Give the strengths and weaknesses of the $\wedge$- and V-shaped multi-level frameworks. Here, the $\wedge$-shaped multi-level framework consists of two stages of bottom-up processing followed by top-down processing, whereas the V-shaped one uses top-down processing followed by bottom-up processing.

10.7. (Boundary Constraints on B*-tree) It is often useful to identify the modules being placed along a chip boundary because those modules are closet to the I/O pads in a traditional chip package with peripheral I/Os. Given a B*-tree, you are asked to derive the feasibility of the B*-tree for the boundary conditions.

(a) For the nodes corresponding to the modules along the bottom boundary of a floorplan, what are their positions in the B*-tree?

(b) For the nodes corresponding to the modules along the left boundary of a floorplan, what are their positions in the B*-tree?

(c) For the nodes corresponding to the modules along the right boundary of a floorplan, what are their positions in the B*-tree?

(d) For the nodes corresponding to the modules along the top boundary of a floorplan, what are their positions in the B*-tree?

10.8. (Programming) This programming assignment asks you to write a chip floorplanner that can handle hard macros and provide *graphic user interface* (GUI) to show the floorplanning result with interconnections (center-to-center connection for each net). The evaluation is based on the resulting floorplan area, wirelength, and running time.

(1) Input/Output specification

Input format

Each test case has two input files, *problem_no.mac* and *problem_no.net*. The first file defines chip and macro information includes chip width and chip height. The later includes name, area, and aspect ratio constraints of a macro and lists all nets. For example, there are two input files, problem1.mac and problem1.net. The first file format is as follows:

```
.chip_bbox (width, height)
//the lower-left corner of this bounding box is (0, 0)
.module name width height
.module name width height
. . . More modules
```

The format of the second file (netlist) is:

```
.net net_name module_name1 module_name2 ...
.net net_name module_name1 module_name2 ...
... More nets
// one line defines a net
// for example, if net N1 connects macro A, B, and C, the definition is
// .net N1 A B C
```

Output format

The output file consists of three parts: (1) bounding box for each macro (specified by the coordinates of the lower-left and upper-right corners), (2) total wirelength estimated by the ***half-perimeter wirelength*** (HPWL) of all nets, and (3) area (it may be smaller than the chip bounding box). The area can be obtained by $X * Y$, where X (Y) is the difference between rightmost (topmost) edge and leftmost (bottommost) edge among all modules. The report file format is as follows:

```
.module module_name (x1, y1) (x2, y2)
.module module_name (x1, y1) (x2, y2)
// (x1, y1): lower-left corner, (x2, y2): upper-right corner
... More modules
.wire total_wire_length
.area chip_area
// area = (max_x2 – min_x1) * (max_y2 – min_y1)
```

(2) Problem statement

Given (1) a set of rectangular modules and (2) a set of nets interconnecting these modules, the floorplanner places all modules within a specified fixed-outline (*i.e.,* a rectangular bounding box). We assume that the lower-left corner of this bounding box is the origin (0, 0) and no space (channel) is needed between two modules. The main objective is to minimize the total wirelength. The net terminals are assumed to be at the center of their corresponding module. The second objective is to minimize the chip area. Figure 10.41 illustrates an example of all input/output files.

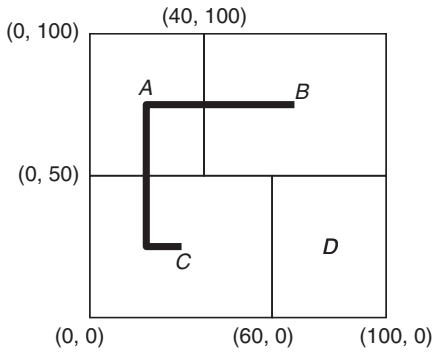

Figure 10.41. A floorplan problem and its solution, the bold line representing net N1.

Input files

[PROBLEM1.MAC]

```
.chip_bbox (100, 100)
.module A 50 40
.module B 60 50
.module C 60 50
.module D 50 40
```

[PROBLEM1.NET]

```
.net N1 A B C
```

Output file

[PROBLEM1.RPT]

```
.module A (0, 50) (40, 100)
.module B (40, 50) (100, 100)
.module C (0, 0) (60, 50)
.module D (60, 0) (100, 50)
.wire 100
.area 10000
```

ACKNOWLEDGMENTS

We thank Dr. Laung-Terng Wang of SynTest Technologies Inc., Professor Chris Chu of Iowa State University, Professor Cheng-Kok Koh of Purdue University, Professor Evangeline F.-Y. Young of the Chinese University of Hong Kong, Professor Martin D. F. Wong of the University of Illinois at Urbana-Champaign, and the National Taiwan University students in the physical design class for their very careful review of this chapter. We also thank SpringSoft Inc. for providing the programming assignment.

REFERENCES

R10.0 Books

[Lawler 1976] E. L. Lawler, *Combinatorial Optimization: Networks and Matroids,* Holt, Rinehart, and Winston, New York, 1976.

[Sait 1999] S. M Sait and H. Youssef, *VLSI Physical Design Automation: Theory and Practice,* World Scientific, Singapore, 1999.

[Sherwani 1999] N. Sherwani, *Algorithms for VLSI Physical Design Automation,* Third Edition, Kluwer Academic, Boston, 1999.

[Wong 1988] D. F. Wong, H. W. Leong, and C. L. Liu, *Simulated Annealing for VLSI Design,* Kluwer Academic, Boston, 1988.

R10.1 Introduction

[Kahng 2000] A. B. Kahng, Classical floorplanning harmful?, *in Proc. ACM Int. Symp. on Physical Design*, pp. 207–213, April 2000.

[Otten 1982] R. H. J. M. Otten, Automatic floorplan design, *in Proc. ACM/IEEE Design Automation Conf.*, pp. 261–267, June 1982.

R10.2 Simulated Annealing Approach

[Chang 2000] Y.-C. Chang, Y.-W. Chang, G.-M. Wu, and S.-W. Wu, "B*-trees: a new representation for non-slicing floorplans," *in Proc. ACM/IEEE Design Automation Conf.*, pp. 458–463, June 2000.

[Guo 1999] P.-N. Guo, C.-K. Cheng, and T. Yoshimura, "An O-tree representation of non-slicing floorplan and its applications," *in Proc. ACM/IEEE Design Automation Conf.*, pp. 268–273, June 1999.

[Hilton 1991] P. Hilton and J. Pederson, "Catalan numbers, their generalization, and their uses," *Math. Intelligencer*, 13(2), pp. 64–75, February 1991.

[Hong 2000] X. Hong, G. Huang, Y. Cai, J. Gu, S. Dong, C.-K. Cheng, and J. Gu, "Corner block list: An effective and efficient topological representation of non-slicing floorplan," *in Proc. IEEE/ACM Int. Conf. on Computer-Aided Design*, pp. 8–13, November 2000.

[Kirpatrick 1983] S. Kirpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by simulated annealing," *Science*, 220(4598), pp. 671–680, May 13, 1983.

[Lin 2001] J.-M. Lin and Y.-W. Chang, "TCG: A transitive closure graph based representation for general floorplans," *in Proc. ACM/IEEE Design Automation Conf.*, pp. 764–769, June 2001.

[Lin 2002] J.-M. Lin and Y.-W. Chang, "TCG-S: Orthogonal coupling of P*-admissible representations for general floorplans," *in Proc. ACM/IEEE Design Automation Conf.*, pp. 842–847, June 2002.

[Lin 2003] J.-M. Lin, Y.-W. Chang, and S.-P. Lin, "Corner sequence: A P-admissible floorplan representation with a worst case linear-time packing scheme," *IEEE Trans. on Very Large Scale Integration Systems*, 11(4), pp. 679–686, August 2003.

[Murata 1995] H. Murata, K. Fujiyoshi, S. Nakatake, and Y. Kajatani, "Rectangle packing based module placement," *in Proc. IEEE/ACM Int. Conf. on Computer-Aided Design*, pp. 472–479, November 1995.

[Otten 1982] R. H. J. M. Otten, "Automatic floorplan design," *in Proc. ACM/IEEE Design Automation Conf.*, pp. 261–267, June 1982.

[Otten 1983] R. H. J. M. Otten, "Efficient floorplan optimization," *in Proc. IEEE Int. Conf. on Computer Design*, pp. 499–502, November 1983.

[Sechen 1986] C. Sechen and A. Sangiovanni-Vincentelli, "TimberWolf3.2: A new standard cell placement and global routing package," *in Proc. IEEE/ACM Design Automation Conf.*, pp. 432–439, June 1986.

[Sechen 1988] C. Sechen, "Chip-planning, placement, and global routing of macro/custom cell integrated circuits using simulated annealing," *in Proc. IEEE/ACM Design Automation Conf.*, pp. 73–80, June 1988.

[Shen 2003] C. Shen and C. Chu, "Bounds on the number of slicing, mosaic and general floorplans," *IEEE Trans. on Computer-Aided Design*, 22(10), pp. 1354–1361, October 2003.

[Stockmeyer 1983] L. Stockmeyer, "Optimal orientations of cells in slicing floorplan designs", *Information and Control*, 57(2-3), pp. 91–101, May/June 1983.

[Tang 2001] X. Tang and D. F. Wong, "FAST-SP: A fast algorithm for block placement based on sequence pair," *in Proc. IEEE/ACM Asia South Pacific Design Automation Conf.*, pp. 521–526, January 2001.

[Wong 1986] D. F. Wong and C. L. Liu, "A new algorithm for floorplan design," *in Proc. ACM/IEEE Design Automation Conf.*, pp. 101–107, June 1986.

[Young 2003] E. F.-Y. Young, C. C.-N. Chu, and Z. C. Shen, "Twin binary sequences: A non-redundant representation for general non-slicing floorplan," *IEEE Trans. on Computer-Aided Design*, 22(4), pp. 457–469, April 2003.

[Zhou 2004] H. Zhou and J. Wang, "ACG-adjacent constraint graph for general floorplans" *in Proc. IEEE Int. Conf. on Computer Design*, pp. 572–575, October 2004.

R10.3 Analytical Approach

[GLPK 2008] *GLPK* (GNU Linear Programming Kit), http://www.gnu.org/software/glpk/, 2008.

[ILOG 2008] ILOG CPLEX, http://www.ilog.com/products/cplex/, 2008.

[LINDO 2008] LINDO System Inc., http://www.lindo.com/, 2008.

[lp_solve] lp_solve, http://tech.groups.yahoo.com/group/lp_solve/, 2008.

[Sutanthavibul 1991] S. Sutanthavibul, E. Shragowitz, and J. B. Rosen, "An analytical approach to floorplan design and optimization," *IEEE Trans. on Computer-Aided Design of Integrated Circuits an Systems*, 10(6), pp. 761–769, June 1991.

[Zhan 2006] Y. Zhan, Y. Feng, and S. S. Sapatnekar, "A fixed-die floorplanning algorithm using an analytical approach," *in Proc. IEEE/ACM Asia South Pacific Design Automation Conf.*, pp. 771–776, January 2006.

R10.4 Modern Floorplanning Considerations

[Adya 2003] S. N. Adya and I. Markov, "Fixed-outline floorplanning: enabling hierarchical design," *IEEE Trans. on Very Large Scale Integration Systems*, 11(6), pp. 1120–1135, December 2003.

[Alpert 1997] C. J. Alpert and A. Devgan, "Wire segmenting for improved buffer insertion," *in Proc. ACM/IEEE Design Automation Conf.*, pp. 588–593, June 1997.

[Chang 2000] Y.-C. Chang, Y.-W. Chang, G.-M. Wu, and S.-W. Wu, "B*-trees: A new representation for non-slicing floorplans," *in Proc. ACM/IEEE Design Automation Conf.*, pp. 458–463, June 2000.

[Chen 2005a] T.-C. Chen and Y.-W. Chang, "Modern floorplanning based on fast simulated annealing," *in Proc. ACM Int. Symp. on Physical Design*, pp. 104–112, April 2005.

[Chen 2005b] T.-C. Chen, Y.-W. Chang, and S.-C. Lin, "IMF: interconnect-driven multilevel floorplanning for large-scale building-module designs," *in Proc. IEEE/ACM Int. Conf. on Computer-Aided Design*, pp. 159–164, November 2005.

[Chen 2006] T.-C. Chen and Y.-W. Chang, "Modern floorplanning based on B*-tree and fast simulated annealing," *IEEE Trans. on Computer-Aided Design*, 25(4), pp. 637–650, April 2006.

[Chi 2003] J.-C. Chi and M. C. Chi, "A block placement algorithm for VLSI circuits," *Chung Yuan Journal*, 31(1), pp. 69–75, March 2003.

[Cho 2006] M. Cho, H. Shin, and D. Z. Pan, "Fast substrate noise-aware floorplanning with preference directed graph for mixed-signal SOCs," *in Proc. IEEE/ACM Asia South Pacific Design Automation Conf.*, pp. 765–770, January 2006.

[Cong 1997] J. Cong, L. He, K.-Y. Khoo, C.-K. Koh, and Z. Pan, "Interconnect design for deep submicron ICs," *in Proc. IEEE/ACM Int. Conf. on Computer-Aided Design*, pp. 478–485, November 1997.

[Cong 1999] J. Cong, T. Kong, and D. Z. Pan, "Buffer block planning for interconnect-driven floorplanning," *in Proc. IEEE/ACM Int. Conf. on Computer-Aided Design*, pp. 358–363, November 1999.

[Cong 2004] J. Cong, J. Wei, and Y. Zhang, "A thermal-driven floorplanning algorithm for 3D ICs," *in Proc. Int. Conf. on Computer-Aided Design*, pp. 306–313, November 2004.

[Cong 2005] J. Cong, M. Romesis, and J. R. Shinnerl, "Fast floorplanning by look-ahead enabled recursive bipartitioning," *in Proc. IEEE/ACM Asia South Pacific Design Automation Conf.*, pp. 1119–1122, January 2005.

[Jagannathan 2005] A. Jagannathan, H. H. Yang, K. Konigsfeld, D. Milliron, M. Mohan, Mi. Romesis, G. Reinman, and J. Cong, "Microarchitecture evaluation with floorplanning and interconnect pipelining," *in Proc. ACM/IEEE Asia South Pacific Design Automation Conf*, pp. 8–15, January 2005.

[Kawai 2005] H. Kawai and K. Fujiyoshi, "3D-block packing using a tree representation," *in Proc. Workshop on Circuits and Systems in Karuizawa*, pp. 199–204, April 2005.

[Law 2005] J. H. Y. Law and E. F. Y. Young, "Multi-bend bus driven floorplanning," *in Proc. ACM Int. Symp. Physical Design*, pp. 113–120, April 2005.

[Lee 2003] H.-C. Lee, Y.-W. Chang, J.-M. Hsu, and H. H. Yang, "Multilevel floorplanning/placement for large-scale modules using B*-trees," *in Proc. ACM/IEEE Design Automation Conf.*, pp. 812–817, June 2003.

[Lin 2001] S. Lin and N. Chang, "Challenges in power-ground integrity," *in Proc. IEEE/ACM Int. Conf. on Computer-Aided Design*, pp. 651–654, November 2001.

[Lin 2004] C.-T. Lin, D.-S. Chen, and Y.-W. Wang, "Robust fixed-outline floorplanning through evolutionary search," *in Proc. IEEE/ACM Asia and South Pacific Design Automation Conf.*, pp. 42–44, January 2004.

[Liu 2007] C.-W. Liu and Y.-W. Chang, "Power/ground network and floorplan co-synthesis for fast design convergence," *IEEE Trans. on Computer-Aided Design*, 26(4), pp. 693–704, April 2007.

[Ma 2007] Y. Ma, Z. Li, J. Cong, X. Hong, G. Reinman, S. Dong, and Q. Zhou, Micro-architecture pipelining optimization with throughput-aware floorplanning, *in Proc. ACM/IEEE Asia South Pacific Design Automation Conf.*, pp. 920–925, January 2007.

[Nakatake 1996] S. Nakatake, K. Fujiyoshi, H. Murata, and Y. Kajatani, "Module placement on BSG-structure and IC layout applications," *in Proc. IEEE/ACM Int. Conf. on Computer-Aided Design*, pp. 261–267, November 1996.

[Wu 2004] S.-W. Wu and Y.-W. Chang, "Efficient power/ground network analysis for power integrity-driven design methodology," *in Proc. ACM/IEEE Design Automation Conf.*, pp. 177–180, June 2004.

[Wu 2005] J.-Y. Wu, T.-C. Chen, and Y.-W. Chang, "SoC test scheduling using the B*-tree based floorplanning technique," *in Proc. ACM/IEEE Asia South Pacific Design Automation Conf.*, pp. 1188–1191, January 2005.

[Xiang 2003] H. Xiang, X. Tang, and M. D. F. Wong, "Bus-driven floorplanning," *in Proc. IEEE/ACM Int. Conf. Computer-Aided Design*, pp. 66–73, November 2003.

[Yan 2008] J. Z. Yan and C. Chu, "DeFer: Deferred decision making enabled fixed-outline floorplanner," *in Proc. IEEE/ACM Design Automation Conf.,* June 2008.

[Yamazaki 2003] H. Yamazaki, K. Sakanushi, S. Nakatake, and Y. Kajitani, "The 3D-packing by meta data structure and packing heuristics," *IEICE Trans. on Fundamentals of Electronics, Communications and Computer*, E82-A(4), pp. 639–645, April 2003.

[Yuh 2004a] P.-H. Yuh, C.-L. Yang, and Y.-W. Chang, "Temporal floorplanning using the T-tree representation," *in Proc. IEEE/ACM Int. Conf. on Computer-Aided Design*, pp. 300–305, November 2004.

[Yuh 2004b] P.-H. Yuh, C.-L. Yang, Y.-W. Chang, and H.-L. Chen, "Temporal floorplanning using 3D-subTCG," *in Proc. IEEE Asia and South Pacific Conf. on Circuits and Systems*, pp. 725–730, January 2004.

[Yuh 2006] P-H. Yuh, C.-L. Yang, and Y.-W. Chang, "Placement of digital microfluidic biochips using the T-tree formulation," *in Proc of ACM/IEEE Design Automation Conf.* pp. 931–934, July 2006.

[Yim 1999] J.-S. Yim, S.-O. Bae, and C.-M. Kyung, "A floorplan-based planning methodology for power and clock distribution in ASICs," *in Proc. ACM/IEEE Design Automation Conf.*, pp. 766–771, June 1999.

[Zhou 2004] H. Zhou and J. Wang, "ACG–adjacent constraint graph for general floorplans," *in Proc. IEEE Int. Conf. on Computer Design*, pp. 572–575, October 2004.

R10.5 Concluding Remarks

[Zhan 2006] Y. Zhan, Feng, and S. S. Sapatnekar, "A fixed-die floorplanning algorithm using an analytical approach," *in Proc. IEEE/ACM Asia South Pacific Design Automation Conf.*, pp. 771–776, January 2006.