# 8 VLSI Layout Algorithms*

8.1 Background ..................... 8-2
8.2 Placement Techniques ..................... 8-4
8.3 Compaction and the Single-Source Shortest Path Problem ..................... 8-6
8.4 Floorplan Sizing and Classic Divide and Conquer ..... 8-10
8.5 Routing Problems ..................... 8-13
8.6 Global Routing ..................... 8-15
8.7 Channel Routing ..................... 8-16
Manhattan Routing • Single-Layer Routing
8.8 Research Issues and Summary ..................... 8-22
8.9 Further Information ..................... 8-22
Defining Terms ..................... 8-23
Acknowledgment ..................... 8-24
References ..................... 8-25

Andrea S. LaPaugh
*Princeton University*

One of the many application areas that has made effective use of algorithm design and analysis is computer-aided design (CAD) of digital circuits. Many aspects of circuit design yield to combinatorial models and the algorithmic techniques discussed in other chapters. In this chapter we focus on one area within the field of CAD: layout of very large scale integrated (VLSI) circuits, which is a particularly good example of the effective use of algorithm design and analysis. We will discuss specific problems in VLSI layout and how algorithmic techniques have been successfully applied. This chapter will not provide a broad survey of CAD techniques for layout, but will highlight a few problems that are important and have particularly nice algorithmic solutions. The reader may find more complete discussions of the field in the references discussed in Section 8.9 at the end of this chapter.

Integrated circuits are made by arranging active elements (usually transistors) on a planar substrate and interconnecting these elements with conducting wires that are also patterned on the planar substrate [41]. There may be several layers that can be used for wires, but there are restrictions on how wires in different layers can connect, as well as requirements of separations between wires and elements. Therefore, the layout of integrated circuits is usually modeled as a planar-embedding problem with several layers in the plane.

VLSI circuits contain millions of transistors. Therefore, it is not feasible to consider the positioning of each transistor separately. Transistors are organized into subcircuits called components; this may be done hierarchically, resulting in several levels of component definition between the individual

* Supported in part by the National Science Foundation, FAW award MIP-9023542.

transistors and the complete VLSI circuit. The layout problem for VLSI circuits becomes one of positioning components and their interconnecting wires on a plane, following the design rules, and optimizing some measure such as area or wire length. Within this basic problem structure are a multitude of variations rising from changes in design rules and flexibilities within components as to size, shape, and regions where wires may connect to components. Graph models are used heavily, both to model the components and interconnections themselves and to capture constraint between objects. Geometric aspects of the layout problem must also be modeled. Most layout problems are optimization problems and most are NP-complete. Therefore, heuristics are also employed heavily. In the following text, we present several of the best known and best understood problems in VLSI layout.

## 8.1 Background

We will consider a design style known as "general cell." In **general cell layout**, components vary in size and degree of functional complexity. Some components may be from a predesigned component library and have rigidly defined layouts (e.g., a register bank) and others may be full custom designs, in which the building blocks are individual transistors and wires.* Components may be defined hierarchically, so that the degree of flexibility in the layout of each component is quite variable. Other design styles, such as standard cell and gate array, are more constrained but share many of the same layout techniques.

The layout problem for a VLSI chip is often decomposed into two stages: placement of components and routing of wires. For this decomposition, the circuit is described as a set of components and a set of interconnections among those components. The components are first placed on the plane based on their size, shape, and interconnectivity. Paths for wires are then found to interconnect specified positions on the components. Thus, a placement problem is to position a set of components in a planar region; either the region is bounded, or a measure such as the total area of the region used is optimized. The area needed for the yet undetermined routing must be taken into account. A routing problem is, given a collection of sets of points in the plane, to interconnect each set of points (called a **net**) using paths from an allowable set of paths. The allowable set of paths captures all the constraints on wire routes. In routing problems, the width of wires is abstracted away by representing only the midline of each wire and ensuring enough room for the actual wires through the definition of the set of allowable paths.

This description of the decomposition of a layout problem into placement and routing is meant to be very general. To discuss specific problems and algorithms, we will use a more constrained model. In our model, components will be rectangles. Each component will contain a set of points along its boundary, the **terminals**. Sets of these terminals are the nets, which must be interconnected. A layout consists of a placement of the components in the plane and a set of paths in the plane that do not intersect the components except at terminals and interconnect the terminals as specified by the nets. The paths are composed of segments in various layers of the plane. Further constraints on the set of allowable paths define the routing style, and will be discussed for each style individually. The area of a layout will be the area of the minimum-area rectangle that contains the components and wire paths (see Figure 8.1).

While we still have a fairly general model, we have now restricted our component shapes to be rectangular, our terminals to be single points on component boundaries, and our routing paths to avoid components. (Components are thus assumed to be densely populated with circuitry.) While

* We will not discuss special algorithms for laying out transistors, used in tools called "cell generators" or "leaf-cell generators" for building components from low-level layout primitives. The reader is referred to [22] as a starting point for an investigation of this topic.

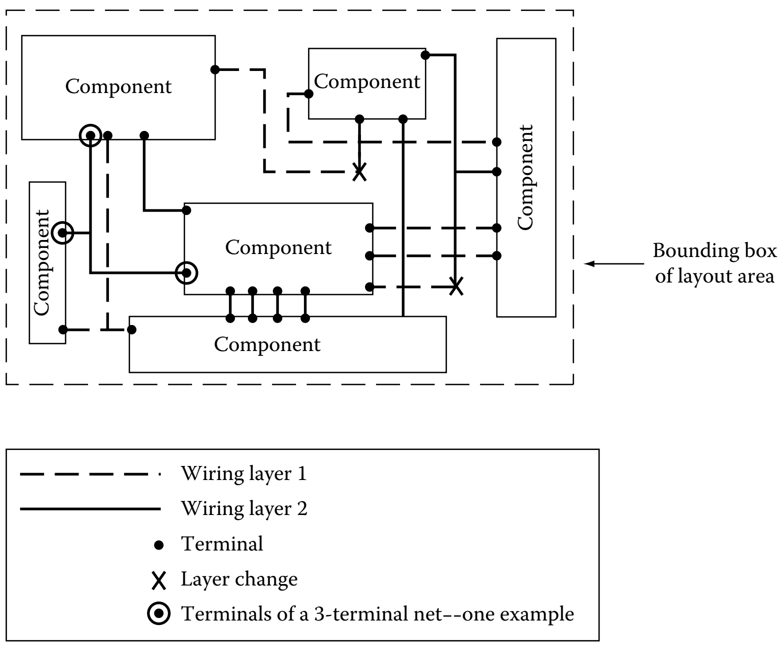

**FIGURE 8.1** Example of a layout. This layout is rectilinear and has two layers for wiring.

these assumptions are common and allow us to illustrate important algorithmic results, there is quite a bit of work on nonrectangular components (e.g., [16,21]), more flexible terminals (see [37]), and "over-the-cell" routing (see [37]). Often, layouts are further constrained to be **rectilinear**. In rectilinear layouts, there is an underlying pair of orthogonal axes defining "horizontal" and "vertical" and the sides of the components are oriented parallel to these axes. The paths of wires are composed of horizontal and vertical segments. In our ensuing discussion, we too will often assume rectilinear layouts.

If a VLSI system is too large to fit on one chip, then it is first partitioned into chip-sized pieces. During partitioning, the goal is to create the fewest chips with the fewest connections between chips. Estimates are used for the amount of space needed by wires to interconnect the components on one chip. The underlying graph problem for this task is **graph partitioning**, which is discussed in the following text.

Closely related to the placement problem is the **floorplanning** problem. Floorplanning occurs before the designs of components in a general cell design have been completed. The resulting approximate layout is called a **floorplan**. Estimates are used for the size of each component, based on either the functionality of the component or a hierarchical decomposition of the component. Rough positions are determined for the components. These positions can influence the shape and terminal placement within each component as its layout is refined. For hierarchically defined components, one can work bottom up to get rough estimates of size, then top down to get rough estimates of position, then bottom up again to refine positions and sizes.

Once a layout is obtained for a VLSI circuit, either through the use of tools or by hand with a layout editor, there may still be room for improvement. **Compaction** refers to the process of modifying a given layout to remove extra spaces between features of the layout, spaces not required by design rules. Humans may introduce such space by virtue of the complexity of the layout task. Tools may place artificial restrictions on layout in order to have tractable models for the main problems of placement and routing. Compaction becomes a postprocessing step to make improvements too difficult to carry out during placement and routing.

## 8.2 Placement Techniques

Placement algorithms can be divided into two types: constructive initial placement algorithms and iterative improvement algorithms. A constructive initial placement algorithm has as input a set of components and a set of nets. The algorithm constructs a legal placement with the goal of optimizing some cost function for the layout. Common cost functions measure component area, estimated routing area, estimated total wire length, estimated maximum wire length of a net, or a combination of these. An iterative improvement algorithm has as input the set of components, set of nets, and an initial placement; it modifies the placement, usually repeatedly, to improve a cost function. The initial placement may be a random placement or may be the output of a constructive initial placement algorithm.

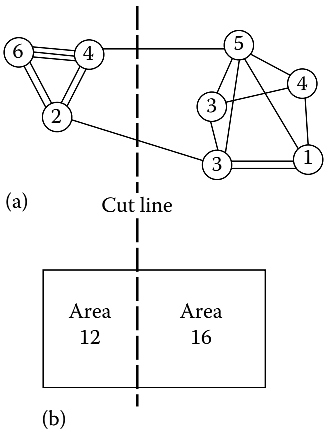

**FIGURE 8.2** Partitioning used in placement construction. (a) Partitioning the circuit: Each vertex represents a component; the area of the component is the number inside the vertex. Connections between components are represented by edges. Multiple edges between vertices indicate multiple nets connecting components. (b) Partitioning the layout rectangle proportionally to the partition of component area.

The iterative improvement of placements can proceed in a variety of ways. A set of allowable moves, that is, ways in which a placement can be modified, must be identified. These moves should be simple to carry out, including reevaluating the cost function. Typically, a component is rotated or a pair of components are exchanged. An iterative improvement algorithm repeatedly modifies the placement, evaluates the new placement, and decides whether the move should be kept as a starting point for further moves or as a tentative final placement. In the simplest of iterative improvement algorithms, a move is kept only if the placement cost function is improved by it. One more sophisticated paradigm for iterative improvement is simulated annealing (see Chapter 33 of *Algorithms and Theory of Computation Handbook, Second Edition: General Concepts and Techniques*). It has been applied successfully to the placement problem, for example, [36], and the floorplanning problem, for example, [4].

For general cell placement, one of the most widely used initial placement algorithms is a recursive partitioning method (see [24, p. 333]). In this method, a rectangular area for the layout is estimated based on component sizes and connectivity. The algorithm partitions the set of components into two roughly equal-sized subsets such that the number of interconnections between the two subsets is minimized and simultaneously partitions the layout area into two subrectangles of sizes equal to the sizes of the subsets (see Figure 8.2). This partitioning proceeds recursively on the two subsets and subrectangles until each component is in its own subset and the rectangle contains a region for each component. The recursive partitioning method is also heavily used in floorplanning [16].

The fundamental problem underlying placement by partitioning is **graph partitioning**.* Given a graph $G = (V, E)$, a vertex weight function $w : V \to N$, an edge cost function $c : E \to N$, and a balance factor $\beta \in [1/2, 1]$, the graph partitioning problem is to partition $V$ into two subsets $V_1$ and $V_2$ such that

$$\sum_{v \in V_1} w(v) \leq \beta \sum_{v \in V} w(v) \qquad (8.1)$$

$$\sum_{v \in V_2} w(v) \leq \beta \sum_{v \in V} w(v) \qquad (8.2)$$

* Our definitions follow those in [24].

and the cost of the partition,

$$\sum_{e \in E \cap (V_1 \times V_2)} c(e) \tag{8.3}$$

is minimized. This problem is NP-complete (see [10, pp. 209,210]). Graph partitioning is a well-studied problem. The version we have defined is actually bipartitioning. Heuristics for this problem form the core of heuristic algorithms for more general versions of the partition problem where one partitions into more than two vertex sets. The hypergraph version of partitioning, in which each edge is a set of two or more vertices rather than simply a pair of vertices, is a more accurate version for placement, since nets may contain many terminals on many components. But heuristics for the hypergraph version again are based on techniques used for the graph bipartitioning problem.

Among many techniques for graph partitioning, two—the Kernighan–Lin algorithm [17], and simulated annealing—are best known. Both are techniques for iteratively improving a partition. The Kernighan–Lin approach involves exchanging pairs of vertices across the partition. It was improved in the context of layout problems by Fiduccia and Mattheyses [9], who moved a single vertex at a time. As applied to graph partitioning, simulated annealing also considers the exchange of vertices across the partition or the movement of a vertex from one side of the partition to the other. The methods of deciding which partitions are altered, which moves are tried, and when to stop the iteration process differentiate the two techniques.

Alternatives to iterative improvement have also received attention for circuit applications, especially as performance-related criteria have been added to the partitioning problem. Examples of these alternatives are spectral methods, based on eigenvectors of the Laplacian, and the use of network flow. The reader is referred to [1] or [32] for a more complete discussion of partitioning heuristics.

A second group of algorithms for constructing placements is based on agglomerative clustering. In this approach, components are selected one at a time and placed in the layout area according to their connectivity to components previously placed. Components are clustered so that highly connected sets of components are close to each other.

When the cost function for a layout involves estimating wire length, several methods can be used. The goal is to define a measure for each net that estimates the length of that net after it is routed. These estimated lengths can then be summed over all nets to get the estimate on the total wire length, or the maximum can be taken over all nets to get maximum net length. Two estimates are commonly used: (1) the half-perimeter of the smallest rectangle containing all terminals of a net and (2) the minimum Euclidean spanning tree of the net. Given a placement, the Euclidean spanning tree of a net is the spanning tree of a graph whose vertices are the terminals of the net, whose edges are all edges between vertices, and whose edge costs are the Euclidean distances in the given placement between the pair of terminals that are endpoints of the edges. Another often-used estimate is the minimum rectilinear spanning tree. This is because rectilinear layouts are common. For a rectilinear layout, the length of the shortest path between a pair of points, $(x_a, y_a)$ and $(x_b, y_b)$, is $|x_a - x_b| + |y_a - y_b|$ (assuming no obstacles). This distance, rather than the Euclidean distance, is used as the distance between terminals. (This is also called the $L_1$ or Manhattan metric.) A more accurate estimate of the wire length of a net would be the minimum Euclidean (or rectilinear) **Steiner tree** for the net. A Steiner tree for a set of points in the plane is a spanning tree for a superset of the points, that is, additional points may be introduced to decrease the length of the tree. Finding minimum Steiner trees is NP-hard (see [10]), while finding minimum spanning trees can be done in $O(|E| + |V| \log |V|)$ time for general graphs* and in $O(|V| \log |V|)$ time for Euclidean or rectilinear spanning trees (see Chapter 7 of *Algorithms and Theory of Computation Handbook, Second Edition:*

* Actually, using more sophisticated techniques, finding minimum spanning trees can be done in time almost linear in $|E|$.

*General Concepts and Techniques* and Chapter 1 of this book). The cost of a minimum spanning tree is an upper bound on the cost of a minimum Steiner tree. For rectilinear Steiner trees, the half-perimeter measure and two-thirds the cost of a minimum rectilinear spanning tree are lower bounds on the cost of a Steiner tree [14].

The minimum spanning tree is also useful for estimating routing area. The minimum spanning tree for each net is used as an approximation of the set of paths for the wires of the net. Congested areas of the layout can then be identified and space for routing allocated accordingly.

## 8.3 Compaction and the Single-Source Shortest Path Problem

Compaction can be done at various levels of design: an entire layout can be compacted at the level of transistors and wires; the layouts of individual components can be compacted; a layout of components can be compacted without changing the layout within components. To simplify our discussion, we will assume that layouts are rectilinear. For compaction, we model a layout as composed entirely of rectangles. These rectangles may represent the most basic geometric building blocks of the circuit: pieces of transistors and segments of wires, or may represent more complex objects such as complex components. We refer to these rectangles as the features of the layout. We distinguish two types of features: those that are of fixed size and shape and those that can be stretched or shrunk in one or both dimensions. For example, a wire segment may be able to stretch or shrink in one dimension, representing a lengthening or shortening of the wire, but be fixed in the other dimension, representing a wire of fixed width. We refer to the horizontal dimension of a feature as its width and the vertical dimension as its height.

Compaction is fundamentally a two-dimensional problem. However, two-dimensional compaction is very difficult. Algorithms based on branch and bound techniques for integer linear programming (see Chapter 31 of *Algorithms and Theory of Computation Handbook, Second Edition: General Concepts and Techniques*) have been developed, but none is efficient enough to use in practice (see [24], Chapter 6 of [34]). Therefore, the problem is commonly simplified by compacting in each dimension separately: first all features of a layout are pushed together horizontally as much as the design rules will allow, keeping their vertical positions fixed; then all features of a layout are pushed together vertically as much as the design rules will allow, keeping their horizontal positions fixed. The vertical compaction may in fact make possible more horizontal compaction, and so the process may be iterated. This method is not guaranteed to find a minimum area compaction, but, for each dimension, the compaction problem can be solved optimally. We are assuming that we start with a legal layout and that one-dimensional compaction cannot change order relationships in the compaction direction. That is, if two features are intersected by the same horizontal (vertical) line and one feature is to the left of (above) the other, then horizontal (vertical) compaction will maintain this property. The algorithm we present is based on the single-source shortest path algorithm (see Chapter 7 of *Algorithms and Theory of Computation Handbook, Second Edition: General Concepts and Techniques*). It is an excellent example of a widely used application of this graph algorithm.

The compaction approach we are presenting is called “constraint-based” compaction because it models constraints on and between features explicitly. We shall discuss the algorithm in terms of horizontal compaction, vertical compaction being analogous. We use a graph model in which vertices represent the horizontal positions of features; edges represent constraints between the positions of features. Constraints capture the layout design rules, relationships between features such as connectivity, and possibly other desirable constraints such as performance-related constraints. Design rules are of two types: feature-size rules and separation rules. Feature-size rules give exact sizes or minimum dimensions of features. For example, each type of wire has a minimum width; each transistor in a layout is of a fixed size. Separation rules require that certain features of a layout be at least a minimum distance apart to avoid electrical interaction or problems during fabrication. Connectivity constraints occur when a wire segment is allowed to connect to a component (or another wire segment) anywhere in a given interval along the component boundary. Performance requirements may dictate that certain elements are not too far apart. A detailed discussion of the issues in the extraction of constraints from a layout can be found in Chapter 6 of [34].

In the simplest case, we start with a legal layout and only consider feature-size rules and separation rules. We assume all wire segments connect at fixed positions on component boundaries and there are no performance constraints. Furthermore, we assume all features that have a variable width attach at their left and right edges to features with fixed width, for example, a wire segment stretched between two components. Then, we need only represent features with fixed width; we can use one variable for each feature. In this case, any constraints on the width of a variable-width feature are translated into constraints on the positions of the fixed-width features attached to either end (see Figure 8.3). We are left with only separation constraints, which are of the form

$$x_B \geq x_A + d_{\min} \tag{8.4}$$

or equivalently

$$x_B - x_A \geq d_{\min} \tag{8.5}$$

where $B$ is a feature to the right of $A$, $x_A$ is the horizontal position of $A$, $x_B$ is the horizontal position of $B$, and the minimum separation between $x_A$ and $x_B$ is $d_{\min}$. In our graph model, there is an edge from the vertex for feature $A$ to the vertex for feature $B$ with length $d_{\min}$. We add a single extra source vertex to the graph and a 0-length edge from this source vertex to every other vertex in the graph. This source vertex represents the left edge of the layout. Then finding the longest path from this source vertex to every vertex in the graph will give the leftmost legal position of each feature—as if we had pushed each feature as far to the left as possible. Finding the longest path is converted to a single-source shortest path problem by negating all the lengths on edges. This is equivalent to rewriting the constraint as

$$x_A - x_B \leq -d_{\min}. \tag{8.6}$$

From now on, we will write constraints in this form. Note that this graph is acyclic. Therefore, as explained in the following text, the single-source shortest path problem can be solved in time $O(n + |E|)$ by a **topological sort**, where $n$ is the number of features and $E$ is the set of edges in the constraint graph.

A topological sorting algorithm is an algorithm for visiting the vertices of a directed acyclic graph (DAG). The edges of a DAG induce a partial order on the vertices: $v < u$ if there is a (directed) path

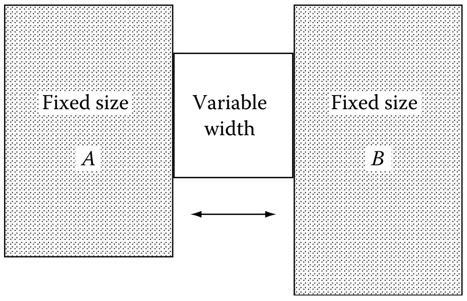

**FIGURE 8.3** Generating separation constraints. The constraint on the separation between features $A$ and $B$ is the larger of the minimum separation between them and the minimum width of the flexible feature connecting them.

from $v$ to $u$ in the graph. A topological order is any total ordering of the vertices that is consistent with this partial order. A topological sorting algorithm visits the vertices in some topological order. In Chapter 7 of *Algorithms and Theory of Computation Handbook, Second Edition: General Concepts and Techniques*, a topological sorting algorithm based on depth-first search is presented. For our purposes, a modified breadth-first search approach is more convenient. In this modification, we visit a node as soon as all its immediate predecessors have been visited (rather than as soon as any single immediate predecessor has been visited). For a graph $G = (V, E)$ this algorithm can be expressed as follows:

```
TOPOLOGICAL SORT ($G$)

1  S $\leftarrow$ all vertices with no incoming edges (sources)
2  $U \leftarrow V$
3  **while** $S$ is not empty
4     **do** choose any vertex $v$ from $S$
5        VISIT $v$
6        **for** each vertex $u$ such that $(v, u) \in E$
7           **do** $E \leftarrow E - \{(v, u)\}$
8              **if** $u$ is now a source
9                 **then** $S \leftarrow S \cup \{u\}$
10       $U \leftarrow U - \{v\}$
11       $S \leftarrow S - \{v\}$
12 **if** $U$ is not empty
13    **then** error $\triangleright$ G is not acyclic
```

In our single-source shortest path problem, we start with only one source, $s$, the vertex representing the left edge of the layout. We compute the length of the shortest path from $s$ to each vertex $v$, denoted as $\ell(v)$. We initialize $\ell(v)$ before line 3 to be 0 for $\ell(s)$ and $\infty$ for all other vertices. Then for each vertex $v$ we select at line 4, and each edge $(v, u)$ we delete at line 7 we update for all shortest paths that go through $v$ by $\ell(u) \leftarrow \min\{\ell(u), \ell(v) + \text{length}(v, u)\}$. When the topological sort has completed, $\ell(v)$ will contain the length of the shortest path from $s$ to $v$ (unless $G$ was not acyclic to begin with). The algorithm takes $O(|V| + |E|)$ time.

In our simplest case, all our constraints were minimum separation constraints. In the general case, we may have maximum separation constraints as well. These occur when connectivity constraints are used and also when performance constraints that limit the length of wires are used. Then we have constraints of the form

$$x_B \leq x_A + d_{\max} \tag{8.7}$$

or equivalently

$$x_B - x_A \leq d_{\max}. \tag{8.8}$$

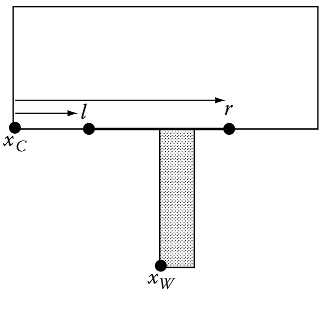

**FIGURE 8.4** Modeling a connection that can be made along an interval of a component boundary.

Such a constraint is modeled as an edge from the vertex for feature $B$ to the vertex for feature $A$ with weight $d_{\max}$. For example, to model a horizontal wire $W$ that has an interval from $l$ to $r$ along which it can connect to a component $C$ (see Figure 8.4), we use the pair of constraints

$$x_C - x_W \leq -l \tag{8.9}$$

and

$$x_W - x_C \leq r. \tag{8.10}$$

Once we allow both minimum and maximum separation constraints, we have a much more general linear constraint system. All constraints are still of the form

$$x - y \leq d, \tag{8.11}$$

but the resulting constraint graph need not be acyclic. (Note that equality constraints $y - x = d$ may be expressed as $y - x \leq d$ and $x - y \leq -d$. Equality constraints may also be handled in a preprocessing step that merges vertices that are related by equality constraints.) To solve the single-source shortest path algorithm, we now need the $O(|V||E|)$-time Bellman–Ford algorithm (see [6]). This algorithm only works if the graph contains no negative cycle. If the constraint graph is derived from a layout that satisfies all the constraints, this will be true, since a negative cycle represents an infeasible set of constraints. However, if the initial layout does not satisfy all the constraints, for example, the designer adds constraints for performance to a rough layout, then the graph may be cyclic. The Bellman–Ford algorithm can detect this condition, but since the set of constraints is infeasible, no layout can be produced.

If the constraint graph is derived from a layout that satisfies all the constraints, an observation by Maley [26] allows us to use Dijkstra's algorithm to compute the shortest paths more efficiently. To use Dijkstra's algorithm, the weights on all the edges must be positive (see Chapter 7 of *Algorithms and Theory of Computation Handbook, Second Edition: General Concepts and Techniques*). Maley observed that when an initial layout exists, the initial positions of the features can be used to convert all lengths to positive lengths as follows. Let $p_A$ and $p_B$ be initial positions of features $A$ and $B$. The constraint graph is modified so that the length of an edge $(v_A, v_B)$ from the vertex for $A$ to the vertex for $B$ becomes length$(v_A, v_B) + p_B - p_A$. Since the initial layout satisfies the constraint $x_A - x_B \leq \text{length}(v_A, v_B)$, we have $p_B - p_A \geq -\text{length}(v_A, v_B)$ and $p_B - p_A + \text{length}(v_A, v_B) \geq 0$. Maley shows that this transformation of the edge lengths preserves the shortest paths. Since all edge weights have been converted to positive lengths, Dijkstra's algorithm can be used, giving a running time of $O(|V| \log |V| + |E|)$ or $O(|E| \log |V|)$, depending on the implementation used.*

Even when the constraint graph is not acyclic and an initial feasible layout is not available, restrictions on the type or structure of constraints can be used to get faster algorithms. For example, Lengauer and Mehlhorn give an $O(|V| + |E|)$-time algorithm when the constraint graph has a special structure called a "chain DAG" that is found when the only constraints other than minimum separation constraints are those coming from flexible connections such as those modeled by Equations 8.9 and 8.10 (see [24, p. 614]). Liao and Wong [25] and Mata [28]† present $O(|E_x| \times |E|)$-time algorithms, where $E_x$ is the set of edges derived from constraints other than the minimum-separation constraints. These algorithms are based on the observation that $E - E_x$ is a DAG (as in our simple case earlier). Topological sort is used as a subroutine to solve the single-source shortest path problem with edges $E - E_x$. The solution to this problem may violate constraints represented by $E_x$. Therefore, after finding the shortest paths for $E - E_x$, positions are modified in an attempt to satisfy the other constraints (represented by $E_x$), and the single-source shortest path algorithm for $E - E_x$ is run again. This technique is iterated until it converges to a solution for the entire set of constraints or the set of constraints is shown to be infeasible, which is proven to be within $|E_x|$ iterations. If $|E_x|$ is small, this algorithm is more efficient than using Bellman–Ford.

The single-dimensional compaction that we have discussed ignores many practical issues. One major issue is the introduction of bends in wires. The fact that a straight wire segment connects two components may be an artifact of the layout, but it puts the components in lock-step during compaction. Adding a bend to the wire would allow the components to move independently, stretching the bend accordingly. Although the bend may require extra area, the overall area might improve through compaction with the components moving separately.

* The $O(|V| \log |V| + |E|)$ running time depends on using Fibonacci heaps for a priority queue. If the simpler binary heap is used, the running time is $O(|E| \log |V|)$. This comment also holds for finding minimum spanning trees using Prim's algorithm. See Chapter 7 of *Algorithms and Theory of Computation Handbook, Second Edition: General Concepts and Techniques* for a discussion on the running times of Dijkstra's algorithm and Prim's algorithm.

† The technique used by Mata is the same as that by Liao and Wong, but Mata has a different stopping condition for his search, which can lead to more efficient execution.

Another issue is allowing components to change their order from left to right or top to bottom. This change might allow for a smaller layout, but the compaction problem becomes much more difficult. In fact, a definition of one-dimensional compaction that allows for such exchanges is NP-complete (see [24, p. 587]). Practically speaking, such exchanges may cause problems for wire routing. The compaction problem we have presented requires that the topological relationships between the layout features remain unchanged while space is compressed.

## 8.4 Floorplan Sizing and Classic Divide and Conquer

The problem we will now consider, called **floorplan sizing**, is one encountered during certain styles of placement or floorplanning. With some reasonable assumptions about the form of the layout, the problem can be solved optimally by a polynomial-time algorithm that is an example of classic divide and conquer.

Floorplan sizing occurs when a floorplan is initially specified as a partitioning of a rectangular layout area, representing the chip, into subrectangles, representing components (see Figure 8.5a).

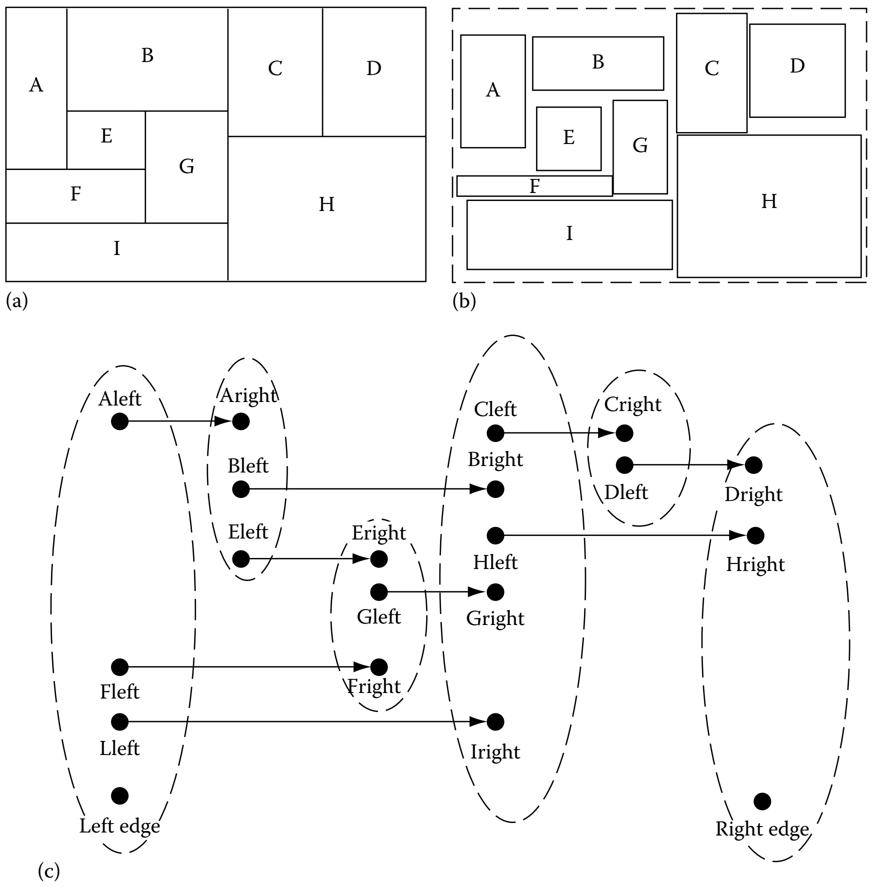

FIGURE 8.5 A floorplan and the derived layout. (a) A partition of a rectangle representing a floorplan. (b) A layout with actual components corresponding to the floorplan. (c) The horizontal constraint graph for the layout. Bidirectional 0-length edges are not shown. Instead, vertices constrained to have the same horizontal position due to chains of 0-length edges are shown in the same oval. They are treated as a single vertex.

Each subrectangle corresponds to some component. We assume that the rectangular partitioning is rectilinear. For this discussion, given a set of components $C$, by "a floorplan for $C$," we shall mean such a rectangular partition of a rectangle into $|C|$ subrectangles, and a one-to-one mapping from $C$ to the subrectangles. This partition indicates the relative position of components, but the subrectangles are constrained only to have approximately the same area as the components (possibly with some bloating to account for routing), not to have the same aspect ratio as the components. When the actual components are placed in the locations, the layout will change (see Figure 8.5b). Furthermore, it may be possible to orient each component so that the longer dimension may be either horizontal or vertical, affecting the ultimate size of the layout. In fact, if the component layouts have not been completed, the components may be able to assume many shapes while satisfying an area bound that is represented by the corresponding subrectangle. We will formalize this through the use of a **shape function**.

*Definition:* A shape function for a component is a mapping $s : [w_{\min}, \infty] \rightarrow [h_{\min}, \infty]$ such that $s$ is monotonically decreasing, where $[w_{\min}, \infty]$ and $[h_{\min}, \infty]$ are intervals of $\Re^{+}$.

The interpretation of $s(w)$ is that it is the minimum height (vertical dimension) of any rectangle of width $w$ that contains a layout of the component. $w_{\min}$ is the minimum width of any rectangle that contains a layout and $h_{\min}$ is the minimum height. The monotonicity requirement represents the fact that if there is a layout for a component that fits in a $w \times s(w)$ rectangle, it certainly must fit in a $(w + d) \times s(w)$ rectangle for any $d \geq 0$; therefore, $s(w + d) \leq s(w)$. In this discussion we will restrict ourselves to piecewise linear shape functions.

Given an actual shape (width and height) for each component, determining the minimum width and minimum height of the rectangular layout area becomes a simple compaction problem as discussed earlier. Each dimension is done separately, and two constraint graphs are built. We will discuss the horizontal constraint graph; the vertical constraint graph is analogous. The reader should refer to Figure 8.5c. The horizontal constraint graph has a vertex for each vertical side of the layout rectangle and each vertical side of a subrectangle (representing a component) in the rectangular partition. There is a directed edge from each vertex representing the left side of a subrectangle to each vertex representing the right side of a subrectangle; this edge has a length that is the width of the corresponding component. There are also two directed edges (one in each direction) between the vertices representing any two overlapping sides of the layout rectangle or subrectangles; the length of these edges is 0. Thus, the vertex representing the left side of the layout rectangle has 0-length edges between it and the vertices representing the left sides of the leftmost subrectangles. Note that these constraints force two subrectangles that do not touch but are related through a chain of 0-length edges between one's left side and the other's right side to lie on opposite sides of a vertical line in the layout (e.g., components B and H in Figure 8.5a). This is an added restriction to the layout, but an important one for the correctness of the algorithm presented later for the sizing of **slicing floorplans**.

Given an actual width for each component and having constructed the horizontal constraint graph as described in the preceding paragraph, to determine the minimum width of the rectangular layout area, one simply finds the longest path from the vertex representing the left side of the layout rectangle to the vertex representing the right side of the layout rectangle. To simplify the problem to one in an acyclic graph, vertices connected by pairs of 0-length edges can be collapsed into a single vertex; only the longest edge between each pair of (collapsed) vertices is needed. Then topological sort can be used to find the longest path between the left side and the right side of the floorplan, as discussed in Section 8.3.

We now have the machinery to state the problem of interest:

*Floorplan sizing:* Given a set $C$ of components, a piecewise linear shape function for each component, and a floorplan for $C$, find an assignment of specific shapes to the components so that the area of the layout is minimized.

Stockmeyer [39] showed that for general floorplans, the floorplan sizing problem is NP-complete. This holds even if the components are of fixed shape, but can be rotated 90°. In this case, the shape function of each component is a step function with at most two steps: $s(x) = d_2$ for $d_1 \leq x < d_2$ and $s(x) = d_1$ for $d_2 \leq x$, where the dimensions of the component are $d_1$ and $d_2$ with $d_1 \leq d_2$. However, for floorplans of a special form, called slicing floorplans, Stockmeyer gave a polynomial-time algorithm for the floorplan sizing problem when components are of fixed shape but can rotate. Otten [29] generalized this result to any piecewise-linear shape function. A slicing floorplan is one in which the partition of the layout rectangle can be constructed by a recursive cutting of a rectangular region into two subregions using either a vertical or horizontal line segment (see Figure 8.6). The rectangular regions that are not further partitioned are the subrectangles corresponding to components. The recursive partitioning method of constructing a placement discussed earlier produces a slicing floorplan.

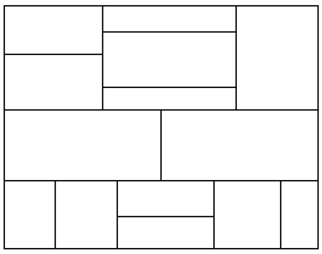
**FIGURE 8.6** A slicing floorplan.

A slicing floorplan can be represented by a binary tree. The root of the tree represents the entire layout rectangle and is labeled with the direction of the first cut. Other interior vertices represent rectangular subregions that are to be further subdivided, and the label on any such vertex is the direction of the cut used to subdivide it. The two children of any vertex are the rectangular regions resulting from cutting the rectangle represented by the vertex. The leaves of the binary tree represent the rectangles corresponding to components.

The algorithm for the sizing of a slicing floorplan uses the binary tree representation in a fundamental way. The key observation is that one only needs the shape functions of the two subregions represented by the children of a vertex to determine the shape function of a vertex. If the shape functions can be represented succinctly and combined efficiently for each vertex, the shape function of the root can be determined efficiently. We will present the combining step for shape functions that are step functions (i.e., piecewise constant) following the description in [24], since it illustrates the technique but is a straightforward calculation. Otten [29] shows how to combine piecewise linear slicing functions, but Lengauer [24] comments that arithmetic precision can become an issue in this case.

We shall represent a shape function that is a step function by a list of pairs $(w_i, s(w_i))$ for $0 \leq i \leq b_s$ and $w_0 = w_{\min}$. The interpretation is that for all $x$, $w_i \leq x < w_{i+1}$ (with $w_{b_s+1} = \infty$), $s(x) = s(w_i)$. Parameter $b_s$ is the number of breaks in the step function. The representation of the function is linear in the number of breaks. (This represents a step function whose constant intervals are left closed and right open and is the most logical form of step function for shape functions. However, other forms of step functions also can be represented in size linear in the number of breaks.)

Given step functions, $s_l$ and $s_r$, for the shapes of two children of a vertex, the shape function, $s$, for the vertex will also be a step function. When the direction of the cut is horizontal, the shape functions can simply be added, that is, $s(x) = s_l(x) + s_r(x)$. $w_{\min}$ for $s$ is the maximum of $w_{\min,l}$ for $s_l$ and $w_{\min,r}$ for $s_r$. Each subsequent break point for $s_l$ or $s_r$ is a break point for $s$, so that $b_s \leq b_{s_l} + b_{s_r}$. Combining the shape functions takes time $O(b_{s_l} + b_{s_r})$. When the direction is vertical, the step functions must first be inverted, the inverted functions combined, and then the combined function inverted back. The inversion of a step function $s$ is straightforward and can be done in $O(b_s)$ time.

To compute the shape function for the root of a slicing floorplan, one simply does a postorder traversal of the binary tree (see Chapter 7 of *Algorithms and Theory of Computation Handbook, Second Edition: General Concepts and Techniques*), computing the shape function for each vertex from the shape functions for the children of the vertices. The number of breaks in the shape function for any vertex is no more than the number of breaks in the shape functions for the leaves of the subtree rooted at that vertex. Let $b$ be the maximum number of breaks in any shape function of a component. Then, the running time of this algorithm for a slicing floorplan with $n$ components is

$$\begin{aligned} T(n) &\leq T(n_l) + T(n_r) + bn && (8.12)\\ &\leq dbn, && (8.13) \end{aligned}$$

where $d$ is the depth of the tree (the length of the longest path from the root to a leaf). We have the following:

> Given an instance of the floorplan sizing problem that has a slicing floorplan and step functions as shape functions for the components, there is an $O(dbn)$-time algorithm to compute the shape function of the layout rectangle.

Given the shape function for the layout rectangle, the minimum area shape can be found by computing the area at each break in linear time in the number of breaks, which is at most $O(bn)$.

Shi [38] has presented a modification to this technique that improves the running time for imbalanced slicing trees. For general (nonslicing) floorplans, many heuristics have been proposed: for example, Pan and Liu [31] present a generalization of the slicing floorplan technique to general floorplans.

## 8.5 Routing Problems

We shall only discuss the most common routing model for general cell placement—the rectilinear **channel-routing** model. In this model, the layout is rectilinear. The regions of the layout that are not covered by components are partitioned into nonoverlapping rectangles, called **channels**. The allowed partitions are restricted so that each channel only has components touching its horizontal edges (a horizontal channel) or its vertical edges (a vertical channel). These edges will be referred to as the "top" and "bottom" of the channel, regardless of whether the channel is horizontal or vertical. The orthogonal edges, referred to as the "left edge" and "right edge" of the channel, can only touch another channel. These channels compose the area for the wire routing. There are several strategies for partitioning the routing area into channels, that is "defining" the channels, but most use maximal rectangles where possible (i.e., no two channels can be merged to form a larger channel). The layout area becomes a rectangle that is partitioned into subrectangles of two types: components and channels (see Figure 8.7).

Given a layout with channels defined, the routing problem can be decomposed into two subproblems: **global routing** and local or detailed routing. **Global routing** is the problem of choosing which channels will be used to make the interconnections for each net. Actual paths are not produced. By doing global routing first, one can determine the actual paths for wires in each channel separately. The problem of detailed routing is to determine these actual paths and is more commonly referred

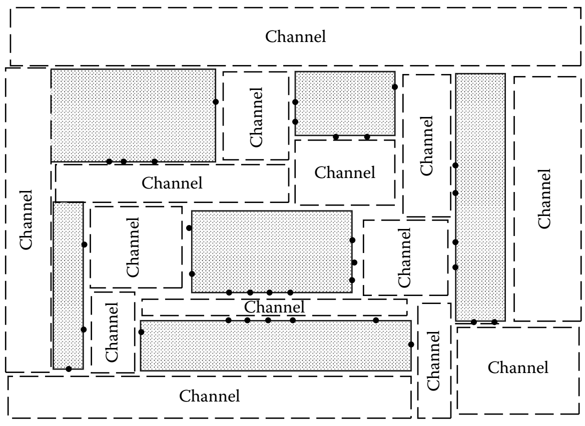

**FIGURE 8.7** The decomposition of a layout into routing channels and components.

to as “channel routing.” Of course, the segments of a wire path in each channel must join at the edges of the channel to produce an actual interconnection of terminals. To understand the approaches for handling this interfacing, we must have a more detailed definition of the channel-routing problem, which we give next.

The channel-routing problem is defined so that there are initial positional constraints in only one dimension. Recall that we define channels to abut components on only two parallel sides, the “top” and “bottom.” This is so that the routes in the channel will only be constrained by terminal positions on two sides. The standard channel-routing problem has the following input: a rectangle (the channel) containing points (terminals) at fixed positions along its top and bottom edges, a set of nets that must be routed in the channel, and an assignment of each of the terminals on the channel to one of these nets. Moreover, two (possibly empty) subsets of nets are specified: one containing nets whose routing must interface with the left edge of the channel and one containing nets whose routing must interface with the right edge of the channel. The positions at which wires must intersect the left and right edges of the channel are not specified. The dimension of the channel from the left edge to the right edge is called the length of the channel; the dimension from the top edge to the bottom edge is the width of the channel (see Figure 8.8). Since there are no terminals on the left and right edges, the width is often taken to be variable, and the goal is to find routing paths achieving the connections specified by the nets and minimizing the width of the channel. In this case, the space needed for the wires determines the width of the channel. The length of the channel is more often viewed as fixed, although there are channel-routing models in which routes are allowed to cross outside the left and right edges of the channel.

We can now discuss the problem of interfacing routes at channel edges. There are two main approaches to handling the interfacing of channels. One is to define the channels so that all adjacent channels form ⊤s (no + s). Then, if the channel routing is done for the base of the ⊤ first, the positions of paths leaving the base of the ⊤ and entering the crosspiece of the ⊤ are fixed by the routing of the base and can be treated as terminal positions for the crosspiece of the ⊤. Using this approach places constraints on the routing order of the channels. We can model these constraints using a directed graph: there is a vertex, $v_C$, for each channel C and an edge from $v_A$ to $v_B$ if channel A and channel B abut with channel A as the base of the ⊤ and channel B as the crosspiece of the ⊤. The edge from $v_A$ to $v_B$ represents that channel A must be routed before channel B. This graph must be acyclic for the set of constraints on the order of routing to be feasible. If the graph is not acyclic, another method must be used to deal with some of the channel interfaces. Slicing floorplans are very good for this approach because if each slice is defined to be a channel, then the channel order constraint graph will be acyclic.

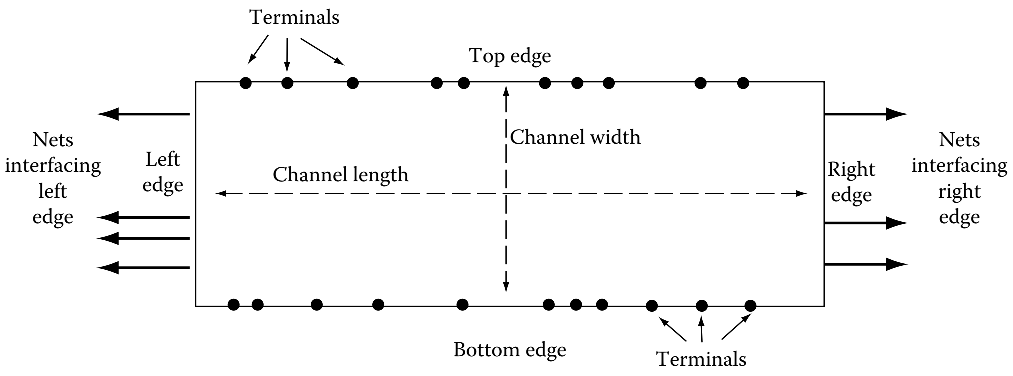

**FIGURE 8.8** A channel. Nets interfacing at the left and right edges are not in a given order.

The second alternative for handling the interfaces of channels is to define a special kind of channel, called a **switch box**, which is constrained by terminal locations on all four sides. In this alternative, some or all of the standard channels abut switch boxes. Special algorithms are used to do the switch box routing, since, practically speaking, this is a more difficult problem than standard channel routing.

## 8.6 Global Routing

Since global routing need not produce actual paths for wires, the channels can be modeled by a graph, called the channel intersection graph. For each channel, project the terminals lying on the top and bottom of the channel onto the midline of the channel. Each channel can be divided into segments by the positions of these projected terminals. The channel intersection graph is an undirected graph with one vertex for each projected terminal and one vertex for each intersection of channels. There is one edge for each segment of a channel, which connects the pair of vertices representing the terminals and/or channel intersections bounding the segment. A length and a capacity can be assigned to each edge, representing, respectively, the length between the ends of the channel segment and the width of the channel. Different versions of the global routing problem use one or both of the length and capacity parameters.

Given the channel intersection graph, the problem of finding a global routing for a set of nets becomes the problem of finding a Steiner tree in the channel intersection graph for the terminals of each net. Earlier in this chapter, we defined Steiner trees for points in the plane. For a general graph $G = (V, E)$, a Steiner tree for a subset of vertices $U \subset V$ is a set of edges of $G$ that form a tree in $G$ and whose set of endpoints contains the set $U$. Various versions of the global routing problem are produced by the constraints and optimization criteria used. For example, one can simply ask for the minimum length Steiner tree for each net, or one can ask for a set of Steiner trees that does not violate capacity constraints and has total minimum length. For each edge, the capacity used by a set of Steiner trees is the number of Steiner trees containing that edge; this must be no greater than the capacity of the edge. Another choice is not to have constraints on the capacity of edges but to minimize the maximum capacity used for any edge. In general, any combination of constraints and cost functions on length and capacity can be used. However, regardless of the criteria, the global routing problem is invariably NP-complete. A more detailed discussion of variations of the problem and their complexity can be found in [24]. There, a number of sophisticated algorithms for Steiner tree problems are also discussed. Here we will only discuss two techniques based on basic graph algorithms: breadth-first search and Dijkstra's single-source shortest path algorithm.

The minimum Steiner tree problem is itself NP-complete (see pages 208–209 in [10]). Therefore, one approach to global routing is to avoid finding Steiner trees by breaking up each net into a collection of point-to-point connections. One way to do this is to find the minimum Euclidean or rectilinear spanning tree for the terminals belonging to each net (ignoring the channel structure), and use the edges of this tree to define the point-to-point connections. Then one can use Dijkstra's single-source shortest path algorithm on the channel intersection graph to find a shortest path for each point-to-point connection. Paths for connections of the same net that share edges can then be merged, yielding a Steiner tree. If there are no capacity constraints on edges, the quality of this solution is only limited by the quality of the approximation of a minimum Steiner tree by the chosen collection of point-to-point paths. If there are capacity constraints, then after solving each shortest path problem, one must remove from the channel intersection graph the edges whose used capacity already equals the edge capacity. In this case, the order in which nets and terminals within a net are routed is significant. Heuristics are used to choose this order. One can better approximate Steiner trees for the nets by, at each iteration for connections within one net, choosing a terminal not yet connected to any other terminals in the net as the source of Dijkstra's algorithm.

Since this algorithm computes the shortest path to every other vertex in the graph from the source, the shortest path, which connects to any other vertex in the channel intersection graph that is already on a path connecting terminals of the net can be used. Of course, there are variations on this idea.

For any graph, breadth-first search from a vertex $v$ will find a shortest path from $v$ to every other vertex in the graph when the length of each edge is 1. Breadth-first search takes time $O(|V| + |E|)$ compared to the best worst-case running time known for Dijkstra's algorithm: $O(|V| \log |V| + |E|)$ time. It is also very straightforward to implement. It is easy to incorporate heuristics that take into account the capacity of an edge already used and bias the search toward edges with little capacity used. Furthermore, breadth-first search can be started from several vertices simultaneously, so that all terminals of a net could be starting points of the search simultaneously. If it is adequate to view all channel segments as being of equal length, then the edge lengths can all be assigned value 1 and breadth-first search can be used. This might occur when the terminals are uniformly distributed and so divide channels into approximately equal-length segments. Alternatively, one can add new vertices to the channel intersection graph to further decompose the channel segments into unit-length segments. This can substantially increase $|V|$ and $|E|$ so that they are proportional to the dimensions of the underlying grid defining the unit of length rather than the number of channels and terminals in the problem. However, this allows one to use breadth-first search to compute shortest paths while modeling the actual lengths of channels. In fact, breadth-first search was developed by Lee [20]* for the routing of circuit boards in exactly this manner. He modeled the entire board by a grid graph, and modeled obstacles to routing paths as grid vertices that were missing from the graph. Each wire route was found by doing a breadth-first search in the grid graph.

## 8.7 Channel Routing

Channel routing is not one single problem, but rather a family of problems based on the allowable paths for wires in the channel. We will limit our discussion to grid-based routing. While both grid-free rectilinear and nonrectilinear routing techniques exist, the most basic techniques are grid-based. We assume that there is a grid underlying the channel, the sides of the channel lie on grid edges, terminals lie on grid points, and all routing paths must follow edges in the grid. For ease of discussion, we shall refer to channel directions as though the channel were oriented with its length running horizontally. The vertical segments of the grid that run from the top to the bottom of the channel are referred to as columns. The horizontal segments of the grid that run from the left edge to the right edge of the channel are referred to as tracks. We will consider channel-routing problems that allow the width of the channel to vary. Therefore, the number of columns, determining the channel length, will be fixed, but the number of tracks, determining the channel width, will be variable. The goal is to minimize the number of tracks used to route the channel.

The next distinction is based on how many routing layers are presumed. If there are $\ell$ routing layers, then there are $\ell$ overlaid copies of the grid, one for each layer. Routes that use the same edge on different layers do not interact and are considered disjoint. Routes change layer at grid points. The exact rules for how routes can change layers vary, but the most common is to view a route that goes from layer $i$ to layer $j$ $(j > i)$ at a grid point as using layers $i + 1, \ldots, j - 1$ as well at the grid point.

* The first published description of breadth-first search was by E.F. Moore for finding a path in a maze. Lee developed the algorithm for routing in grid graphs under a variety of path costs. See the discussion on page 394 of [24].

One can separate the channel-routing problem into two subproblems: finding Steiner trees in the grid that achieve the interconnections, and finding a **layer assignment** for each edge in each Steiner tree so that the resulting set of routes is legal. One channel-routing model for which this separation is made is knock-knee routing (see Section 9.5 of [24]). Given any set of trees in the grid (not only those for knock-knee routes), a legal layer assignment can be determined efficiently for two layers. Maximal single-layer segments in each tree can be represented as vertices of a conflict graph, with segments that must be on different layers related by conflict edges. Then finding a legal two-layer assignment is equivalent to two-coloring the conflict graph. A more challenging problem is **via minimization**, which is to find a legal layer assignment that minimizes the number of grid points at which layer change occurs. This problem is also solvable in polynomial time as long as none of the Steiner trees contain a four-way split (a technical limitation of the algorithm). For more than two layers, both layer assignment and via minimization are NP-complete (see Section 9.5.3 of [24]).

We have chosen to discuss two routing models in detail: single-layer routing and two-layer **Manhattan routing**. Routing models that allow more than two layers, called multilayer models, are becoming more popular as technology is able to provide more layers for wires. However, many of the multilayer routing techniques are derived from single-layer or two-layer Manhattan techniques, so we focus this restricted discussion on those models. A more detailed review of channel-routing algorithms can be found in [19]. Neither model requires layer assignment. In single-layer routing there is no issue of layer assignment; in Manhattan routing, the model is such that the layer assignment is automatically derived from the paths. Therefore, for each model, our discussion need only address how to find a collection of Steiner trees that satisfy the restrictions of the routing model.

## 8.7.1 Manhattan Routing

Manhattan routing is the dominant two-layer routing model. It dates back to printed circuit boards [13]. It dominates because it finesses the issue of layer assignment by defining all vertical wire segments to be on one layer and all horizontal wire segments to be on the other layer. Therefore, a horizontal routing path and a vertical routing path can cross without interacting, but any path that bends at a grid point is on both layers at that point and no path for a disjoint net can use the same point. Thus, under the Manhattan model, the problem of routing can be stated completely in terms of finding a set of paths such that paths for distinct nets may cross but do not share edges or bend points.

Although Manhattan routing provides a simple model of legal routes, the resulting channel-routing problem is NP-complete. An important lower bound on the width of a channel is the **channel density**. The density at any vertical line cutting the channel (not necessarily a column) is the number of nets that have terminals both to the left and the right of the vertical line. The interpretation is that each net must have at least one wire crossing the vertical line, and thus a number of tracks equal to the density at the vertical line is necessary. For columns, nets that contain a terminal on the column are counted in the density unless the net contains exactly two terminals and these are at the top and bottom of the column. (Such a net can be routed with a vertical path the extent of the column.) The channel density is the maximum density of any vertical cut of the channel. In practice, the channel-routing problem is solved with heuristic algorithms that find routes giving a channel width within one or two of density, although such performance is not provably achievable.

If a channel-routing instance has no top terminal and bottom terminal on the same column, then a number of tracks equal to the channel density suffices and a route achieving this density can be solved in $O(m + n \log n)$ time, where $n$ is the number of nets and $m$ is the number of terminals. Under this restriction, the channel-routing problem becomes equivalent to the problem of **interval graph coloring**. The equivalence is based on the observation that if no column contains terminals at its top and bottom, then any terminal can connect to any track by a vertical

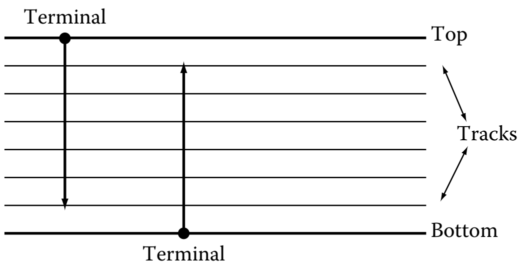

**FIGURE 8.9** Connecting to tracks without conflicts. Each of the nets can reach any of the tracks with a vertical segment in the column containing the terminal.

segment that does not conflict with any other path (see Figure 8.9). Each net can use one horizontal segment that spans all the columns containing its terminals. The issue is to pack these horizontal segments into the minimum number of tracks so that no segments intersect. Equivalently, the goal is to color the segments (or intervals) with the minimum number of colors so that no two segments that intersect are of the same color. Hence, we have the relationship to interval graphs, which have a set of vertices that represent intervals on a line and edges between vertices of intersecting intervals.

A classic greedy algorithm can assign a set $I$ of intervals to $d$ tracks, where $d$ is the **channel density**. Intervals are placed in tracks in order of their left endpoints; all tracks are filled with intervals from left to right concurrently (the actual position of each track does not matter). The set of interval endpoints is first sorted so that the combined order of left (starting) endpoints and right (ending) endpoints is known. At any point in the algorithm, there are tracks containing intervals that have not yet ended and tracks that are free to receive new intervals. A queue $F$ is used to hold the free tracks; the function FREE inserts (enqueues) a track in $F$. The unprocessed endpoints are stored in a queue of points, $P$, sorted from left to right. The function DEQUEUE is the standard deletion operation for a queue (see Chapter 4 of *Algorithms and Theory of Computation Handbook, Second Edition: General Concepts and Techniques* or [6]). The algorithm starts with only one track and adds tracks as needed; variable $t$ holds the number of tracks used.

```
INTERVAL-BY-INTERVAL ASSIGNMENT ($I$)

1  Sort the endpoints of the intervals in $I$ and build queue $P$
2  $t \leftarrow 1$
3  FREE track 1
4  while $P$ is not empty
5     do $p \leftarrow$ DEQUEUE($P$)
6        if $p$ is the left endpoint of an interval $i$
7           then do  if $F$ is empty
8                       then do $t \leftarrow t + 1$
9                               Put $i$ in track $t$
10                      else do $track \leftarrow$ DEQUEUE(F)
11                              Put $i$ in $track$
12          else do FREE the track containing the interval whose right endpoint is $p$
```

To see that this algorithm never uses more than a number of tracks equal to the density $d$ of the channel, note that when $t$ is increased at line 8 it is because the current $t$ tracks all contain intervals that have not yet ended when the left endpoint $p$ obtained in line 5 is considered. Therefore, when $t$ is increased for the last time to value $w$, all tracks numbered less than $w$ contain intervals that span the current $p$; hence $d \geq w$. Since no overlapping intervals are places in the same track, $w = d$. Therefore, the algorithm finds an optimal track assignment. Preprocessing to find the interval for each net takes time $O(m)$ and INTERVAL-BY-INTERVAL ASSIGNMENT has running time $O(|I| \log |I|) = O(n \log n)$, due to the initial sorting of the endpoints of $I$.

Once one allows terminals at both the top and the bottom of a column (except when all such pairs of terminals belong to the same net), one introduces a new set of constraints called vertical constraints. These constraints capture the fact that if net $i$ has a terminal at the top of column $c$ and net $j$ has a terminal at the bottom of column $c$, then to connect these terminals to horizontal segments using vertical segments at column $c$, the horizontal segment for $i$ must be above the horizontal segment for $j$. One can construct a vertical constraint graph that has a vertex $v_i$ for each net $i$ and a directed edge between $v_i$ and $v_j$ if there is a column that contains a terminal in $i$ at the top and a terminal in $j$ at the bottom. If one considers only routes that use at most one horizontal segment per net, then the constraint graph represents order constraints on the tracks used by the horizontal segments. If the vertical constraint graph is cyclic, then the routing cannot be done with one horizontal segment per net. If the vertical constraint graph is acyclic, it is NP-complete to determine if the routing can be achieved in a given number of tracks (see [24, p. 547]). Furthermore, even if an optimal or good routing using one horizontal segment per net is found, the number of tracks required is often substantially larger than what could be obtained using more horizontal segments. For these reasons, practical channel-routing algorithms allow the route for each net to traverse portions of several tracks. Each time a route changes from one track to another, it uses a section of a column; this is called a **jog** (see Figure 8.10).

Manhattan channel routing remains NP-complete even if unrestricted jogs are allowed (see [24, p. 541]). Therefore, the practical routing algorithms for this problem use heuristics. The earliest of these is by Deutsch [7]. Deutsch allows the route for a net to jog only in a column that contains a terminal of the net; he calls these jogs “doglegs” (see Figure 8.10). This approach effectively breaks up each net into two-point subnets, and one can then define a vertical constraint graph in which each vertex represents a subnet. Deutsch’s algorithm is based on a modification of INTERVAL-BY-INTERVAL ASSIGNMENT called track-by-track assignment. Track-by-track assignment also fills tracks greedily from left to right but fills one track to completion before starting the next. Deutsch’s basic algorithm does track-by-track assignment but does not assign an interval for a subnet to a track if the assignment would violate a vertical constraint. Embellishments on the basic algorithm try to improve its performance and minimize the number of doglegs. Others have also modified the approach

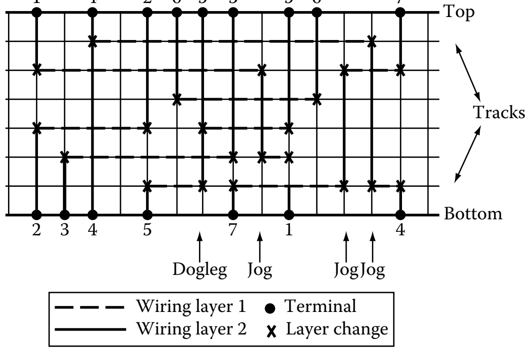

**FIGURE 8.10** A channel routing in the two-layer Manhattan model showing jogs.

(see the discussion in Section 9.6.1.4 of [24]). The class of algorithms based on greedy assignment of sorted intervals is also known as "left-edge algorithms."

Manhattan channel routing is arguably the most widely used and detailed routing model, and many algorithms have been developed. The reader is referred to [37] or [19] for a survey of algorithms. In this chapter, we will discuss only one more algorithm—an algorithm that proceeds column-by-column in contrast to the track-by-track algorithm. The column-by-column algorithm was originally proposed by Rivest and Fiduccia [35] and was called by them a "greedy" router. This algorithm routes all nets simultaneously. It starts at the leftmost column of the channel and proceeds to the right, considering each column in order. As it proceeds left to right it creates, destroys, and continues horizontal segments for nets in the channel. Using this approach, it is easy to introduce a jog for a net in any column. At each column, the algorithm connects terminals at the top and the bottom to horizontal segments for their nets, starting new segments when necessary, and ending segments when justified. At each column, for each continuing net with terminals to the right, it may also create a jog to bring a horizontal segment of the route closer to the channel edge containing the next terminal to the right. Thus, the algorithm employs some "look ahead" in determining what track to use for each net at each column. The algorithm is actually a framework with many parameters that can be adjusted. It may create, for one net, multiple horizontal segments that cross the same column and may extend routes beyond the left and the right edges of the channel. It is a very flexible framework that allows many competing criteria to be considered and allows the interaction of nets more directly than strategies that route one net at a time. Many routing tools have adopted this approach.

## 8.7.2 Single-Layer Routing

From a practical perspective, single-layer channel routing is needed to route bus structures and plays a role in the routing of multilayer channels (e.g., [11]). For bus structures, additional performance constraints are often present (e.g., [30]). From a theoretical perspective, single-layer channel routing is of great significance because, even in its most general form, it can be solved optimally in polynomial time. There is a rich theory of single-layer detailed routing that has been developed not only for channel routing, but also for routing in more general regions (see [27]). The first algorithmic results for single-layer channel routing were by Tompa [40], who considered **river routing** problems. A river routing problem is a single-layer channel-routing problem in which each net contains exactly two terminals, one at the top edge of the channel and one at the bottom edge of the channel. The nets have terminals in the same order along the top and the bottom—a requirement if the problem is to be routable in one layer. Tompa considered unconstrained (versus rectilinear) wire paths and gave an $O(n^2)$-time algorithm for $n$ nets to test routability and find the route that minimizes both the individual wire lengths and the total wire length when the width of the channel is fixed. This algorithm can be used as a subroutine within binary search to find the minimum-width channel in $O(n^2 \log n)$ time. Tompa also suggested how to modify his algorithm for the rectilinear case. Dolev et al. [8] built upon Tompa's theory for the rectilinear case and presented an $O(n)$-time algorithm to compute the minimum width of the channel and an $O(n^2)$-time algorithm to actually produce the rectilinear routing. The difference in running times comes from the fact that the routing may actually have $n$ segments per net, and thus would take $O(n^2)$-time to generate (see Figure 8.11). In contrast, the testing for routability can be done by examining a set of constraints for the channel. The results were generalized to multiterminal nets by Greenberg and Maley [12], where the time to calculate the minimum width remains linear in the number of terminals.

We now present the theory that allows the width of a river-routing channel to be computed in linear time. Our presentation is an amalgam of the presentations in [8] and [23]. The heart of the

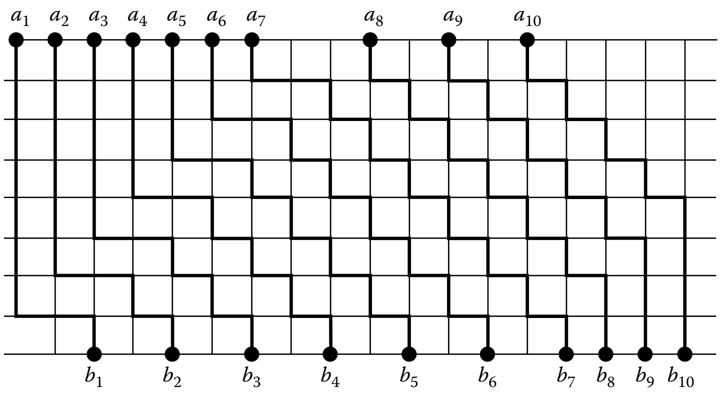
$a_1$ $a_2$ $a_3$ $a_4$ $a_5$ $a_6$ $a_7$ $a_8$ $a_9$ $a_{10}$
$b_1$ $b_2$ $b_3$ $b_4$ $b_5$ $b_6$ $b_7$ $b_8$ $b_9$ $b_{10}$

**FIGURE 8.11** A river routing. The route for a single net may bend $O(n)$ times.

theory is the observation that for river routing, cut lines other than the vertical lines that define channel density also contribute to a lower bound for the width of the channel. This lower bound is then shown to be an upper bound as well. Indexing from left to right, let the $i$th net of a routing channel, $1 \leq i \leq n$, be denoted by the pair $(a_i, b_i)$ where $a_i$ is the horizontal position of its terminal on the top of the channel and $b_i$ is the horizontal position of its terminal on the bottom of the channel. Consider a line from $b_i$ to $a_{i+k}$, $k > 0$, cutting the channel. There are $k + 1$ nets that must cross this line. Measuring slopes from 0° to 180°, if the line has slope $\geq$90°, then $k + 1$ nets must cross the vertical (90°) line at $b_i$, and there must be $k + 1$ tracks. If the line has slope <90° and >45°, then each vertical grid segment that crosses the line can be paired with a horizontal grid segment that must also cross the line and which cannot be used by a different net. Therefore, the line must cross $k + 1$ tracks. Finally, if the line has slope $\leq$45°, then each horizontal grid segment that crosses the line can be paired with a vertical grid segment that must also cross the line and which cannot be used by a different net. Therefore, there must be $k + 1$ columns crossing the line, that is, $a_{i+k} - b_i \geq k$. Similarly, by considering a line from $b_{i+k}$ to $a_i$, we conclude $k + 1$ tracks are necessary unless $b_{k+i} - a_i \geq k$. In the case of $k = 0$, $a_i$ must equal $b_i$ for no horizontal track to be required. Based on this observation, it can be proved that the minimum number of tracks $t$ required by an instance of river routing is the least $t$ such that for all $1 \leq i \leq n - t$

$$b_{i+t} - a_i \geq t \tag{8.14}$$

and

$$a_{i+t} - b_i \geq t. \tag{8.15}$$

To find the minimum such $t$ in linear time, observe that $a_{i+k+1} \geq a_{i+k} + 1$ and $b_{i+k+1} \geq b_{i+k} + 1$. Therefore,

$$\text{if } b_{i+k} - a_i \geq k \text{ then } b_{i+k+1} - a_i \geq b_{i+k} + 1 - a_i \geq k + 1 \tag{8.16}$$

and

$$\text{if } a_{i+k} - b_i \geq k \text{ then } a_{i+k+1} - b_i \geq a_{i+k} + 1 - b_i \geq k + 1. \tag{8.17}$$

Therefore, we can start with $t = 0$ and search for violated constraints from $i = 1$ to $n - t$; each time a constraint of the form of (8.14) or (8.15) above is violated, we increase $t$ by one and continue the search; $t$ can be no larger than $n$. Let $N$ denote the set of nets in a river routing channel, $|N| = n$. The following algorithm calculates the minimum number of tracks needed to route this channel.

RIVER-ROUTING WIDTH ($N$)

```
1  $i \leftarrow 1$
2  $t \leftarrow 0$
3  while $i \leq |N| - t$
4     do if $b_{i+t} - a_i \geq t$ and $a_{i+t} - b_i \geq t$
5        then do $i \leftarrow i + 1$
6        else do $t \leftarrow t + 1$
7  return $t$
```

The actual routing for a river-routing channel can be produced in a greedy fashion by routing one net at a time from left to right, and routing each net beginning with its left terminal. The route of any net travels vertically whenever it is not blocked by a previously routed net, travels horizontally right until it can travel vertically again or until it reaches the horizontal position of the right terminal of the net. This routing takes worst-case time $O(n^2)$, as the example in Figure 8.11 illustrates.

## 8.8 Research Issues and Summary

This chapter has given an overview of the design problems arising from the computer-aided layout of VLSI circuits and some of the algorithmic approaches used. The algorithms presented in this chapter draw upon the theoretical foundations discussed in other chapters. Graph models are predominant and are frequently used to capture constraints. Since many of the problems are NP-complete, heuristics are used. Research continues in this field, both to find better methods of solving these difficult problems—both in the efficiency and the quality of solution—and to model and solve new layout problems arising from the ever-changing technology of VLSI fabrication and packaging and the ever-increasing complexity of VLSI circuits. Layout techniques particular to increasingly popular technologies such as field-programmable gate arrays (FPGAs) have been and continue to be developed. System-on-chip (SoC) designs have brought new challenges due not only to their increased complexity but also to the use of a mixture of design styles for one chip.

A major theme of current research in VLSI layout is the consideration of circuit performance as well as layout area. As feature sizes continue to shrink, wire delay is becoming an increasingly large fraction of total circuit delay. Therefore, the delay introduced by routing is increasingly important. The consideration of performance has necessitated new techniques, not only for routing but also for partitioning and placement as well. In some cases, the techniques for area-based layout have been extended to consider delay. For example, the simulated annealing approach to placement has been modified to consider delay on critical paths. However, many researchers are developing new approaches for performance-based layout. For example, one finds the use of the techniques of linear programming, integer programming, and even specialized higher-order programming (see Chapters 30 and 31 of *Algorithms and Theory of Computation Handbook, Second Edition: General Concepts and Techniques*) in recent work on performance-driven layout. Clock-tree routing, to minimize clock delay and clock skew, is also receiving attention as an important component of performance-driven layout.

The reader is referred to the references given in "Further Information" for broader descriptions of the field and, in particular, for more thorough treatments of current research.

## 8.9 Further Information

This chapter has provided several examples of the successful application of the theory of combinatorial algorithms to problems in VLSI layout. It is by no means a survey of all the important problems and algorithms in the area. Several textbooks have been written on algorithms for VLSI layout, such as [24,34,37,42], and the reader is referred to these for more complete coverage of the area. Other review articles of interest are [33] and [5] on placement, [2] on geometric algorithms for layout, [18] on layout, [3] on relaxation techniques for layout problems, [19] on channel routing, [15] on layer assignment, and [1] and [32] on netlist partitioning. There are many conferences and workshops on CAD that contain papers presenting algorithms for layout. The *ACM/IEEE Design Automation Conference*, the *IEEE/ACM International Conference on Computer-Aided Design*, and the *ACM International Symposium on Physical Design* have traditionally been good sources of algorithms among these conferences. Major journals on the topic are *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *IEEE Transactions on VLSI Systems*, *ACM Transactions on Design Automation of Electronic Systems*, and *Integration, the VLSI Journal* published by North-Holland Publishing. Layout algorithms also appear in journals focusing on general algorithm development such as *SIAM Journal on Computing,* published by the Society of Industrial and Applied Mathematics, *Journal of Algorithms,* published by Elsevier, and *Algorithmica,* published by Springer-Verlag.

## Defining Terms

**Channel:** A rectangular region for routing wires, with terminals lying on two opposite edges, called the "top" and the "bottom." The other two edges contain no terminals, but wires may cross these edges for nets that enter the channel from other channels. The routing area of a layout is decomposed into several channels.

**Channel density:** Orient a channel so that the top and the bottom are horizontal edges. Then the density at any vertical line cutting the channel is the number of nets that have terminals both to the left and the right of the vertical line. Nets with a terminal on the vertical line contribute to the density unless all of the terminals of the net lie on the vertical line. The channel density is the maximum density of any vertical cut of the channel.

**Channel routing:** The problem of determining the routes, that is, paths and layers, for wires in a routing channel.

**Compaction:** The process of modifying a given layout to remove extra space between features of the layout.

**Floorplan:** An approximate layout of a circuit that is made before the layouts of the components composing the circuit have been completely determined.

**Floorplan sizing:** Given a floorplan and a set of components, each with a shape function, finding an assignment of specific shapes to the components so that the area of the layout is minimized.

**Floorplanning:** Designing a floorplan.

**General cell layout:** A style of layout in which the components may be of arbitrary height and width and functional complexity.

**Global routing:** When a layout area is decomposed into channels, global routing is the problem of choosing which channels will be used to make the interconnections for each net.

**Graph partitioning:** Given a graph with weights on its vertices and costs on its edges, the problem of partitioning the vertices into some given number $k$ of approximately equal-weight subsets such that the cost of the edges that connect vertices in different subsets is minimized.

**Interval graph coloring:** Given a finite set of $n$ intervals $\{[l_i, r_i], 1 \leq i \leq n\}$, for integer $l_i$ and $r_i$, color the intervals with a minimum number of colors so that no two intervals that intersect are of the same color. The graph representation is direct: each vertex represents an interval, and there is an edge between two vertices if the corresponding intervals intersect. Then coloring the interval graph corresponds to coloring the intervals.

**Jog:** In a rectilinear routing model, a vertical segment in a path that is generally running horizontally, or vice versa.

**Layer assignment:** Given a set of trees in the plane, each interconnecting the terminals of a net, an assignment of a routing layer to each segment of each tree so that the resulting wiring layout is legal under the routing model.

**Manhattan routing:** A popular rectilinear channel-routing model in which paths for disjoint nets can cross (a vertical segment crosses a horizontal segment) but cannot contain segments that overlap in the same direction at even a point.

**Net:** A set of terminals to be connected together.

**Rectilinear:** With respect to layouts, describes a layout for which there is an underlying pair of orthogonal axes defining "horizontal" and "vertical;" the features of the layout, such as the sides of the components and the segments of the paths of wires, are horizontal and vertical line segments.

**River routing:** A single-layer channel-routing problem in which each net contains exactly two terminals, one at the top edge of the channel and one at the bottom edge of the channel. The nets have terminals in the same order along the top and bottom—a requirement if the problem is to be routable in one layer.

**Shape function:** A function that gives the possible dimensions of the layout of a component with a flexible (or not yet completely determined) layout. For a shape function $s : [w_{\min}, \infty] \rightarrow [h_{\min}, \infty]$ with $[w_{\min}, \infty]$ and $[h_{\min}, \infty]$ subsets of $\Re^+$, $s(w)$ is the minimum height of any rectangle of width $w$ that contains a layout of the component.

**Slicing floorplan:** A floorplan that can be obtained by the recursive bipartitioning of a rectangular layout area using vertical and horizontal line segments.

**Steiner tree:** Given a graph $G = (V, E)$ a Steiner tree for a subset of vertices $U$ of $V$ is a subset of edges of $G$ that form a tree and contain among their endpoints all the vertices of $U$. The tree may contain other vertices than those in $U$. For a Euclidean Steiner tree, $U$ is a set of points in the Euclidean plane, and the tree interconnecting $U$ can contain arbitrary points and line segments in the plane.

**Switch box:** A rectangular routing region containing terminals to be connected on all four sides of the rectangle boundary and for which the entire interior of the rectangle can be used by wires (contains no obstacles).

**Terminal:** A position within a component where a wire attaches. Usually a terminal is a single point on the boundary of a component, but a terminal can be on the interior of a component and may consist of a set of points, any of which may be used for the connection. A typical set of points is an interval along the component boundary.

**Topological sort:** Given a directed, acyclic graph, a topological sort of the vertices of the graph is a total ordering of the vertices such that if vertex $u$ comes before vertex $v$ in the ordering, there is no directed path from $v$ to $u$.

**Via minimization:** Given a set of trees in the plane, each interconnecting the terminals of a net, determining a layer assignment that minimizes the number of points (vias) at which a layer change occurs.

## Acknowledgment

I would like to thank Ron Pinter for his help in improving this chapter.

## References

1. Alpert, C.J. and Kahng, A.B., Recent directions in netlist partitioning: A survey, *Integration: The VLSI Journal.* 19, 1–81, 1995.
2. Asano, T., Sato, M., and Ohtsuki, T., Computational geometry algorithms. In *Layout Design and Verification,* Ohtsuki, T., Ed., pp. 295–347. North-Holland, Amsterdam, the Netherlands, 1986.
3. Chan, T.F., Cong, J., Shinnerl, J.R., Sze, K., Xie, M., and Zhang, Y., Multiscale optimization in VLSI physical design automation. In *Multiscale Optimization Methods and Applications,* Hager, W.W., Huang, S.-J., Pardalos, P.M., and Prokopyev, O.A., editors, pp. 1–68. Springer New York, 2006.
4. Chen, T.-C. and Chang, Y.-W., Modern floorplanning based on fast simulated annealing. In *Proceedings of the 2005 International Symposium on Physical Design,* pp. 104–112. ACM, New York, 2005.
5. Cong, J., Shinnerl, J.R., Xie, M., Kong, T., and Yuan, X., Large-scale circuit placement, *ACM Transactions on Design Automation of Electronic Systems.* 10(2), 389–430, 2005.
6. Cormen, T.H., Leiserson, C.E., Rivest, R.L., and Stein, C. *Introduction to Algorithms,* 2nd ed. MIT Press, Cambridge, MA, 2001.
7. Deutsch, D.N., A dogleg channel router. In *Proceedings of the 13th ACM/IEEE Design Automation Conference,* pp. 425–433. ACM, New York, 1976.
8. Dolev, D., Karplus, K., Siegel, A., Strong, A., and Ullman, J.D., Optimal wiring between rectangles. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing,* pp. 312–317. ACM, New York, 1981.
9. Fiduccia, C.M. and Mattheyses, R.M., A linear-time heuristic for improving network partitions. In *Proceedings of the 19th ACM/IEEE Design Automation Conference,* pp. 175–181. IEEE Press, Piscataway, NJ, 1982.
10. Garey, M.R. and Johnson, D.S., *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman, San Francisco, CA, 1979.
11. Greenberg, R.I., Ishii, A.T., and Sangiovanni-Vincentelli, A.L., MulCh: A multi-layer channel router using one, two and three layer partitions. In *IEEE International Conference on Computer-Aided Design,* pp. 88–91. IEEE Press, Piscataway, NJ, 1988.
12. Greenberg, R.I. and Maley, F.M., Minimum separation for single-layer channel routing. *Information Processing Letters.* 43, 201–205, 1992.
13. Hashimoto, A. and Stevens, J., Wire routing by optimizing channel assignment within large apertures. In *Proceedings of the 8th IEEE Design Automation Workshop,* pp. 155–169. IEEE Press, Piscataway, NJ, 1971.
14. Hwang, F.K., On Steiner minimal trees with rectilinear distance. *SIAM Journal on Applied Mathematics,* 30(1), 104–114, 1976.
15. Joy, D. and Ciesielski, M., Layer assignment for printed circuit boards and integrated circuits. *Proceedings of the IEEE.* 80(2), 311–331, 1992.
16. Kahng, A., Classical floorplanning harmful? In *Proceedings of the International Symposium on Physical Design,* pp. 207–213. ACM, New York, 2000.
17. Kernighan, W. and Lin, S., An efficient heuristic procedure for partitioning graphs. *Bell System Technical Journal,* 49, 291–307, 1970.
18. Kuh, E.S. and Ohtsuki, T., Recent advances in VLSI layout. *Proceedings of the IEEE.* 78(2), 237–263, 1990.
19. LaPaugh, A.S. and Pinter, R.Y., Channel routing for integrated circuits. In *Annual Review of Computer Science,* Vol. 4, J. Traub, Ed., pp. 307–363. Annual Reviews Inc., Palo Alto, CA, 1990.
20. Lee, C.Y., An algorithm for path connection and its applications. *IRE Transactions on Electronic Computers.* EC-10(3), 346–365, 1961.

21. Lee, T.-C., A bounded 2D contour searching algorithm for floorplan design with arbitrarily shaped rectilinear and soft modules. In *Proceedings of the 30th ACM/IEEE Design Automation Conference,* pp. 525–530. ACM, New York, 1993.
22. Lefebvre, M., Marple, D., and Sechen, C., The future of custom cell generation in physical synthesis. In *Proceedings of the 34th Design Automation Conference,* pp. 446-451. ACM, New York, 1997.
23. Leiserson, C.E. and Pinter, R.Y., Optimal placement for river routing. *SIAM Journal on Computing.* 12(3), 447–462, 1983.
24. Lengauer, T., *Combinatorial Algorithms for Integrated Circuit Layout.* John Wiley & Sons, West Sussex, England, 1990.
25. Liao, Y.Z. and Wong, C.K., An algorithm to compact a VLSI symbolic layout with mixed constraints. *IEEE Transactions on Computer-Aided Design.* CAD-2(2), 62–69, 1983.
26. Maley, F.M., An observation concerning constraint-based compaction. *Information Processing Letters.* 25(2), 119–122, 1987.
27. Maley, F.M., *Single-layer wire routing.* PhD thesis, Department of Electrical Engineering and Computer Science, MIT, Cambridge, MA, 1987.
28. Mata, J.M., Solving systems of linear equalities and inequalities efficiently. In *Congressus Numerantum*, vol 45, *Proceedings of the 15th Southeastern International Conference on Combinatorics*, Graph Theory and Computing, Utilitas Mathematica Pub. Inc., Winnipeg, Manitoba, Canada, 1984.
29. Otten, R.H.J.M., Efficient floorplan optimization. In *Proceedings of the International Conference on Computer Design: VLSI in Computers,* pp. 499–502. IEEE Press, Piscataway, NJ, 1983.
30. Ozdal, M.M. and Wong, M.D.F., A provably good algorithm for high performance bus routing. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design.* pp. 830–837. IEEE Computer Society, Washington, DC, 2004.
31. Pan, P. and Liu, C.L., Area minimization for floorplans. *IEEE Transactions on Computer-Aided Design.* CAD-14(1), 123–132, 1995.
32. Papa, D.A. and Markov, I.L. Hypergraph partitioning and clustering. In *Approximation Algorithms and Metaheuristics,* Gonzalez, T., Ed.. CRC Press, Boca Raton, FL, 2006.
33. Preas, B.T. and Karger, P.G., Automatic placement: A review of current techniques. In *Proceedings of the 23rd ACM/IEEE Design Automation Conference,* pp. 622–629. IEEE Press, Piscataway, NJ, 1986.
34. Preas, B.T. and Lorenzetti, M.J., Ed., *Physical Design Automation of VLSI Systems.* Benjamins/Cummings, Menlo Park, CA, 1988.
35. Rivest, R.L. and Fiduccia, C.M., A "greedy" channel router. In *Proceedings of the 19th ACM/IEEE Design Automation Conference,* pp. 418–422. IEEE Press, Piscataway, NJ, 1982.
36. Sechen, C., Chip-planning, placement, and global routing of macro/custom cell integrated circuits using simulated annealing. In *Proceedings of the 25th ACM/IEEE Design Automation Conference,* pp. 73–80. IEEE Computer Society Press, Los Alamitos, CA, 1988.
37. Sherwani, N., *Algorithms for VLSI Physical Design Automation,* 3rd edn. Springer, New York, 1999.
38. Shi, W., An optimal algorithm for area minimization of slicing floorplans. In *IEEE/ACM International Conference on Computer-Aided Design,* pp. 480–484. IEEE Computer Society, Washington, DC, 1995.
39. Stockmeyer, L., Optimal orientations of cells in slicing floorplan designs. *Information and Control.* 57, 91–101, 1983.
40. Tompa, M., An optimal solution to a wire-routing problem. *Journal of Computer and System Sciences.* 23(2), 127–150, 1981.
41. Wolf, W.H., *Modern VLSI Design: System-on-Chip Design,* 3rd edn. Prentice-Hall, Englewood Cliffs, NJ, 2002.
42. Youssef, H. and Sait, S., *VLSI Physical Design Automation: Theory and Practice,* World Scientific, Singapore, 1999.