Chapter 12

Compaction

After completion of detailed routing, the layout is functionally complete. At this stage, the layout is ready to be used to fabricate a chip. However, due to non-optimality of placement and routing algorithms, some vacant space is present in the layout. In order to minimize the cost, improve performance and yield, layouts are reduced in size by removing the vacant space without altering the functionality of the layout. This operation of layout area minimization is called *layout compaction.*

The compaction problem is simplified by using symbols to represent primitive circuit features, such as transistors and wires. The representation of layout using symbols is called a *symbolic layout.* There are special languages [Eic86, LM84a, Mat85] and special graphic editors [Hil84, Hsu79] to describe symbolic layouts. To produce the actual masks, the symbolic layouts are translated into actual geometric features. Although a feature can have any geometric shape, in practice only rectangular shapes are considered.

The goal of compaction is to minimize the total layout area without violating any design rules, maintaining good layout practices and without violating designer specified constraints. The last two objectives are usually motivated by performance verification. The area can be minimized in three different ways:

1. *By reducing the space between features:* This can be performed by bringing the features as close to each other as possible. However, the spacing design rules must be met while moving features closer to each other.

2. *By reducing the size of each feature:* The size rule must be met while resizing the features.

3. *By reshaping the features:* Electrical characteristics must be preserved while reshaping the feature.

Compaction tools are sometimes used as a layout aid. That is, layout is drawn in larger than minimum area. This reduces the design time. Compaction is then used to get a close to minimum area layout.

Compaction is a very complex phase in physical design cycle. It requires understanding of many details of the fabrication process such as the design rules. Compaction is very critical for full-custom layouts, especially for high performance designs. In this chapter, we discuss the compaction phase of physical design cycle.

12.1 Problem Formulation

The layout of a VLSI circuit consists of geometric feature (mostly of rectangular shape). Each feature belongs to a circuit component or to a wire. The compaction problem can be stated as: Given a set of geometric features $M = \{M_1, M_2, \ldots, M_n\}$ representing a layout. Each feature, M_i, has a minimumsize, $s(M_i)$, dictated by the design rules. In addition, minimum separation between features, $d(M_i, M_j)$, between M_i and M_j, for $1 \leq i, j \leq n$ is also given. The objective of compaction is to minimize the total layout area by moving features close to each other and by resizing the features such that

$$size(M_i) \geq s(M_i)$$

$$dist(M_i, M_j) \geq d(M_i, M_j)$$

where $size(M_i)$ and $dist(M_i, M_j)$ are size of M_i and distance between M_i and M_j after the compaction, where $1 \leq i, j \leq n$. If the sizes of the features are assumed to be fixed, then the problem is just to move the features closer to reduce the layout area.

12.1.1 Design Style Specific Compaction Problem

The scope and impact of compaction on layouts differs depending on the design style.

- Full-custom design style: Compaction is very critical in full-custom design style. After placement and routing, a large amount of space is left vacant. The problem is exactly same as the one formulated above. This is not true if significant part of layout is done by hand.

- Standard cell design style: The cell heights are fixed in a standard cell design. So the height of the layout can be minimized by minimizing channel height. Thus a restricted type of compaction, called channel compaction may be used. However, several channel routers produce very compact routings which cannot be compacted any further.

- Gate array design style: Since the position of gates is fixed, compaction is not applicable to gate array designs, except to optimize wiring.

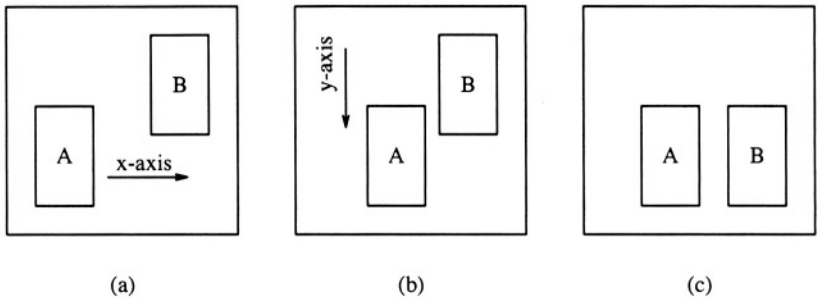

Figure 12.1: 1-D compaction.

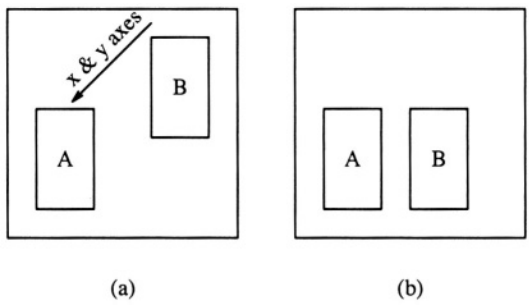

Figure 12.2: 2-D compaction.

12.2 Classification of Compaction Algorithms

Compaction algorithms can be classified in two different ways. The first classification scheme is based on the direction of movements of the components (features): *one-dimensional* (1-D) and *two-dimensional* (2-D). In 1-D compaction, components are moved only in x- or y-direction. As a result, either x- or y-coordinate of the components is changed due to the compaction. If the compaction is done along x-direction then it is called *x-compaction.* Similarly, if the compaction is done along the y-direction, then it is called *y-compaction.* Figure 12.1 shows an example of both x- and y-compactions. In 2-D compaction, the components can be moved in both x- and y-direction simultaneously. As a result, in 2-D compaction, both x- and y-coordinates of the components are changed at the same time in order to minimize the layout area. Figure 12.2 gives an example of a 2-D compaction.

The second approach to classify the compaction algorithms is based on the technique for computing the minimum distance between features. In this approach we have two methods, *constraint-graph based compaction* and *virtual grid based compaction.* In constraint-graph method, the connections and sep-

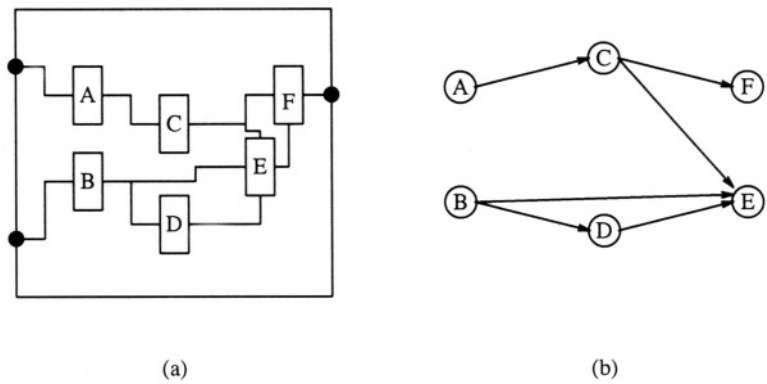

Figure 12.3: Constraint graph generation.

arations rules are described using linear inequalities which can be modeled using a weighted directed graph (constraint graph) as shown in Figure 12.3. This constraint graph is used to compute the new positions for the components.

On the other hand, virtual grid method assumes the layout is to be drawn on a grid. Each component is considered attached to a grid line. The compaction operation compresses the grid along with all components placed on it keeping the grid lines straight along the way. The minimum distance between two adjacent grid-lines depends on the components on these grid lines. The advantage of virtual grid method is that the algorithms are simple and can be easily implemented. However, virtual grid method does not produce compact layouts as compared to the constraint graph method.

In addition, compaction algorithms can also be classified on the basis of the hierarchy of the circuit. If compaction is applied to different levels of the layout, it is called *hierarchical compaction.* Any of the above mentioned methods can be extended to hierarchical compaction. A variety of hierarchical compaction algorithms have been proposed for both constraint-graph and virtual grid method. Some compaction algorithms actually 'flatten the layout' by removing all hierarchy and then perform compaction. In this case, it may not be possible to reconstruct the hierarchy, which may be undesirable.

12.3 One-Dimensional Compaction

In this section, we present two methods of one-dimensional compaction: Constraint graph based compaction and virtual grid compaction. One dimensional compactors are repeatedly used in X and Y directions until no further compaction is possible.

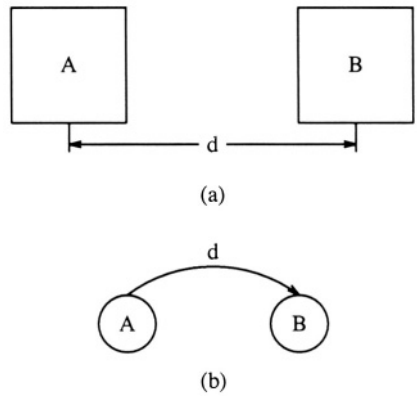

Figure 12.4: Separation constraint.

12.3.1 Constraint-Graph Based Compaction

The constraint graph $G = (V, E)$, is a weighted graph. Each vertex $v \in V$ represents a component while edges represent constraints. There are two types of constraints that should be satisfied in the process of compaction: separation constraints and physical connectivity constraints. Both separation constraints and physical connectivity constraints can be incorporated into a graph representing a 1-D compaction problem. A separation constraint between two features can be represented using a weighted directed edge between the two vertices, with weight equal to the minimum separation. For example, if the two features A and B are required to be at least d units apart from each other; assuming that A is to the left of B, this rule can be written as $B_x \geq A_x + d$ where A_x refers to the x-location of component A. The inequality is represented in the graph as an edge from A to B of weight d (see Figure 12.4). Figure 12.5 shows the connectivity constraints that require two components to be within a distance s of each other. A physical connection can be represented as a cycle of two edges. The condition $|C_x - W_x| \leq s$ can be rewritten as two constraints $W_{1x} \geq C_x - s$ and $C_x \geq W_x - s$, which appear in the graph as a pair of constraints between C_x and W_x, each with weight $-s$.

The constraint graph includes two additional vertices, L and R, which represent two physical boundaries (without the loss of generality, left and right). L can be thought of as a source of the constraint graph, because all other vertices are, explicitly or implicitly, required to the right of L. Similarly, R can be considered as the sink of the graph. Figure 12.6 gives an example of a constraint graph that includes source and sink vertices.

In the process of compaction when the elements are moved by the compactor it is necessary that the original electrical connections be preserved. Most compactors derive the connectivity from the overlapping regions in the original layout. In the following, we discuss two types of connectivity constraints and

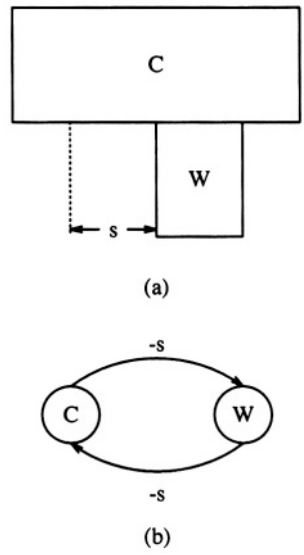

Figure 12.5: Connectivity constraint.

how can they be included in the constraint graph.

1. **Wire-terminal connections:** Connectivity constraints for wire-terminal connections with horizontal wire, vertical wire, and wide vertical wire are illustrated in Figure 12.7(a), (b), and (c) respectively. These rules ensure good electrical connections (See Figure 12.8(a)) when compared with the minimum overlap rules (See Figure 12.8(b)).

2. **Wire-wire constraints:** The connection between two wires is also captured into the graph in a similar way as it is captured for the wire-terminal connections. Figure 12.9 shows good wire-wire connections in which wires overlap by their complete width.

After the layout is compacted, a number of vertices could still be relatively free to move. Therefore, other objectives can be used to determine coordinates of these vertices, e.g., minimize the total length of the interconnect wires located on specified layers to reduce resistance and capacitance [Sch83].

The *longest path* algorithm can be used to assign positions to the vertices that minimizes the distance from the source to the sink, which is equivalent to minimizing the layout width in the dimension of compaction.

12.3.1.1 Constraint Graph Generation

As discussed earlier, once the constraint graph is generated, the actual compaction is quite simple using the longest path algorithm. However, the first

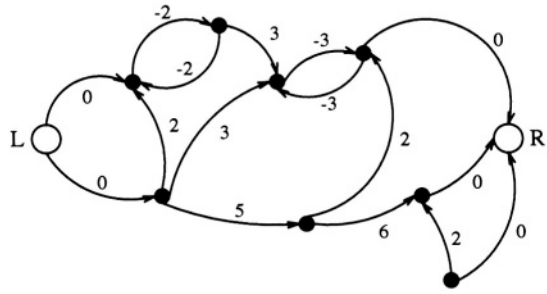

Figure 12.6: An example of a constraint graph.

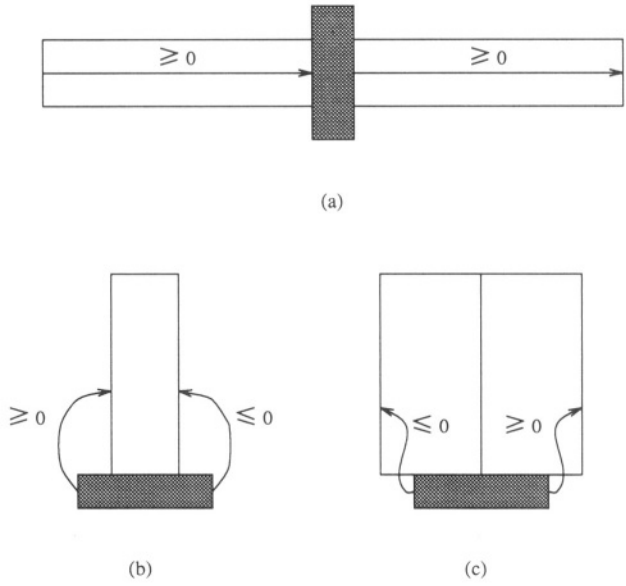

Figure 12.7: Wire-terminal connection constraints.

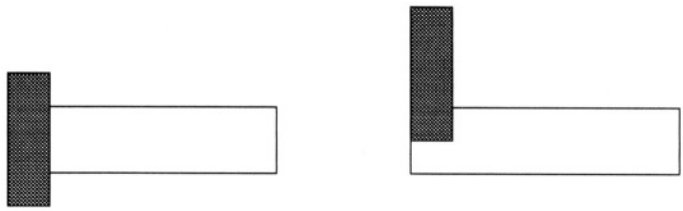

Figure 12.8: Connections after compaction.

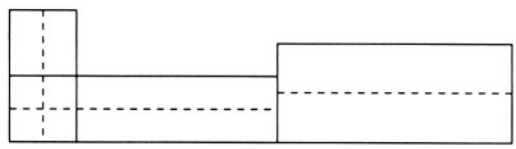

Figure 12.9: Wire-wire connections.

step necessary in constraint graph compaction is to build the constraint graph. The building of the constraint graph is the most time-consuming part of constraint graph based compaction and is $O(n^2)$ in the worst case. This is due to the fact that, in the worst case, there is an edge between every pair of vertices in the constraint graph. Only a small subset of the potential edges are actually needed for constraint graph compaction. A circuit component group typically will only have spacing requirements with its nearest neighbors. Many techniques for generating the constraint graph efficiently have been proposed [HP79, Mal87]. In construction of a constraint graph, the connectivity constraints are generated first. The connectivity constraints can be generated by scanning all legal connections in the symbolic layout. The connectivity information is usually stored in a table and the compactor looks up the table to generate all the constraints. Different types of connectivity constraints include wire-wire, wire-via, wire-source connectivity constraints.

Separation constraints are generated once per compaction step. The constraint generation method used should ideally generate a non-redundant set of constraints since the cost of solving the constraint graph is proportional to the number of edges in the graph, or the number of constraints. Several constraint generation algorithms have been proposed. In the following section, some of the constraint generation algorithms will be discussed.

1. **Shadow-Propagation Algorithm:** A widely used and one of the best known techniques for generating a constraint graph is the *shadow-propagation* used in CABBAGE system [HP79]. The 'shadow' of a feature

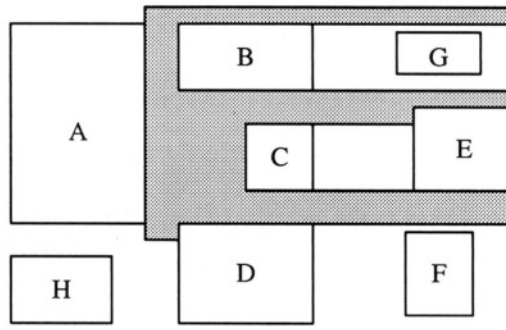

Figure 12.10: Example of shadow propagation.

is propagated along the direction of compaction. The shadow is caused by shining an imaginary light from behind the feature under consideration (see Figure 12.10). Usually the shadow of the feature is extended in both sides of the features in order to account for diagonal constraints. This leads to greater than minimal Euclidean spacings since an enlarged rectangle is used to account for corner interactions. (See shadow of feature in Figure 12.10).

Whenever the shadow is obstructed by another feature, an edge is added to the graph between the vertices corresponding to the propagating feature and the obstructing feature. The obstructed part of the shadow is then removed from the front and no longer propagated. The process is continued until all of the shadow has been obstructed. This process is repeated for each feature in the layout. The algorithm SHADOW-PROPAGATION, given in Figure 12.11, presents an overview of the algorithm for x-compaction of a single feature from left to right.

The SHADOW-PROPOGATION routine accepts the list of components (*Comp_list*), which is sorted on the x-coordinates of the left corner of the components and the component (*component*) for which the constraints are to be generated. The procedure, INITIALIZE-SCANLINE, computes the total length of the interval in which the shadow is to be generated. This length includes the design rule separation distance. The y-coordinate of the top and the bottom of this interval are stored in the global variables, *top* and *bottom* respectively The procedure, GET-NXT-COMP, returns the next component (*curr_comp*) from *Comp_list.* This component is then removed from the list. Procedure LEFT-EDGE returns the vertical interval of component, *curr_comp.* If this interval is within the *top* and *bottom* then *curr_comp* can possibly have a constraint with *component.* This check is performed by the procedure *IN – RANGE.* If the interval for *curr_comp* lies within *top* and *bottom* and if this interval is not already contained within one of the intervals in

```
Algorithm SHADOW-PROPAGATION(Comp_list, component)
begin
    INITIALIZE-SCANLINE( component );
    I = φ;
    while( (LENGTH-SCANLINE( I ) < (top − bottom))
            and ( Comp_list ≠ φ) )
        curr_comp = GET-NXT-COMP( Comp_list );
        I_i = LEFT-EDGE( curr_comp );
        if( IN-RANGE( I_i, top, bottom ) )
            I' = UNION( I_i, I );
            if( I' ≠ I )
                ADD-CONSTRAINT( component, curr_comp );
                I = UNION( I_i, I );
end.
```

Figure 12.11: Shadow-propagation algorithm

the interval set, $\mathcal{I}$, then the component lies in the shadow of *component* and hence a constraint has to be generated. Each interval represents the edge at which the shadow is blocked by a component. The constraint is added to the constraint graph by the procedure ADD-CONSTRAINT. The procedure UNION inserts the interval corresponding to *curr-comp* in the interval set at the appropriate position. This process is carried out till the interval set completely cover the interval from *top* to *bottom* or there are no more components in *Comp_list.* The Figure 12.12(a) shows the layout of components. The constraint for component *A* with other components is being generated. Figure 12.12(b) shows the intervals in the interval set as the shadow is propagated. From Figure 12.12(b) it is clear that the constraints will be generated between components *A* and components *B*, C, and *D* in that order. As component *F* lies outside the interval defined by *top* and *bottom* it is not considered for constraint generation. The interval generated by component *E* lies within one of the intervals in the interval set. Hence, there is no constraint generated between components *A* and *E.*

2. **Scanline Algorithm:** In [Mal87], Malik presented an efficient algorithm based on scanline method. The *scanline* is an imaginary horizontal (or vertical) line that cuts through the layout in *x*-compaction (or *y*-compaction). An example of scanline is shown in Figure 12.13. The scanline data structure contains all the rectangles that are cut by this scanline. The rectangles are stored in non-decreasing order of their *x*-coordinates of the left boundaries. The scanline traverses from the top to the bottom of the layout for *x*-compaction. Similarly, for *y*-compaction, the scanline traverses from the left to the right of the layout. For *x*-compaction, as the

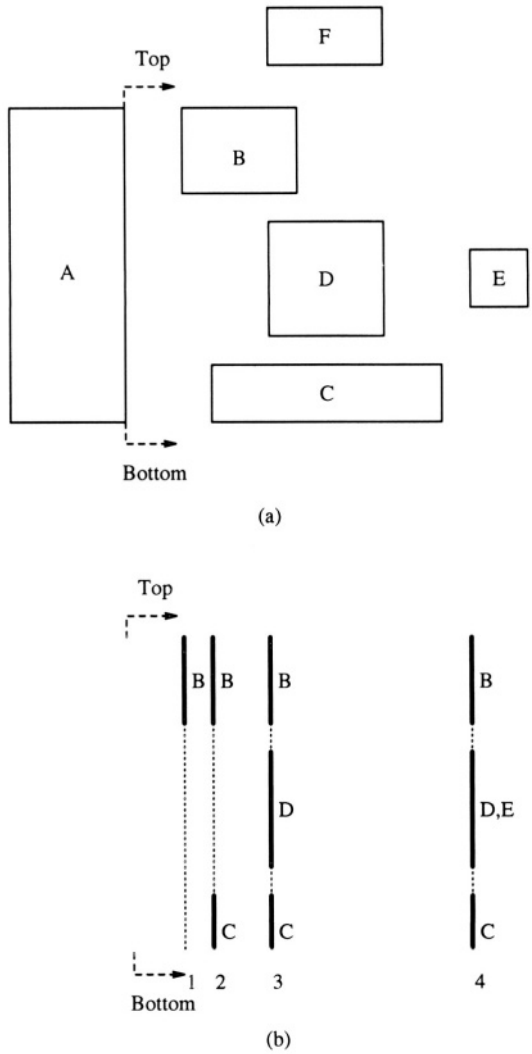

Figure 12.12: Interval generation for shadow propagation.

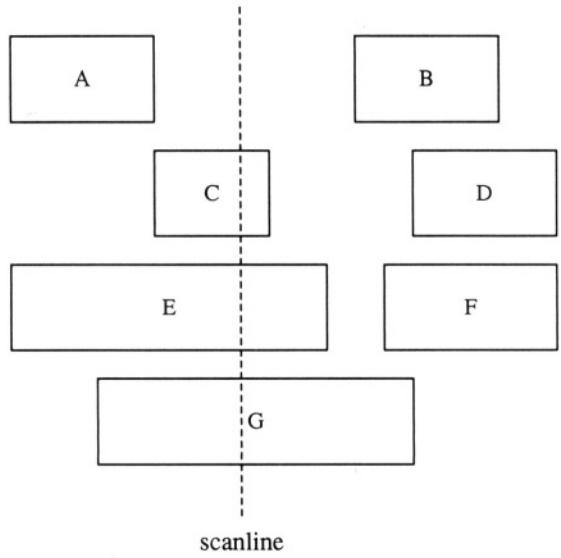

Figure 12.13: Example of a scanline.

line passes over the top edge of a rectangle, the rectangle is added to the scanline. Similarly, when the scanline passes over the bottom edge, the rectangle is deleted from the scanline. Two lists are required to move the scanline over the layout. One list contains rectangles in non-decreasing order of their YTOP and one list contains rectangles in non-decreasing order of their YBOTTOM. Let us denote the two sorted list as *topsorted* and *bottomsorted,* respectively. The algorithm SCANLINE is shown in Figure 12.14.

Note that the algorithm SCANLINE adds many redundant edges. A redundant edge is the one that does not affect the longest path from the source vertex to any other vertex in the graph. Hence the removal of the redundant edge will not change the constraint graph. Consider the example shown in Figure 12.15. If the distance between R_1 and R_i (E_{1i}) is less than the the distance between R_1 and R_2 (E_{12}) plus the distance between R_2 and R_i (E_{2i}), then the constraint between R_1 and R_i is redundant thus can be removed. The scanline algorithm uses these measures to remove redundant constraints.

12.3.1.2 Critical Path Analysis

After generation of constraint graph, the next step in constraint graph compaction is to determine the critical path through the graph. Let us explain the role of critical paths in compaction. The goal of one dimensional compaction is to generate a minimum width layout. The determination of minimum width

```
Algorithm SCANLINE (G)
begin
    while topsorted ≠ φ do
        let R1 be the first rectangle in topsorted and R2 be
            the first rectangle in bottomsorted;
        if YTOP(R1) ≤ YBOTTOM(R2) then
            INSERT(R1, scanline);
            topsorted = topsorted − {R1};
            for each rectangle Ri ∈ scanline that is at the
                    left of R1 do
                ADD-CONSTRAINT(Ri, R1, G);
            for each rectangle Rj ∈ scanline that is at the
                    right of R1 do
                ADD-CONSTRAINT(R1, Rj, G);
end.
```

Figure 12.14: Scanline algorithm.

layout translates into a longest path problem. The longest path from source to a vertex is then the coordinate of the vertex. The longest path problem can be viewed as a shortest path problem by inverting the signs on the edge weights. As a result, this problem is also called the critical path problem. The edges that determine the minimum distance between the source and the sink form the *critical path* and vertices on the critical path are said to be *critical.* Tarjan [Tar83] describes a variety of algorithms to solve longest path problems; many others, including Lengauer [Len84], Liao and Wong [LW83] describe the application of various longest path algorithms to compaction. These algorithms calculate for all the vertices coordinates that are as small as possible. The worst case complexity of these algorithms is $O(|V| \times |E|)$, where V is the set of vertices in the graph and E is the set of edges. In [LW83], the complexity has been reduced to L iterations, while the run time of each iteration is $O(E)$. Where L is the number of negative weighted edges. If the constraint graph has special properties, more efficient algorithm can be used. In particular, for acyclic graphs, the worst case complexity is $O(|V| + |E|)$.

In practical layouts, the run times are almost linear in the number of layout elements. This is due to locality of the graph, that is, most edges represent very local constraints in the layout. In addition the number of edges that start at a vertex is usually quite small.

The algorithms described above can be improved by using a divide and conquer approach. The basic idea is to divide the graph in smaller subgraphs, which can be solved independently. A strong component is a subgraph in which there is a path from every node to every other node. Strong components are formed by the connectivity constraints; all the features in a strong component must move more or less together during compaction. If each strong component

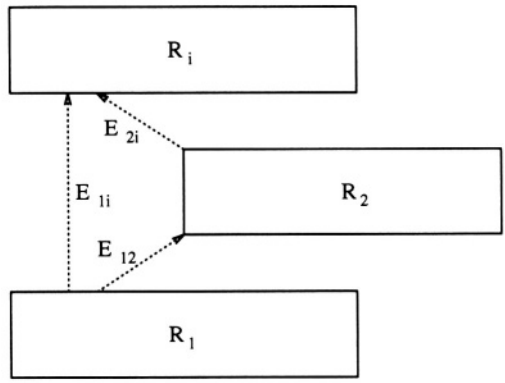

Figure 12.15: Example of redundant edge.

is reduced to a vertex, then the resulting graph is acyclic. In addition, the strong components can be assigned an ordering, which allows us to compute the effect of one strong component on the next one. The source and sink are modelled as separate strong components. Algorithms for finding strong components are described by Even [Eve79]; the best algorithm has a worst case time complexity of $O(|V| + |E|)$. The number of strong components and the number of vertices in strong components depend on the graph. In practical layout designs, the strong components are rather small.

Another method of improving the critical path algorithms is by reducing the total number of vertices and (or) edges of the graph. This reduction should not change the solution space of the constraint graph, which means that that all possible solutions that can be obtained directly or with reduction must be the same. The vertices can be reduced by grouping all the vertices which must have same relative positions. The edge reduction can be achieved by eliminating redundant edges. An edge is redundant, if there exists a path between the two vertices of the edge that does not contain the edge and which is longer (or has the same length) as the weight of the edge.

There may be vertices in the graph that are not critical and therefore have a range of legal positions. These vertices are said to have slack. Some secondary criterion must be used to assign unique positions to these vertices; one common objective is the minimization of total wire length in the cell. The compactor can place vertices with slack in such a way to increase circuit performance, to minimize wire length, to optimize fabrication yield, etc. Schiele [Sch83] and Eichenberger [Eic86] discuss algorithms that can be used to minimize wire length. Wire-length minimization significantly reduces the values of parasitic features associated with wires [Sch85].

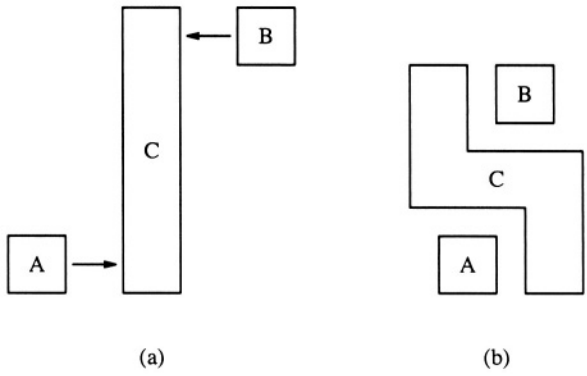

Figure 12.16: Example of wire jogging.

12.3.1.3 Wire Jogging

Both automatic jogging of wires and wire length minimization have received much attention in the last couple of years. This is, in part, due to the recent interest in channel compaction [Deu85]. One of the first approaches to jogging wires was reported by Hsueh [HP79]. In this approach jogs in wires were introduced at 'torque' points on a wire (see Figure 12.16). Wire jogging had limited success because it could reduce the size in one direction while, potentially, increasing the size in the other direction.

12.3.1.4 Wire Length Minimization

Features not on the critical path will typically find themselves pulled towards a layout edge because they are given their minimal legal spacing. This tends to increase wire length and reduce circuit performance.

One of the first methods used to reduce wire length was the 'average slack' method of Hsueh [HP79]. This approach uniformly distributes the empty space in a circuit among the features that were not on the critical path. Burns [BN86] presents a force-based heuristic that not only considers the effect of each wire layer but it also considers the cumulative effect of multiple wires connecting adjoining modules. This is effective in minimizing wires in an hierarchical layout.

12.3.2 Virtual Grid Based Compaction

The virtual grid compaction is a structured approach to compacting layout. The virtual grid is used to establish the relative placement of circuit features and does not correspond to physical grid. In this approach, the compactor gives locations to the virtual grid lines, not to the circuit component themselves.

Several compactors have been designed using virtual grid approach. Following are the two widely used algorithms based on this approach.

12.3.2.1 Basic Virtual Grid Algorithm

In this method, each component is attached to a grid line. Consider the example shown in Figure 12.17(a). Components A, B, and C are attached to the first grid line, while D, E, and F are attached to the second grid line. In the second step, the maximum necessary distance between any two grid lines is computed. In our example, the distance between C and F is required to be 14. In other words, the grid lines can be at distance 14 to each other without violating any design rules. In Figure 12.17(b), we show the compacted layout. This process is repeated for all adjacent grid lines. X-Compaction is usually followed by Y-Compaction. The basic advantage of virtual grid method is it is fast.

12.3.2.2 Split Grid Compaction

In [Boy87], Boyer introduced *split grid compactor which* places distinct circuit features that fall on the same virtual grid separately, splitting the virtual grids where necessary. Split grid compactor uses a data structure that allows only to store the grid points that are of interest. The grid points which contain features are the ones added to the data structure. This allows fast access to the circuit features.

Initially, the compactor identifies groups of circuit features falling on the same virtual grid lines that need to be placed together. Local connectivity is used to identify the groups. For example, consider a vertical virtual grid line. There are two situations that might occur: features are connected by a vertical wire segment or the features are connected by a vertical transistor. In any case, the features are grouped together. Groups are identified by traversing each virtual grid line. After the circuit features are grouped together, the compaction is done in two passes: first the x-compaction and then y-compaction. A group is first compacted by determining the spacing necessary for each component in the the group. Then each group is placed independently. Features are spaced with respect to the features in the neighboring groups. Consider the example shown in Figure 12.18(a). We group A, B, and C together on the first grid line. On the second grid we form two groups. The first group consists of D and E, while the second just consists of F. Figure 12.18(b) shows the solution after compaction.

Compression-Ridge Method: Compression-ridge method was first suggested by Akers, Geyer, and Roberts [AGR70]. In this method, vertical and horizontal regions of empty spaces, called *compression ridges* are formed. They have the following properties:

1. Compression ridge is a constant width band of empty space stretching from one side of the layout to the other side.

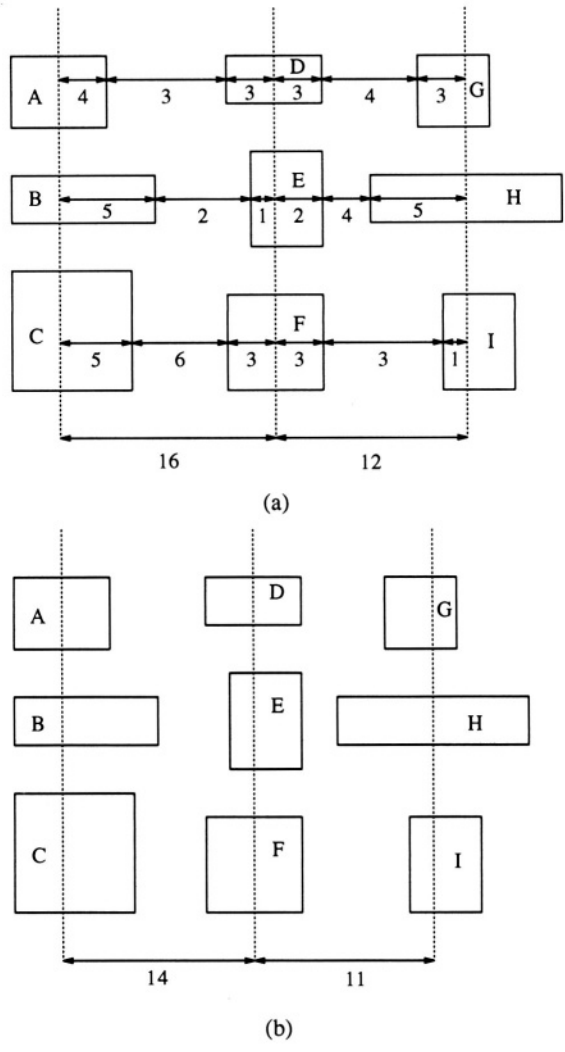

Figure 12.17: Virtual grid symbolic inverter.

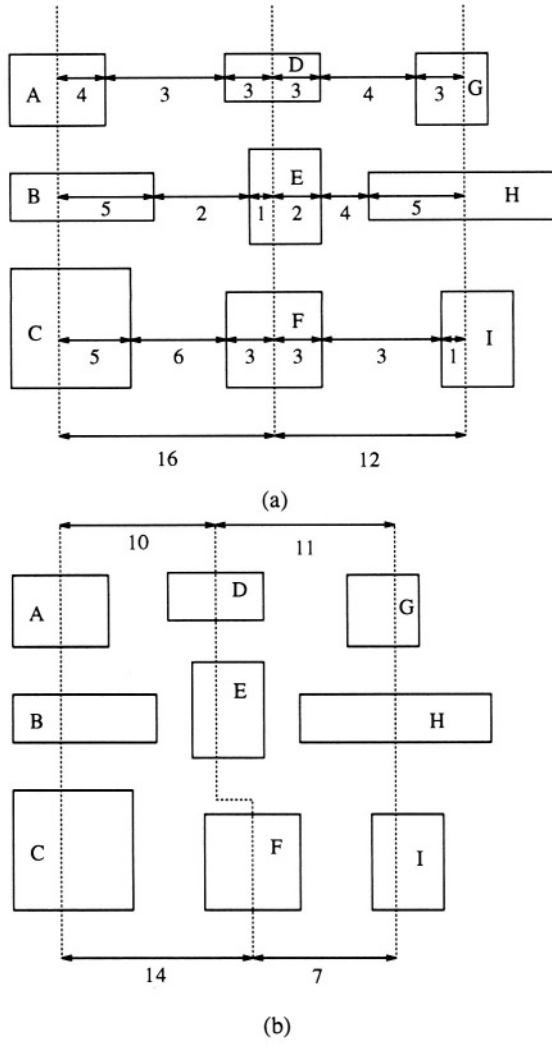

Figure 12.18: Split grid compaction.

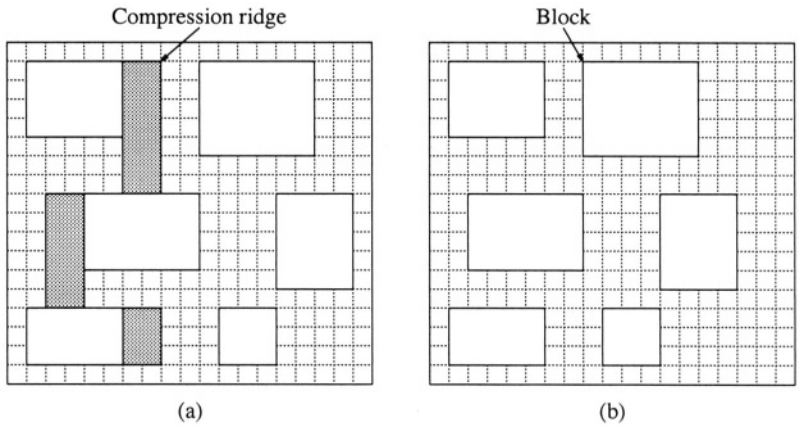

Figure 12.19: Compression ridge.

2. Compression ridge can intersect wires that are perpendicular to it. However, it cannot intersect those wires that are parallel to it.

3. The width of a compression ridge is such that when the space is removed, no design rules should be violated in the resulting layout.

The example of compression ridge is shown in Figure 12.19. The compression ridge is shown by the shaded region in Figure 12.19(a). In this technique, these compression ridges are subsequently removed from the layout until no more ridges are found. One method for finding the compression ridges is to use the virtual split-grid method. The vacant space along the grid line can be replaced by a rectangular compression ridge. The width of this ridge is the minimum compression that can be achieved along the grid line. Figure 12.19(b) shows the layout after removing the compression ridge. Note that compression ridges are formed from one end of the layout to the other which in the worst case is very time consuming. Therefore, an efficient algorithm is required to find the compression ridges. Dai and Kuh [DK87b] proposed an $O(n \log n)$ algorithm for finding compression ridges that allows the largest decrease in layout width. One of the main advantages of the compression ridge method is that the compaction can be broken into smaller steps.

12.3.2.3 Most Recent Layer Algorithm

In [BW83], Boyer and Weste presented a virtual grid compaction algorithm called *most recent layer algorithm.* The algorithm consists of two different passes: first in the x-direction, then in the y-direction. The diagonal checks are done during the y-compaction. The algorithm does not require any backtracking and the time complexity of this algorithm is $O(n)$, where n is the total number of features in the layout.

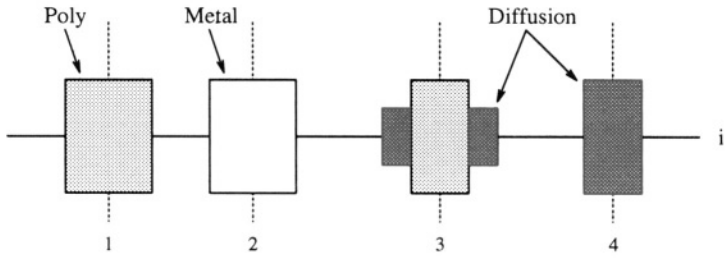

Figure 12.20: Most recent layer compaction.

For the x-compaction, each horizontal grid line has a set of reference lines ('pickets'), one for each layer, to keep track of the right edge of the most recent placement of a mask feature on that grid line. Initially, a column is placed as close to the picket as possible (without violating the design rules). The pickets are then updated. If there is no feature in a layer for a certain column, the picket position for that layer is not updated (it remains unchanged). To position a mask feature in a layer, the left edge of the feature is used to determine the necessary location of that layer with respect to picket. The right edge of the feature is used to update the pickets. For an illustration, consider the example shown in Figure 12.20. There are three different pickets for three layers, one for metal, one for poly, and one for diffusion. Consider the x-compaction in horizontal virtual grid line i. We assume that features in columns 1, 2, 3 are already placed and the pickets are updated. Now, the feature of the diffusion layer has to be placed in column 4. Only the picket of diffusion layer will be used to find the location of this feature. After placing the feature in its minimal distance position, the picket in diffusion layer is updated to the coordinate of the right side boundary of the feature.

The y-compaction is done in similar way. However, additional information is necessary in order to handle the diagonal constraints. The left and right edges, as well as the upper edges, of the mask features must be recorded in order to do the diagonal checking.

12.4 $1\frac{1}{2}$-Dimensional Compaction

In [SSVS86], Shin, Sangiovanni-Vincentelli, and Sequin presented a new compactor based on simulation of zone refining process. Although compactor is based on simulation of an engineering process, it is a deterministic algorithm and differs sharply from other simulation based approaches such as simulated annealing and simulated evolution. The key idea is to provide enough lateral movements to blocks during compaction to resolve interferences. In that sense, this compactor can be considered $1\frac{1}{2}$-dimensional compactor, since the geometry is not as free as in true 2-dimensional compaction.

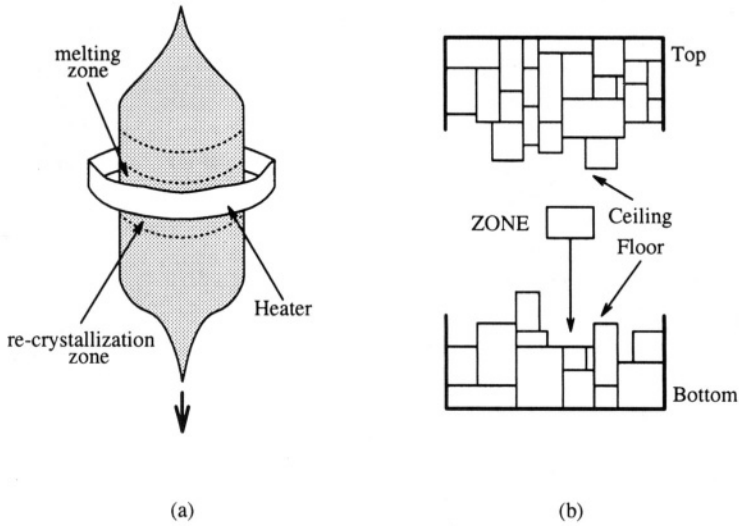

Figure 12.21: Zone refining.

The process of zone refining is used to purify crystal ingots. The basic idea is to allow limited melting of the crystal and let purities drain out of the crystal. The zone refining process starts with an already developed 'impure' crystal. As shown in Figure 12.21 (a), the crystal is slowly pulled through a heating element to locally heat the crystal to melting temperature. At the exit end of the heater, the material re-crystallizes. Since impurities are built into the crystal lattice at a much slower rate than the crystal material, impurities have tendency of being left out during re-crystallization process. That is, the impurities are left in the molten state in the heated zone. Eventually, impurities are drained out of one end of the crystal.

In terms of layout compaction, the algorithm starts with a layout. The vacant space in the layout is considered the impurity. Starting from one side, blocks are considered row by row and are re-arranged after they have been moved across the open zone. During its movement in the free zone, the blocks travel through the entire width of the layout and hence may be placed anywhere along the boundary. In Figure 12.21(b), we show the process of compaction by zone refining.

The algorithm maintains an XY adjacency graph. In an XY adjacency graph, vertices represent blocks, while edges represent horizontal and vertical adjacency. That is, two blocks have a horizontal edge if they share a vertical boundary. Similarly, two blocks have a vertical edge if they share a horizontal boundary. The labels on the edges represent the minimum allowable distance between blocks. Four additional vertices are added to keep all the blocks within the required bounded rectangle. Note that free space is ignored in computing

the neighborhood edges between blocks. Figure 12.22(a) an instance of problem along with its XY adjacency graph in Figure 12.22(b).

Algorithm assumes that the input is partially compacted layout, which can be obtained by two applications of a 1-D compactor. It maintains two lists called *floor* and *ceiling*. Floor consists of all the blocks which are visible from the top and may become a neighbor of future block. Ceiling is a list of all blocks which can be moved immediately. That is, ceiling is the list of blocks visible from the bottom. The algorithm selects the lowest block in the ceiling list and moves it to the place on the floor, which maximizes the gap between floor and ceiling. This process is continued until all blocks are moved from ceiling to floor.

Let us illustrate the algorithm with an example in Figure 12.22. Since C is the lowest block in the ceiling list, it is selected for the move. Figure 12.22(c) shows that the gap is maximum at the boundary between blocks A and B. Therefore C is moved between and A and B. The modified layout and the XY-adjacency graph are shown in Figures 12.22(d) and (e) respectively.

12.5 Two-Dimensional Compaction

Recall that there are three different types of constraints imposed by the design rules that must be satisfied to obtain a valid layout. These constraints are size constraints, overlap constraints, and separation constraints. Furthermore, designer can impose extra constraints known as *user defined constraints.* Given a symbolic layout consisting of rectangular features, all these constraints can be written using linear constraint equations. Let us assume that for each block B_i, two coordinates, (x_i, y_i) and (x_i', y_i'), are given for the lower left corner and the upper right corner, respectively. Let a block B_i has height h_i and width w_i, then the size constraints can be written as:

$$x_i + w_i \leq x_i' \qquad y_i + h_i \leq y_i'$$

$$x_i' - w_i \leq x_i \qquad y_i' - h_i \leq y_i$$

Wires have a fixed width but variable length. A vertical wire W_j with width w_j is specified by the constraints:

$$x_j + w_j \leq x_j' \qquad y_j \leq y_j' \qquad x_j' + w_j \leq x_j$$

The constraints for the horizontal wires can be given in the similar way.

In the course of compaction, the algorithm must maintain appropriate connections between blocks and wires and between wires. Wires on the boundary of a block can slide along the boundary within the range specified by the user, provided that no constraints are violated. The overlap constraints between a block B_i and a wire segment W_j can be given as:

$$x_j \leq x_i' \qquad x_i' \leq x_j' \qquad x_i \leq x_j$$

$$y_i + r_{ij}^1 \leq y_j \qquad y_j' + r_{ij}^2 \leq y_i'$$

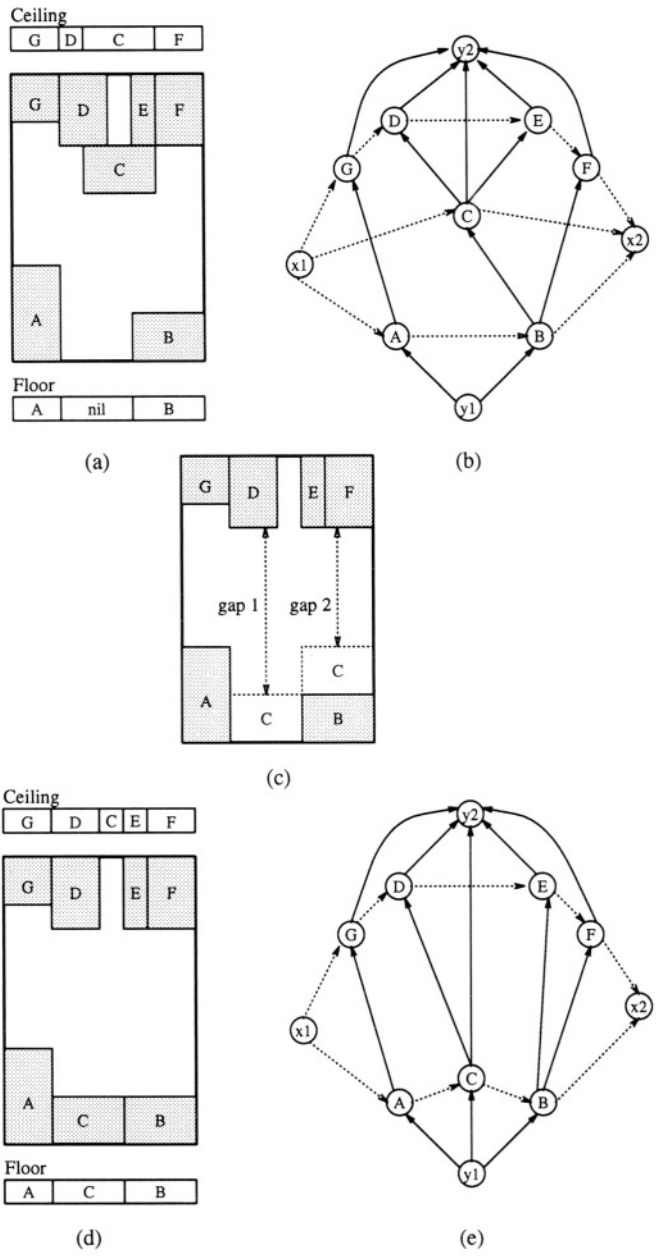

Figure 12.22: Problem instance and its XY adjacency graph.

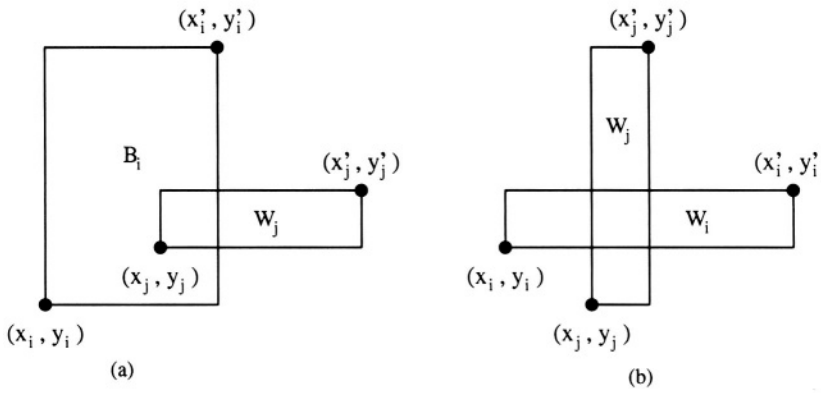

Figure 12.23: Overlap constraints.

Note that W_j is connected to the right boundary of B_i and the range is $[r_{ij}^1, r_{ij}^2]$ (Figure 12.23(a)). The overlap between two wires W_i and W_j, as shown in Figure 12.23(b), can be specified as:

$$x_i \leq x_j \qquad x'_j \leq x'_i$$

$$y_j \leq y_i \qquad y'_i \leq y'_j$$

Next the separation constraints can be specified as minimum distance constraints between two non-overlapping features. If blocks B_i and B_j are not supposed to overlap, they must be separated by a certain distance. There are four possible cases: j is on the right of i at a distance of at least d_{ij}^1, j is on the left of i at a distance of at least d_{ij}^2, j is on the top of i at a distance of at least d_{ij}^3, and j is below i at a distance of at least d_{ij}^4. Thus, one of the following must be satisfied.

$$c_{ij}^1 : x'_i + d_{ij}^1 \leq x_j \qquad c_{ij}^2 : x'_j + d_{ij}^2 \leq x_i$$

$$c_{ij}^3 : y'_i + d_{ij}^3 \leq y_j \qquad c_{ij}^4 : y'_j + d_{ij}^4 \leq y_i$$

Let $C_{ij} = \{c_{ij}^1, c_{ij}^2, c_{ij}^3, c_{ij}^4\}$ and $D = \{C_{ij} | \ i$ and j are non-overlapping features $\}$. Thus to ensure that the design rules are satisfied, one of the four constraints in C_{ij} must be satisfied.

Therefore, the constraints can be divided into two classes:

1. set of constraints, B, that must be satisfied, which include size, overlap and user defined constraints.

2. set of constraints, D, that are divided into groups and at least one of the constraints in each group must be satisfied.

After the generation of constraints, the problem can be solved using integer linear programming technique. However, the complexity of the linear programming technique is exponential thereby making it impractical even for a moderate size problem.

In [SLW83], Schlag, Liao, and Wong showed that the 2-D compaction problem is NP-complete and gave a branch-and-bound solution for the problem. However, again the complexity of the algorithm is in the worst case exponential.

12.5.1 Simulated Annealing based Algorithm

In [HLL88], Hseih, Leong, and Liu proposed a solution to 2-D compaction using simulated annealing technique. Although this technique produces sub-optimal solution, this is much faster than branch-and-bound and integer linear programming technique. The layout can be represented by a valid set of constraints. A valid set of constraints is a subset E of constraints that contains all the constraints from B and at least one constraint from D. We use the notation $E = B \cup M$, where M contains exactly one constraint from each group of D. In the simulated annealing algorithm, given a solution $B \cup M$, a *move* is defined as selecting a group in D and exchanging the constraints in that group. Two solutions $B \cup M$ and $B \cup M'$ are said to be neighbors if M' can be obtained from M by interchanging the chosen constraint in one of the groups of D. Clearly, it is possible to go from one given solution to another by a sequence of moves.

12.6 Hierarchical Compaction

The compactors discussed in the previous sections, perform compaction on layouts composed from a library of pre-defined features and wire segments. Since the characteristics of the layout primitives other than their basic shapes are typically exploited to generate compact layouts, it can be difficult to use such system for hierarchical design.

Hierarchical compaction can be used for hierarchical designs to reduce the space and computation time of the layout compaction. In the hierarchical compaction, transistors, contacts, and modules are treated in the same manner. In this section, we discuss one hierarchical compaction algorithm based on constraint-graph generation.

12.6.1 Constraint-Graph Based Hierarchical Compaction

Given a hierarchical symbolic layout, hierarchical constraint graph is generated at each level of the hierarchy of the design from bottom up. Initially, constraints are generated for all the leaf cells consisting of basic features. Each leaf cell is compacted using the corresponding constraint graph and the boundary of the compacted leaf cell is fixed. Once the leaf cell is compacted and the boundary is fixed, the cell can be treated as a single cell in the next level in the hierarchy and constraints can be generated for the cells in that level. The

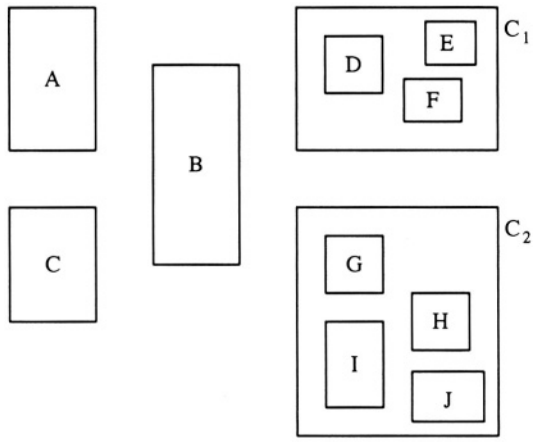

Figure 12.24: Hierarchical layout.

compaction is carried out by generating constraints at each level. For an illustration, consider the example of a hierarchical design shown in Figure 12.24. The layout consists of two levels of hierarchy. In the leaf level, cells C_1 and C_2 are compacted by generating their constraint graphs G_1 and G_2 as shown in Figure 12.25. Once C_1 and C_2 are compacted, their layout is represented by a vertex in the next level of the constraint generation as shown in the graph G_3 in Figure 12.25.

The hierarchy of the design will be preserved if at each level, the boundary of the compacted cells are kept rectangular. However, keeping the boundary rectangular does not produce a good solution. To get better results, the boundary of the compacted cell at any level can be given any arbitrary rectilinear shape and the constraint graph may be allowed to have multiple constraint edges between two vertices, specifying different separation constraints. However, this approach does not preserve the hierarchy of the layout.

12.7 Recent trends in compaction

In this section, two new trends in compaction are briefly reviewed. These include performance-driven compaction and compaction techniques for yield enhancement.

12.7.1 Performance-driven compaction

In [OCK95], authors use an iterative parameterized LP formulation to model a force-directed wire respacing scheme for compaction under timing constraints and compaction under peak crosstalk constraints. Although it uses a distributed delay model that factors in the coupling capacitances of the nets,

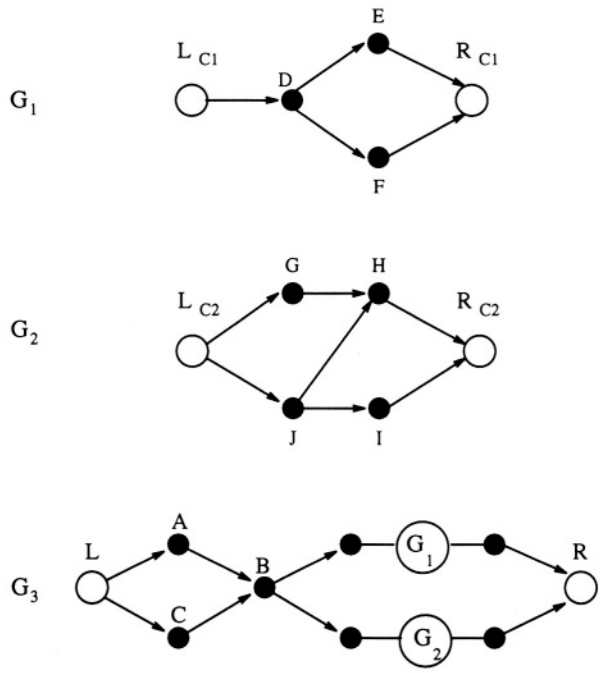

Figure 12.25: Hierarchical constraint graph.

some of the intuition behind the force-directed scheme is lost in translating the two-dimensional optimization problem into a problem with a one-dimensional objective function (delay/crosstalk) and a constrained penalty parameter associated with the second objective (area).

In [WLLC93], a LP formulation is developed for timing constraints on critical paths (in addition to regular layout constraints), and then shows how to solve them efficiently using a graph-based simplex algorithm. The delay model assumes that the delay of a wire is proportional to its length. First, an LP is used to find a tight upper bound on the delays of the timing critical paths by adjusting the wire lengths. Next, another LP is used to perform one-dimensional compaction on the layout while ensure that the delay of each of the timing critical paths remains less than the upper bound determined by the first LP.

12.7.2 Compaction techniques for yield enhancement

Chiluvuri and Koren [CRK95] developed a constraint-graph based compaction strategy using empirical heuristic desired locations for objects not on critical paths in the constraint graph to make the spacing between different elements more uniform, in an effort to decrease the sensitivity of the layout to random point defects.

Bamji and Malavasi [CM96] extended the work done by Chiluvuri and Koren by using a network flow based formulation for the layout respacing. The formulation models objectives such as yield, crosstalk or wire length (or their linear combinations) using a piece-wise linear convex cost function for the edges.

12.8 Summary

Compaction is a very important phase in physical design cycle. The objective of a compaction algorithm is to reduce the layout area. Research in symbolic layout compaction has resulted in two major compaction strategies: the constraint-graph based compaction and the virtual grid based compaction. Constraint-graph based compactors usually produce smaller area layouts as compared to the virtual grid compactors, while virtual grid compactors typically run faster. The speed of both type of algorithms can be improved by using hierarchy of the layout.

12.9 Exercises

1. For an instance shown in Figure 12.10, generate constraints using scanline algorithm.

2. Develop an algorithm to remove the redundant constraints in a constraint graph.

3. Symbolic layout of a single cell may described using a unit size. Develop an algorithm to produce a full-size layout of a cell from a unit-size description by adding wires to the cell.

4. Extend the virtual grid based algorithm to include jog insertion if necessary to reduce the layout area.

5. Extend the two-dimensional compaction to include jog insertion.

†6. If two neighboring cells have to be connected, then the compaction algorithm can stretch the cells such that their terminals to be connected lie in the same x- or y-position. This process of stretching cells to align terminals is known as *pitchmatching*. Design the necessary constraints to handle the pitchmatching in a compaction algorithm.

‡7. Implement the zone refining algorithm for L-shaped blocks. Note that rotation, flipping of blocks have to be taken into consideration.

‡8. Develop constraints for L-shaped blocks. Can we represent each L-shaped block as a combination of two rectangles ? What additional constraints are needed ?

9. Consider the channel compaction problem. Channels are compacted in the y-directions. Which channel routing algorithm produces the most compactable solution ? Which channel router is easiest to adapt to integrated channel routing and compaction ?

Bibliographic Notes

The first compaction algorithm called *shear-line compaction* was proposed by Akers et al. [AGR70]. Symbolic layout and compaction were first combined in the STICKS system [Wil78]. In [BN87], a constraint generation technique for hierarchical compaction has been proposed. In [KW84], the compaction problem was formulated into a mixed integer linear programming problem of a very special form. Symbolic layout compaction with symmetric constraints was considered in [OSOT89]. The symmetric constraint maintains the geometrical symmetry of the circuit components during the compaction.

[SL89b, WLC90] present efficient two-dimensional layout compaction algorithms. In [dDWLS91], a two-dimensional topological compactor with octagonal geometry is presented.

Algorithmic Aspects of one dimensional layout compaction are discussed in [DL87a]. In [RPV+87], geometrical compaction in one dimension for channel routing is considered. In [LV90], an $O(n^{1.5} \log n)$ 1-d compaction algorithm is presented. [DL91] presents on minimal closure constraint generation for symbolic cell assembly. [CH87] explains how to generate incremental compaction spacing constraints. In [BV93], discusses a method for identifying overconstraints during hierarchical compaction. [Ono90] presents layout compaction with attractive and repulsive constraints. It has been shown in [PDL97] that the traditional problem of removing redundant constraints to yield the smallest possible constraint graph in symbolic compaction is NP-hard. However, if one is also allowed to add new constraints, the smallest possible constraint graph can be obtained in polynomial time. In [DPLL96], a global strategy for the elimination of positive cycles in overconstrained graph-based compaction problems is presented. It uses new polynomial LP-based formulations that are of independent interest and applicability by themselves. [Koc96] uses local logic resynthesis specific to the FPGA architecture being compacted to perform compaction. In [ASST97], a min cost flow based formulation of the compaction problem (wire length minimization) is presented.

In terms of parallel algorithms, [SC96] presents a parallelization of the "cut, compact and merge" approach towards full-chip compaction. In [CTC94] a parallel algorithm for integrated compaction and wire balancing on a shared memory multiprocessor has been evaluated.

[Har91, YCD+95, BV92, Mar90, DMH+93], present several schemes for hierarchical compaction and methods for dealing with large databases. In [FCMSV92], an efficient methodology for symbolic compaction of analog IC's with multiple symmetry constraints is presented.