M. Sc. Engineering Thesis

Linear Time Algorithms for Floor-Planning and Routing Problems

by

A. K. M. Azad

Submitted to

Department of Computer Science and Engineering
in partial fulfillment of the requirements for the degree of M. Sc.
Engineering (Computer Science and Engineering)

Department of Computer Science and Engineering (CSE)
Bangladesh University of Engineering and Technology (BUET)
Dhaka-1000, Bangladesh

August 27, 2005

Linear Time Algorithms for Floor-Planning and Routing Problems

A thesis submitted by

A. K. M. Azad
Student No. 9605039P

For the partial fulfillment of the degree of
M. Sc. Engineering (Computer Science and Engineering)
Examination held on August 27, 2005

Approved as to style and contents by:

[illegible]

Dr. Md. Abul Kashem Mia
Professor
Department of Computer Science and Engineering
BUET, Dhaka-1000, Bangladesh

Chairman
and
Supervisor

[illegible]

Dr. Md. Shamsul Alam
Professor and Head
Department of Computer Science and Engineering
BUET, Dhaka-1000, Bangladesh

Member
(Ex-officio)

[illegible]

Dr. Md. Saidur Rahman
Assistant Professor
Department of Computer Science and Engineering
BUET, Dhaka-1000, Bangladesh

Member

Md. Elias

Dr. Md. Elias
Associate Professor
Department of Mathematics
BUET, Dhaka-1000, Bangladesh

Member
(External)

Certificate

This is to certify that the work presented in this thesis paper is the outcome of the investigation carried out by the candidate under the supervision of Dr. Md. Abul Kashem Mia in the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka-1000. It is also declared that neither of this thesis nor any part thereof has been submitted or is being concurrently submitted anywhere else for the award of any degree or diploma.

[illegible]

Signature of the Supervisor

[illegible]

Signature of the Author

Contents

List of Figures

Acknowledgement

First and foremost, I would like to acknowledge my gratitude to Dr. Md. Abul Kashem Mia, Professor, Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology. His constant supervision, scholarly guidance, valuable advice and encouragement have been a great impetus to this thesis work. I would also like to acknowledge my gratitude to Dr. Md. Saidur Rahman, Associate Professor, Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology. His vast experience and in-depth knowledge in graph theory has helped significantly to achieve a smooth completion of this thesis work. He gives me valuable guide lines through out the thesis work.

I would like to thank the members of the graduate committee, Dr. Md. Shamsul Alam, Head and Professor, Department of Computer Science and Engineering, BUET, and Dr. Md. Elias, Associate Professor, Department of Mathematics, BUET, for their valuable suggestions.

I would like to acknowledge the all-out co-operation and services rendered by the faculty members and staffs of the CSE Department. Finally, I also express my ever gratefulness to all who contributed to this thesis work.

Abstract

The designer of an integrated circuit (IC) transforms a circuit description to a geometric description, called the VLSI layout. The task of converting the specification of an electrical circuit into a layout is called the physical design. In physical design cycle a circuit specification is converted into a VLSI layout in the four phases: partitioning, floor-planning, routing (global routing and detailed routing) and compaction. A large circuit is divided into a set of smaller blocks and the interconnections (nets) between them in the partitioning phase. Each block is then placed (floorplanning phase) in a plane so that interconnections can be routed (routing phase) through the remaining space. Most of the works that appear in the literature deal floorplanning and global routing in two different phases. Although there are some linear time floorplanning algorithms, but the known best algorithms for global routing and detailed routing run in $O(N^2)$ time, where N is the number of given nets. In this thesis, we present an integrated algorithm based on orthogonal drawing of plane graph that handle floorplanning and global routing in a single phase. The time complexity of our algorithm is linear. We also develop a linear time detailed routing algorithm.

Chapter 1

Introduction

Integrated Circuit (*IC*) has transformed our society into an information society. This revolutionary development and widespread use of ICs has been one of the greatest achievements of mankind. Since its inception in 1960's, IC technology has evolved from integration of a few transistors in *Small Scale Integration* (*SSI*) to integration of millions of transistors in *Very Larger Scale Integration* (*VLSI*) chips currently in use. ICs consist of a number of electronic components, built by layering several different materials in a well-defined fashion on a silicon base called *wafer*. The designer of an IC transforms a circuit description into a geometric description, called the *layout*. The process of converting the specification of an electrical circuit into a layout is called the *Physical Design Cycle*. Due to the tight tolerance requirements and the extremely small size of individual components, physical design is an extremely tedious and an error prone process. As a result, almost all phases of physical design extensively use *Computer Aided Design* (*CAD*) tools and many phases have already been partially or fully automated.

1.1 VLSI Design Cycle

The VLSI design cycle starts with a formal specification of a VLSI chip, follows a series of steps, and eventually produces a packaged chip. A typical design cycle may be represented by the flow chart shown in Figure 1.1.

System specification is a high level representation of the system. The factors to be considered in this process include: performance, functionality and the physical dimensions. It is a compromise between market requirements, technological and economical viability.

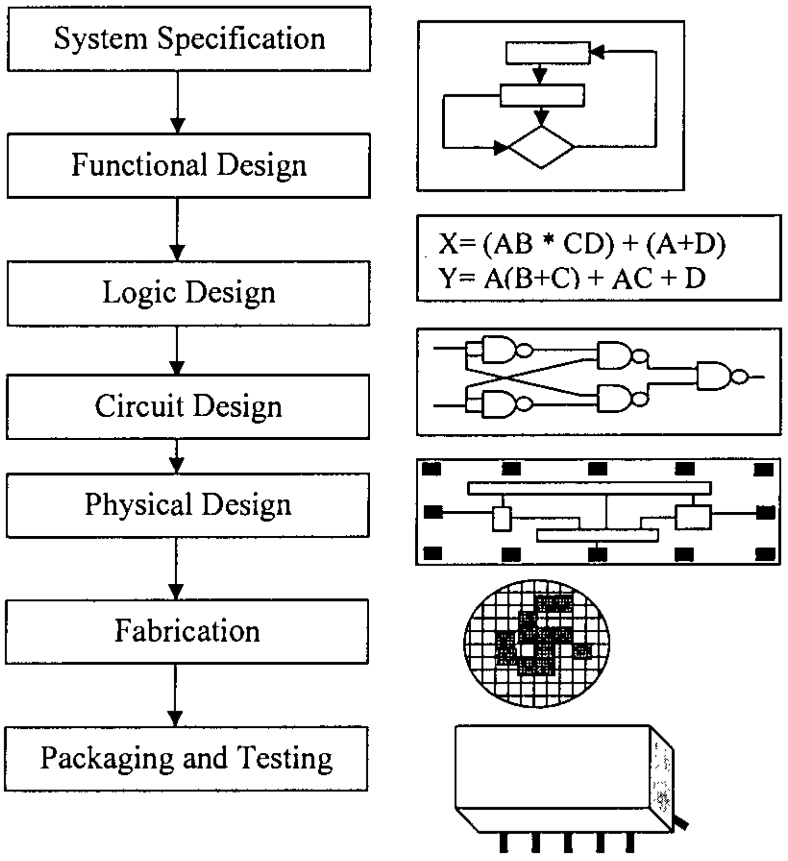


Figure 1.1: A Simple VLSI Design Cycle

In functional design step, main functional units and different parameters of the system are identified. This step also identifies the communications between the units.

In logic design, the control flow, word widths, register allocation, arithmetic operation and logic operations of the design that represent the functional design are derived and tested.

The purpose of the circuit design is to develop a circuit specification based on the logic design.

In physical design, the circuit specification is converted into a geometric representation called *layout*.

Layout data is converted into photolithographic *masks* during fabrication. Mask identifies spaces on the wafer, where a certain material needs to be deposited, diffused or even removed.

Fabricated chip is then packaged and tested to ensure that it meets all design specifications and that it functions properly.

1.2 Physical Design Cycle

Physical design is a very complex process and therefore it is usually broken into various sub steps. The input to the physical design cycle is a circuit specification and the output is the layout of the circuit. This is accomplished in several stages such as *Partitioning*, *Floorplanning*, *Routing* and *Compaction*.

1.2.1 Partitioning

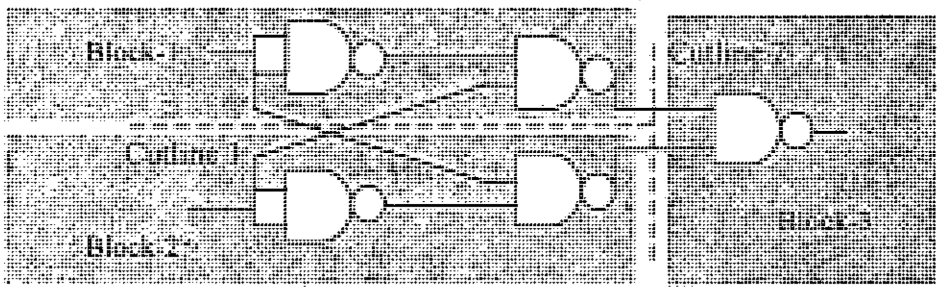


Figure 1.2: Partitioning into three blocks

The input to physical design cycle is a circuit design and output is the layout of the circuit which is later been fabricated, packaged and tested. A chip may contain several millions of transistors. Layout of the entire circuit cannot be handled at once due to the limitation of memory space as well as computation power available. Therefore, it is normally partitioned by grouping the components into *blocks*. The output of partitioning is a set of blocks and the interconnections required between the blocks. Figure 1.2 shows that the input circuit has been partitioned into three blocks. In the large circuit, the partitioning process is hierarchical and at the topmost level a chip may have 5 to 25 blocks [2]. Each block is then partitioned recursively into smaller blocks.

1.2.2 Floorplanning

After the circuit partitioning, the area occupied by each block can be calculated and the number of terminals (pins) required by each block is known. Given a set of blocks and the interconnection among them, the floorplanning problem is to assign each block a rectangular area on a plane such that each connection can be routed through the space which is not occupied by any block and no two nets intersect each other. Floorplanning is a key step in physical design cycle. A poor floorplanning consumes larger area and results in performance degradation. It generally leads to a difficult or sometimes impossible routing task.

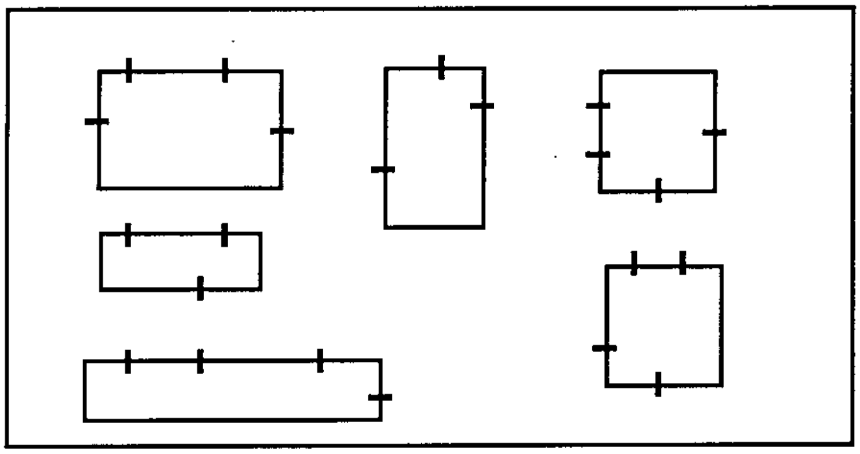


Figure 1.3: Circuit blocks and pins after floorplanning.

1.2.3 Routing

Spaces not occupied by the blocks during floorplanning can be viewed as a collection of regions. These regions are used for routing and is called *routing region.* The process of finding the geometric layout of all the interconnectios among the blocks through the routing region is called *routing.* The entire process of routing is divided into two phases: global routing and detailed routing. If the position of the connection terminals on each block is flexible, that is, actual position of connection could be floated along the side lines of the block, then the routing problem is called *global routing problem.* It generates a loose route for each net.

But if the position of the connection terminals on each block is fixed then the routing problem is called *detailed routing problem.* Detailed routing finds the actual geometric layout of each net within the routing region. The concepts of global and detailed routing are shown in Figures 1.4 and 1.5, respectively.

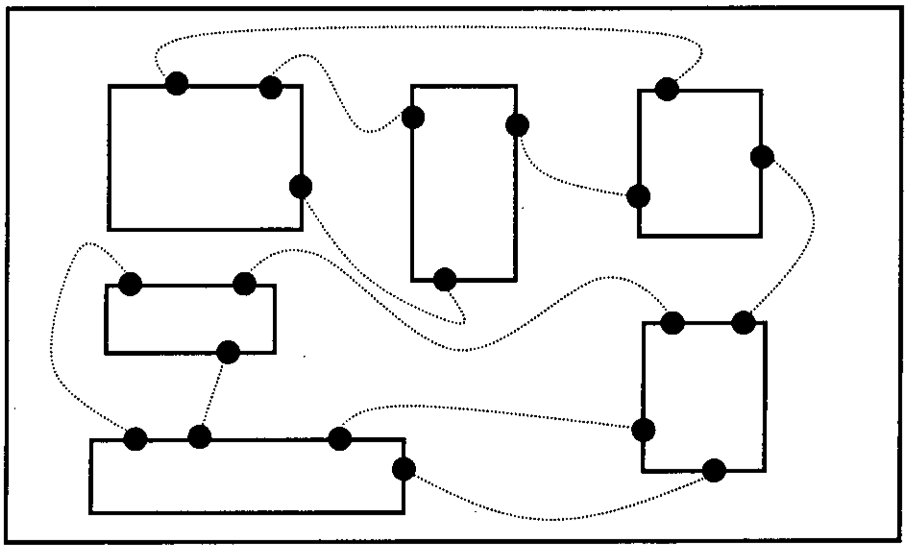


Figure 1.4: Global routing.

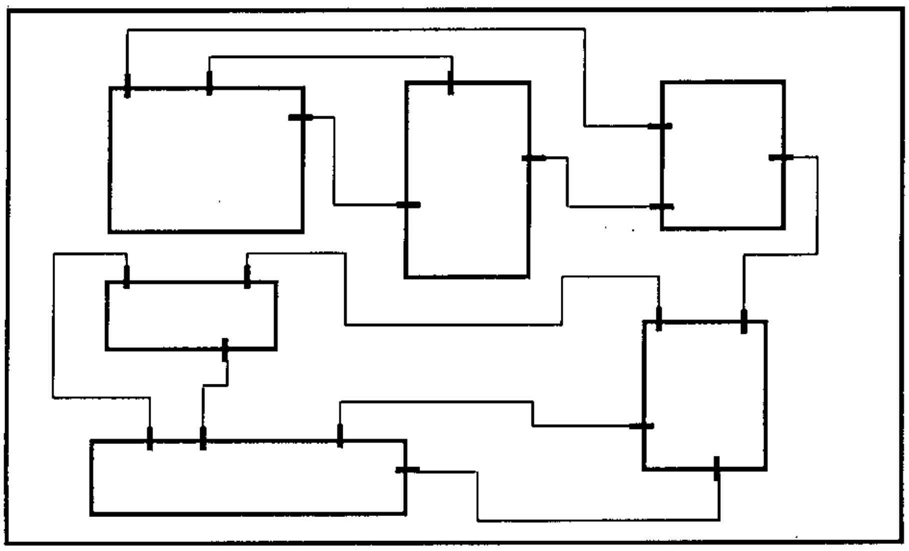


Figure 1.5: Detailed routing.

1.3 Traditional Approach of Floorplanning and Routing

A set of blocks and a netlist (as obtained from partitioning) is the input to the floorplanning problem. A floorplanning algorithm generates a floorplan for the given partitioning of the circuit. Then the global routing of the floorplan is determined if there is any feasible global routing exists; otherwise we backtrack to the floorplanning phase for any other alternative floorplan. A feasible global routing is the base of the detailed routing phase. If no feasible detailed routing is possible for a given global routing we backtrack for alternative global routing. Thus the overall floorplanning and routing process may iterate for several times. The traditional approach of handling floorplanning and routing problems is shown in Figure 1.6.

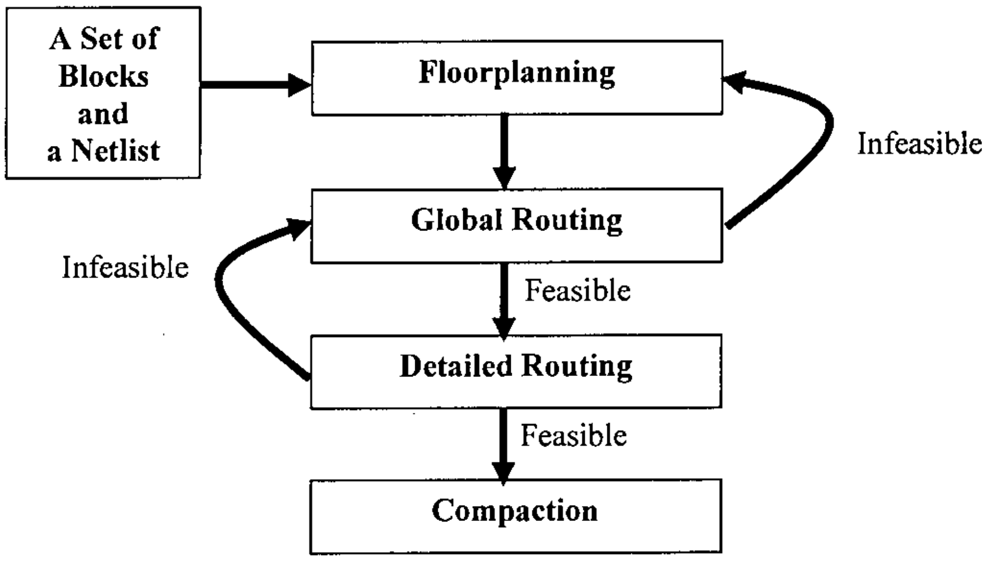


Figure 1.6: Traditional approach of floorplanning and routing

1.4 Objectives of this Thesis

Since 1960, many researchers have been worked on floorplanning and routing problems. Most of the work had treated floorplanning [1, 10, 14, 15, 16, 18, 19, 20, 21] and routing [3, 29, 31] as two successive phase in the physical design. Here we have presented an integrated approach to solve floorplanning and global routing problems in linear time using orthogonal graph drawing. We also have developed a linear-time algorithm to find the detailed routing from the global routing that we obtained from our integrated floorplanning and global routing algorithm. In Figure 1.7, a brief overview of the overall technique is shown.

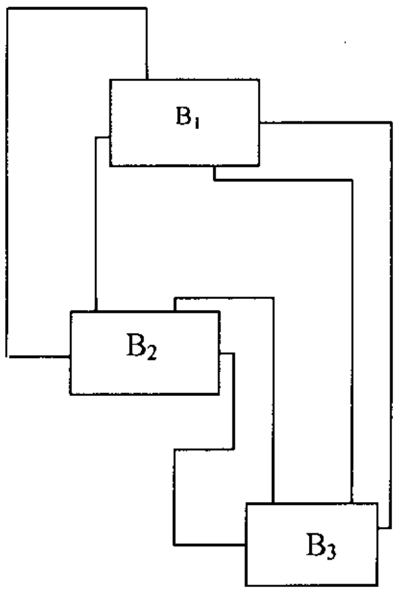


Figure 1.7: (a) A circuit C (set of blocks and their interconnections)

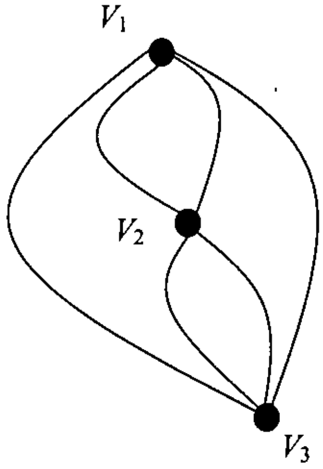


Figure 1.7: (b) Multi-graph G for the circuit C

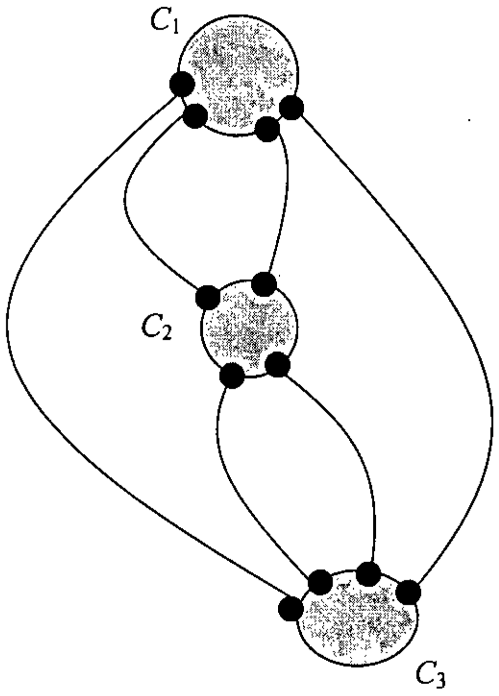


Figure 1.7: (c) A 3-graph G' equivalent to G

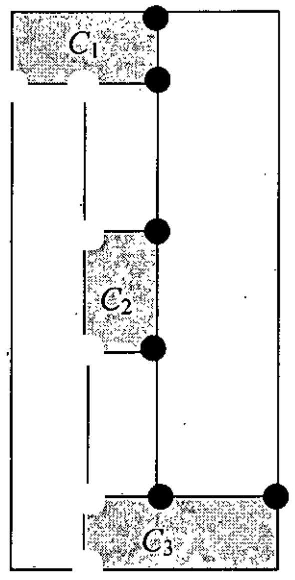


Figure 1.7: (d) Floorplanning and global routing for circuit *C* using orthogonal graph drawing

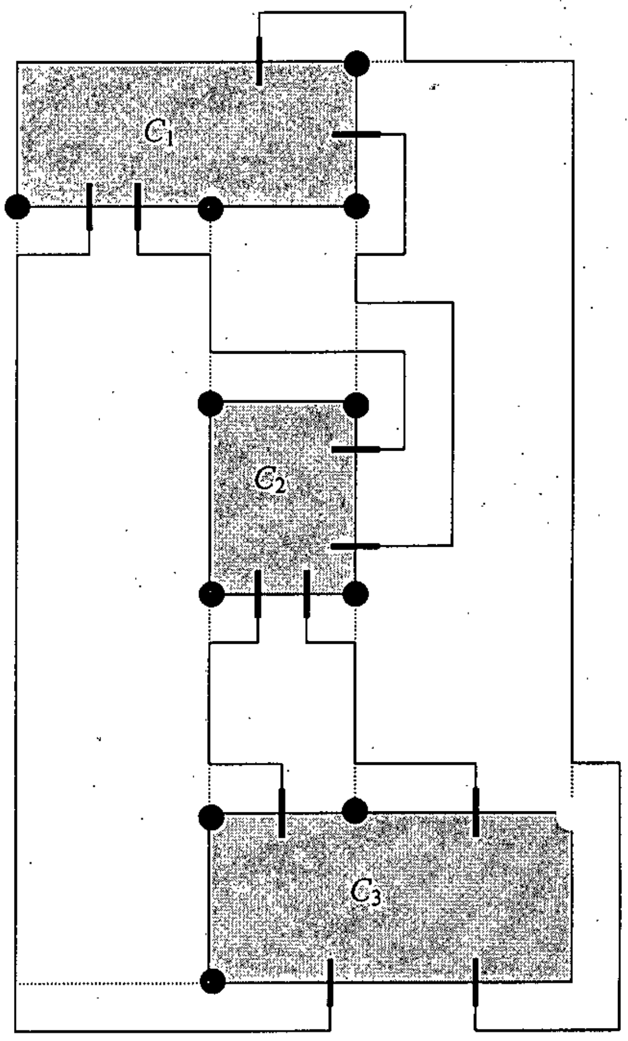


Figure 1.7: (e) Detailed routing *R* for circuit *C*

At first, the circuit blocks and the interconnections between them (Figure 1.7a) is represented by an *interconnection graph* G (Figure 1.7b), which may be a multi-graph. Then this multi-graph is converted to a 3-graph G' (Figure 1.7c) for which we can get an orthogonal graph drawing (Figure 1.7d). This orthogonal drawing gives the solution for the floorplanning and global routing problems but the actual pin assignment remain unsolved. Later, our detailed routing algorithm gives solution for the pin assignment and thus gets the detailed routing *R* (Figure 1.7e) for the circuit *C*. The details of this approach will be discussed at the later chapters.

1.5 Organization of the Thesis

Chapter 1 of this thesis gives a brief introduction of the floorplanning and routing problems and how these problems are handled traditionally. Here we also give a very brief overview of our approach to handle these problems. In Chapter 2 fundamentals of graph theoretic terminologies and some graph drawing techniques have been defined. We also stated some known results on graph drawing there. In Chapter 3 we developed an algorithm that solves floorplanning and global routing in an integrated approach. An algorithm for detailed routing of the global routing obtained in Chapter 3 is developed in Chapter 4. Finally Chapter 5 concludes our works.

Chapter 2

Preliminary

In this chapter we present some basic terms and easy observations related to our work. Definitions that are not included in this chapter will be introduced as they are needed. In Section 2.1, we start with some definitions of the standard graph theoretic terminologies used throughout the thesis. In Section 2.2, we define very common graph drawing techniques. Finally, some known results on orthogonal graph drawings are described in Section 2.3.

2.1 Basic Terminology

2.1.1 Graphs and Multigraphs

A *graph* is an ordered pair of (V, E) which consists of a finite set of vertices V and a finite set of edges E; each edge is an unordered pair of distinct vertices. We call $V(G)$ the *vertex-set* of graph G and $E(G)$ the *edge-set* of G. If $e = (v, w)$ is an edge, then e is said to join the vertices v and w and these vertices are said to be *adjacent*. In this case we also say that w is a *neighbour* of v and that e is *incident* to v and w. If the graph G has no multiple edge or loop then G is said to be a *simple graph*. *Multiple edges* join the same pair of vertices, while a *loop* joins a vertex to itself. The graph in which loops and multiple edges are allowed is called a *multigraph*. Sometimes a simple graph is simply called *graph*, if doing so creates no confusion.

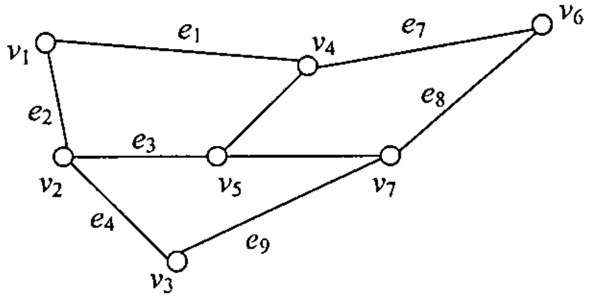


Figure 2.1: A simple graph.

The *degree* of a vertex v is the number of neighbors of v in G. We denote the maximum degree of graph G by $\Delta(G)$ or simply by Δ. If every vertex of a graph G has degree three or less then G is called a *3-graph.* If every vertex of a graph G has degree exactly three then G is called a *cubic graph.*

2.1.2 Subgraphs

A *subgraph* of a graph $G = (V, E)$ is a graph $G' = (V', E')$ such that $V' \subseteq V$ and $E' \subseteq E$; we write this as $G' \subseteq G$. If G' contains all the edges of G that join two vertices in V' then G' is said to be the *subgraph induced by* V' and is denoted by $G[V']$. If V' consists of exactly the vertices on which edges in E' are incident then G' is said to be the *subgraph induced by* E' and is denoted by $G[E']$.

We often construct new graphs from old ones by deleting some vertices or edges. If v is a vertex of a given graph $G = (V, E)$ then $G - v$ is a subgraph of G obtained by deleting the vertex v and all the edges incident to v. More generally, if V' is a subset of V then $G - V'$ is subgraph of G obtained by deleting the vertices in V' and the edges incident to them. Then $G - V'$ is a subgraph of G induced by $V - V'$. Similarly if e is an edge of G then $G - e$ is the subgraph of G obtained by deleting the edge e. More generally, if $E' \subseteq E$ then $G - E'$ is a subgraph of G obtained by deleting all edges in E'.

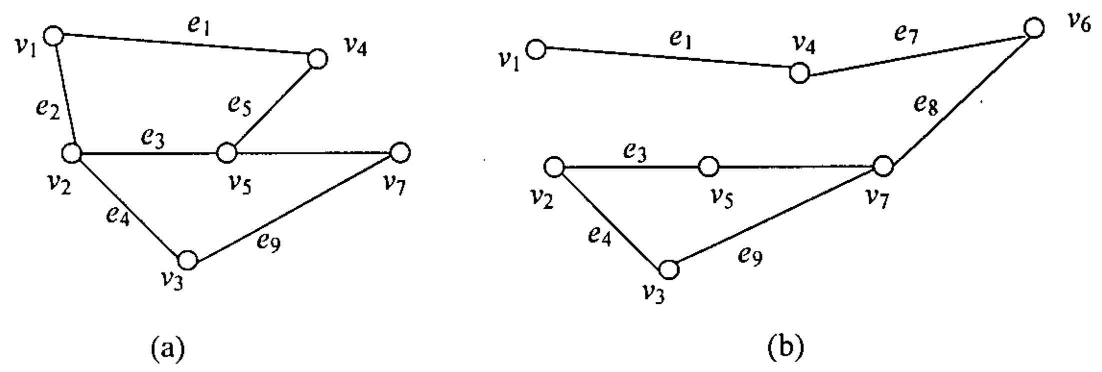


Figure 2.2: Subgraphs of G in Fig. 2.1: (a) vertex-induced and (b) edge-induced subgraph.

2.1.3 Weighted Graphs

A graph where each of the edges has a positive weight associated with it is called a *weighted graph*. Now if N is the set of all positive integers then we can define the weight function for the edges as $w: E \rightarrow N$. Figure 2.3 shows a weighted graph with five vertices and six edges.

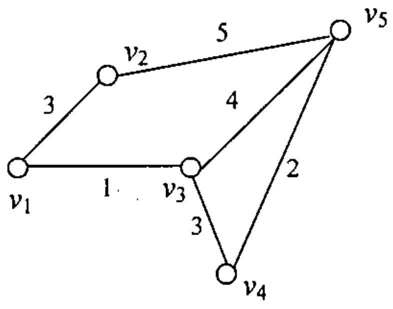


Figure 2.3: A weighted graph with five vertices.

2.1.4 Paths, Distances and Cycles

A v_0–v_l *walk* in G is an alternating sequence of vertices and edges of G, v_0, e_1, v_1,..., v_{l-1}, e_l, v_l, beginning and ending with a vertex, in which each edge is incident to two vertices immediately preceding and following it. If the vertices v_0, v_1,...,v_l are distinct (except possibly v_0, v_l), then the walk is called a *path* and is usually denoted by v_0 v_1,..., v_l. The *length* of a path in an unweighted graph is l, one less than the number of vertices on the path. In a weighted graph, the length of a path is determined by weights of the edges constituting the path. So the length $w(P)$ of a path P is defined as $w(P)=\sum_{e \in P} w(e)$.The *distance* between any two vertices in a graph is the length of the shortest path in the graph between the two vertices. We denote the distance from a vertex u to another vertex v by $dist(u, v)$. Now if the shortest path in G from u to v is P, then $dist(u, v) = w(P)$.

A path or walk is *closed* if $v_0 = v_l$. A closed path of length at least one is called a *cycle* where $v_0 = v_l$ is the only vertex repetition. An edge of G, which is incident to exactly one vertex of a cycle C and located outside of C is called a *leg* of the cycle C. The vertex of C to which a leg is incident is called a *leg-vertex* of C. A cycle in G is called a *k-legged* cycle of G if C has exactly k legs in G.

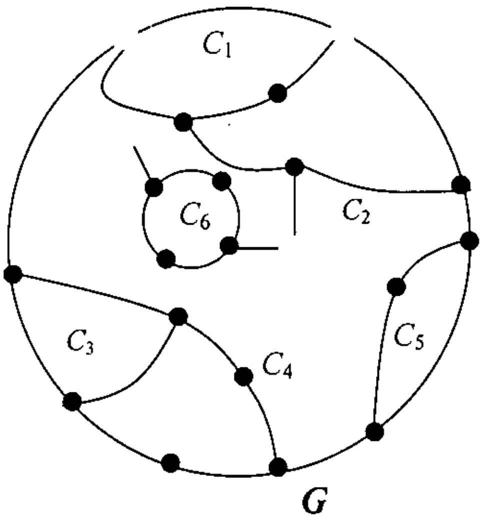


Figure 2.4: C_1 and C_3 are two 3-legged cycles.

2.1.5 Connectivity

The connectivity $\kappa(G)$ of a graph G is the minimum number of vertices whose removal results in a disconnected graph or a single vertex graph. We say that G is *k-connected* if $\kappa(G) > k$. We call a vertex of G a *cut vertex* if its removal results in a disconnected graph.

2.1.6 Planar Graph

A graph is *planner* if it can be embedded in the plane so that no two edges intersect geometrically except at a vertex to which they both are incident. A *plane* graph is a planar graph with a fixed embedding in the plane. A plane graph G divides the plane into connected regions called *faces*. The unbounded region is called *outer face*. We regard the *contour* of a face as a clockwise cycle formed by the edges on the boundary of the face. We denote the contour of the outer face of the graph G by $C_o(G)$.

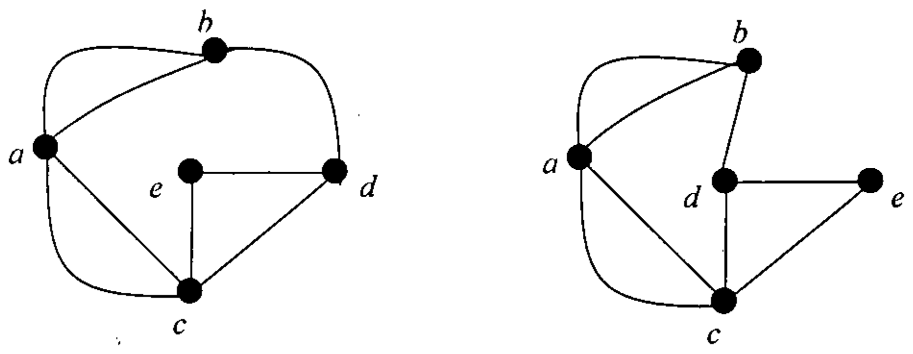


Figure 2.5: Two plane graphs of a same planar graph.

Let G be a plane connected graph. For a cycle C in G, we denote by $G(C)$ the plane subgraph of G inside C (including C). We say that cycles C and C' in a plane graph G are independent if $G(C)$ and $G(C')$ have no common vertex. A set S of cycles is independent if any pair of cycles in S is independent.

2.1.7 Trees

A *tree* is a connected graph without any cycles. Figure 2.6 is an example of a tree. A *rooted tree* is a tree in which one of the nodes is distinguished from others. This distinguished node is called the *root* of the tree. The root of the tree is generally drawn at the top. A tree is called *binary tree* if each node of the tree has at most two children. In Figure 2.5, the root is node 1.

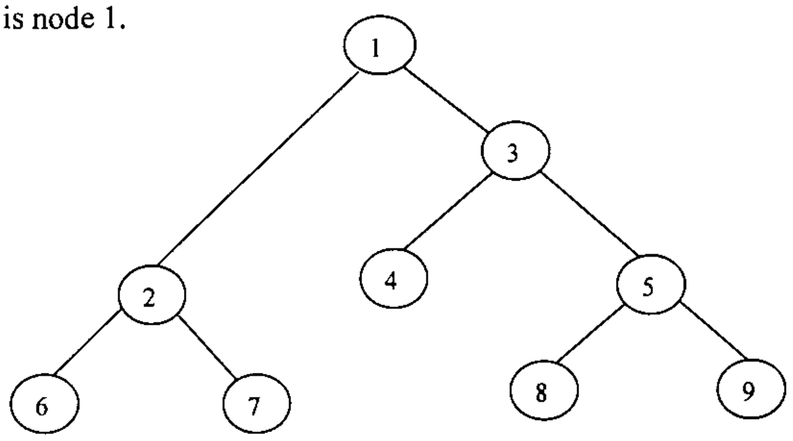


Figure 2.6: A binary tree with 9 vertices.

Every vertex u other than the root is connected by an edge to some other vertex p called the *parent* of u. We also call u a *child* of p. We draw the parent of a node above that node. For example, in figure 2.5, node 1 is the parent of node 2 and node 3. Alternately, nodes 6 and 7 are children of node 2. A *leaf* is a node of a tree that has no child. Thus every node of a tree is either a leaf or an internal node, but not both. In figure 2.6, the leaves are 4, 6, 7, 8 and 9 and nodes 1, 2, 3 and 5 are internal nodes.

The parent-child relationship can be extended naturally to ancestors and descendants. Suppose, u_1, u_2, ..., u_l is a sequence of nodes in a tree such that u_1 is the parent of u_2, which is the parent of u_3 and so on. The node u_1 is called an *ancestor* of u_3 and node u_3 is called a *descendant* of u_1. The root is the ancestor of every node in a tree and every node is a descendant of the root. In Figure 2.6, all nodes other than node 1 are descendants of node 1 and node 1 is an ancestor of all other nodes.

In a tree T, a node u together with all of its descendants, if any, is called a *subtree* of T. Node u is the root of this subtree. Referring again to Figure 2.5, node 6 by itself is a subtree, since node 6 has no descendant. Again, nodes 2, 6 and 7 form a subtree with root 2. Finally the entire tree of Figure 2.6 is a subtree of itself with root 1. The *height* of a node u in a tree is the length of the longest path from u to a leaf under u. The height of a tree is the height of a root. The *depth* of a node u in the tree is the length of a path from the root to u. In Figure 2.6, for example, node 3 is of height 2 and depth 1. The tree has height 3.

2.2 Graph Drawing Conventions

Automatic graph drawings have numerous applications in VLSI layout, networking, computer architecture, circuit schematics, etc [5, 17]. For the last few years many researchers have concentrated their attention on graph drawings and introduced a number of conventions. A drawing convention is some basic rules that the drawing must satisfy. A list of widely used drawing conventions is [17]

- Orthogonal Drawing,
- Box-orthogonal Drawing,
- Rectangular Drawing,
- Box-rectangular Drawing, and
- Grid Drawing

2.2.1 Orthogonal Drawing

An *orthogonal drawing* of a plane graph G is a drawing of G in which each vertex is drawn as a grid point on an integer grid and each edge is drawn as a sequence of alternate horizontal and vertical line segments along grid lines as illustrated in Figure 2.7(b). Any plane graph with maximum degree at most four has an orthogonal drawing [26], but may need bends. Minimization of number of bends in an orthogonal drawing is a challenging problem. Several works have been done on this issue [7, 8, 23, 25, 30]. In a VLSI floorplanning problem, an input is often a plane graph of maximum degree 3 [12, 24, 25]. Rahman *et al.* gave an algorithm to find an orthogonal drawing of a given plane 3-graph with minimum number of bends in linear time [27]. However, a plane graph with a vertex of degree 5 of more has no orthogonal drawing. A plane graph G may have an orthogonal drawing without bends [26]. However, not every plane graph has an orthogonal drawing without bends. For example, the cubic plane graph in Figure 2.7(a) has no orthogonal drawing without bends.

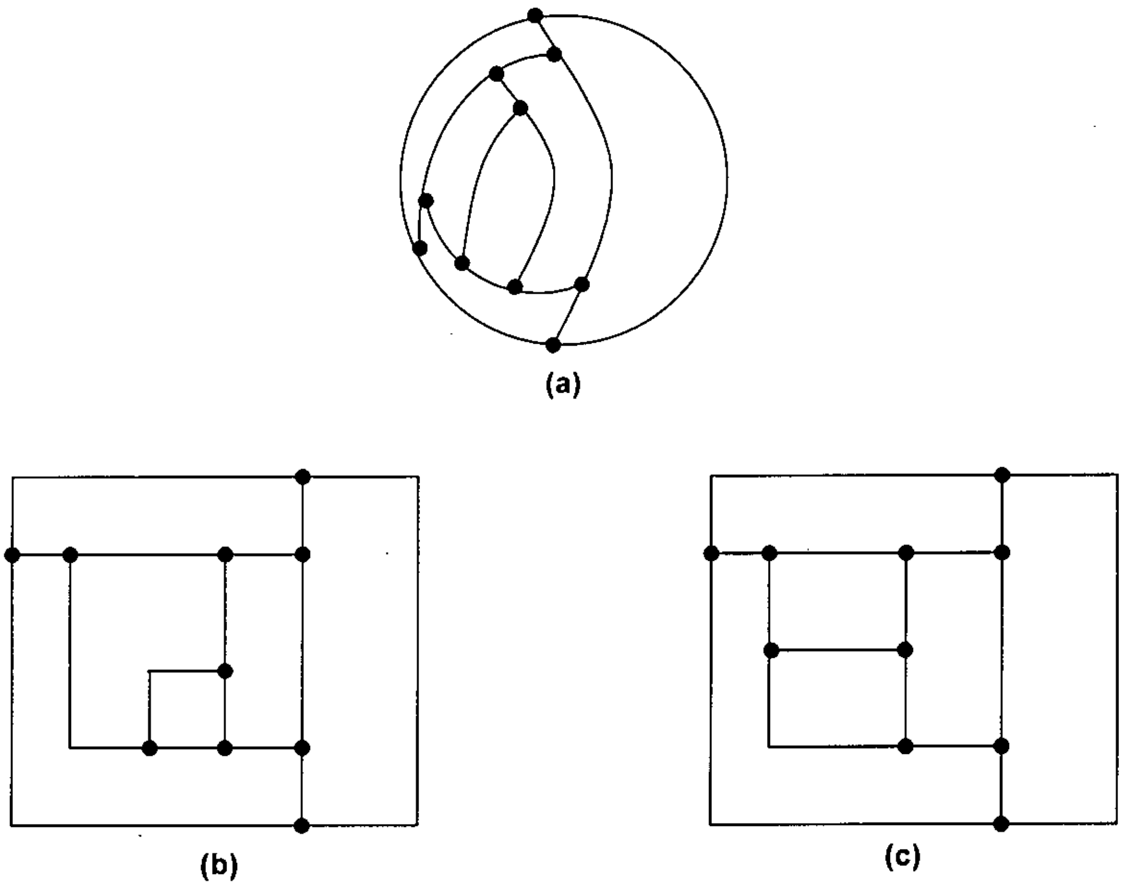
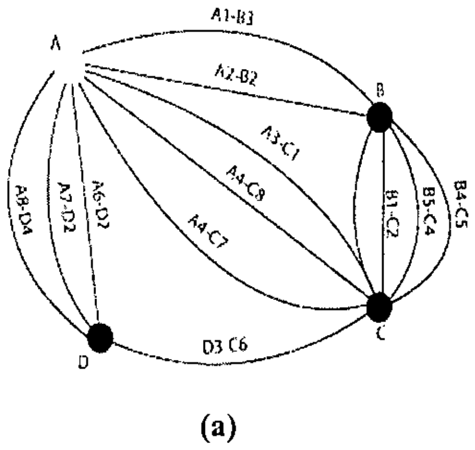


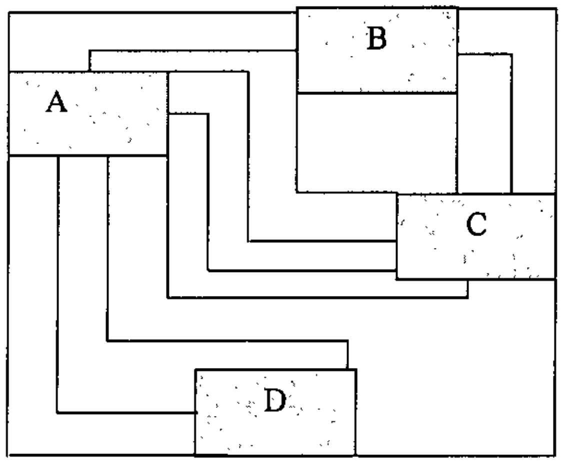
Figure 2.7: (a) A plane graph G, (b) an orthogonal drawing of G with 6 bends, (c) an orthogonal drawing of G with 5 bends.

2.2.2 Box-Orthogonal Drawing

A *box-orthogonal drawing* of a plane graph G is a drawing of G on an integer grid such that every vertex is drawn as a (possibly generated) rectangle, called a box, and each edge is drawn as sequence of alternate horizontal and vertical line segments along grid lines, as illustrated in Figure 2.8. Some of the boxes may be degenerated rectangles, i.e, points. A box-orthogonal drawing is a natural generalization of an ordinary orthogonal drawing.

Moreover, any plane (multi) graph has box-orthogonal drawing even if there is a vertex of degree 5 or more.





(b)

Figure 2.8: (a) A plane multi-graph *G*, (b) A box-orthogonal drawing of *G*.

2.2.3 Rectangular Drawing

A *rectangular drawing* of a plane graph is an orthogonal drawing in which there is no bend and each face of the graph is drawn as a rectangle. The drawing in Figure 2.9 is an example of a rectangular drawing.

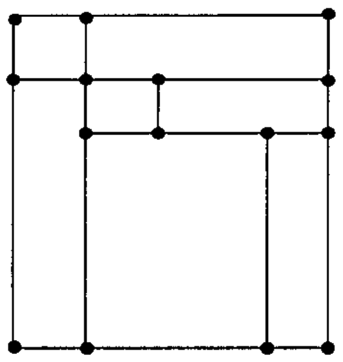


Figure 2.9: A rectangular graph drawing.

2.2.4 Box-Rectangular Drawing

A *box-rectangular drawing* of a graph is a drawing such that each vertex is drawn as a box, and the contour of each face is drawn as a rectangle. The drawing in Figure 2.10 is an example of a box-rectangular drawing. Unfortunately, not every plane graph has a box- rectangular drawing. Some works on box-rectangular drawing are done in [25].

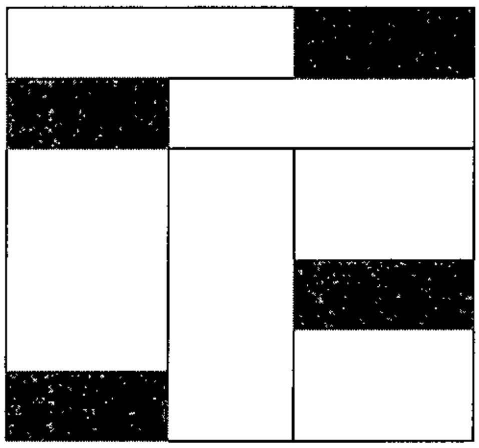


Figure 2.10: A box-rectangular graph drawing.

2.2.5 Grid Drawing

A drawing of a graph in which vertices and bends are located at grid points of an integer grid is called a *grid drawing*. The drawing in Figure 2.11 is an example of grid drawing. It is a very challenging problem to draw a plane graph on a grid of the minimum size. In recent years, several works have been done in this field [6, 23].

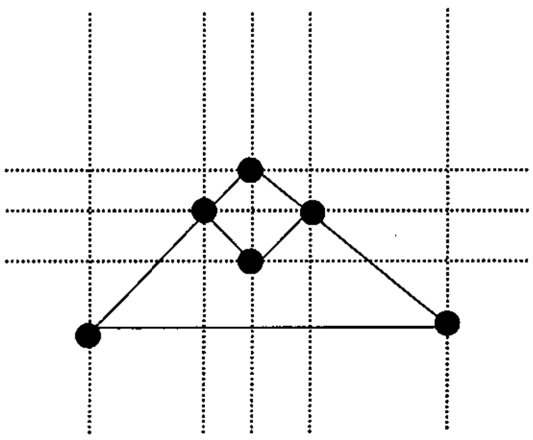


Figure 2.11: A grid drawing.

2.3 Some Known Results on Orthogonal Drawing

In Chapter 3, we present a floorplanning algorithm using orthogonal drawing of plane graph. Thus it is important to know some definitions and known results regarding orthogonal drawing of plane graphs. We first have the following lemma [30]:

Lemma 2.1 *Let G be a plane biconnected graph with maximum degree $\Delta \leq 3$. Assume that, four vertices of degree 2 on $C_o(G)$ are designated as the four corners of the outer rectangle. Then G has a rectangular drawing if and only if G satisfies the following two conditions*:

(*r1*) *every 2-legged cycle contains at least two designated vertices, and*

(*r2*) *every 3-legged cycle contains at least one designated vertex.*

Furthermore, one can check in linear time whether a graph G satisfies the conditions above, and if G does then one can find a rectangular drawing in linear time [22].

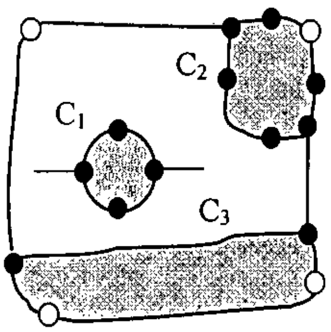


Figure 2.12: (a) 2-legged cycles.

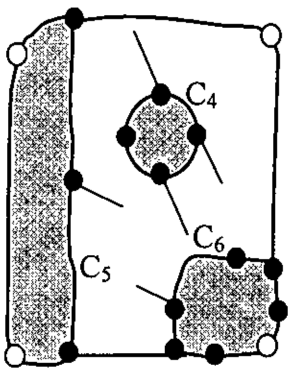


Figure 2.12: (b) 3-legged cycles.

In Figure 2.12, white circles are the four designated corner vertices and the black circles are the non-designated vertices of each graph. Cycles C_1, C_2 and C_3 are 2-legged and cycles C_4, C_5 and C_6 are 3-legged. Cycles C_3, C_5 and C_6 do not violate any of the two conditions in lemma 2.1. On the other hand cycles C_1, C_2 and C_4 violate the condition.

A cycle in G violating the condition ($r1$) or ($r2$) of Lemma 2.1 is called a *bad cycle*: a 2-legged cycle is bad if it contains at most one designated vertex; a 3-legged cycle is bad if it contains no designated vertex. Thus cycles C_1, C_2 and C_4 are bad cycles.

For a cycle C in a plane graph G, we denote by $G(C)$ the plane subgraph of G inside C including C. A bad cycle C in G is called a *maximal bad cycle* if $G(C)$ is not contained in $G(C')$ for any other bad cycle C' of G. In Figure 2.13 C_1, C_2, C_4, C_5 and C_6 are bad cycles. Cycles C_1, C_4, C_5 and C_6 are maximal bad cycles. C_3 is not a maximal bad cycle because subgraph $G(C_3)$ is contained in a subgraph $G(C_4)$.

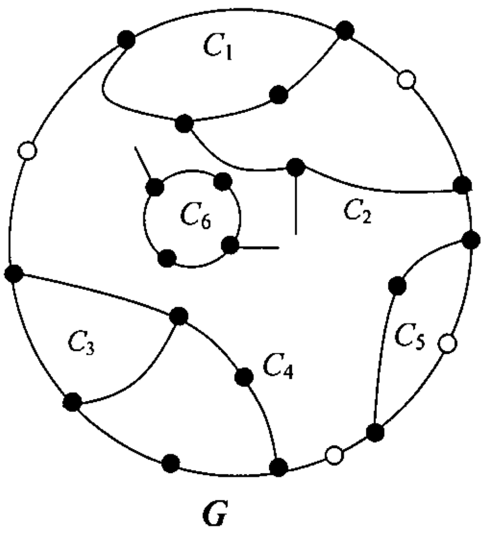


Figure 2.13: The maximal bad cycles C_1, C_4, C_5 and C_6 in graph G.

Cycles C and C^* in a plane graph G are *independent* of each other if $G(C)$ and $G(C^*)$ have no common vertex. We have the following lemma [26]:

Lemma 2.2 *Let G be a plane biconnected graph with maximum degree $\Delta \leq 3$. Assume that four vertices of degree 2 on $C_o(G)$ are designated as the four corners of the outer rectangle. Then the maximal bad cycles in G are independent of each other.*

An orthogonal drawing of a plane graph G is a drawing of G with the given embedding in which each vertex is mapped to a point and each edge is drawn as a sequence of alternate horizontal and vertical line segments and any two edges do not cross except at their common end. A *bend* is a point where an edge changes its direction in the drawing. The following results are known on the orthogonal drawing of a plane graph [26]:

Lemma 2.3 *Let G be a plane bi-connected graph with maximum degree $\Delta \leq 3$ and four or more vertices on $C_o(G)$ with degree two. Then G has an orthogonal drawing without bend if and only if every 2-legged cycle in G contains at least two vertices of degree 2 and every 3-legged cycle contains at least one vertex of degree 2.*

A subgraph H of G is a *biconnected component* of G if H is a biconnected subgraph of G having no cut vertex. A single edge (u, v) of G together with the vertices u and v is a *weakly biconnected component* of G if either both u and v are cut vertex or one of u and v is a cut vertex and the other is a vertex of degree one. The graph G in Figure 2.14(a) has three biconnected components H_1, H_2 and H_3 as shown in figure 2.14(b). Edges (u_1, v_1), (u_2, v_2), (u_3, v_3) and (u_4, v_4) are weakly biconnected components.

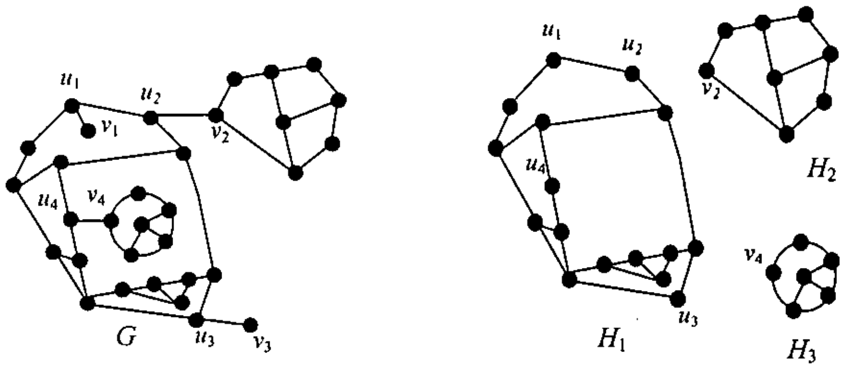


Figure 2.14: (a) A plane graph G, (b) three biconnected components of G.

We call two subgraph H_i and H_j of G are *disjoint* with each other if H_i and H_j have no common vertex. The following lemma is known [26]:

Lemma 2.5 *Let G be a plane bi-connected graph with maximum degree $\Delta \leq 3$. Then the bi-connected components in G are disjoint with each other.*

A *block* of a connected graph G is either a bi-connected component or a weakly bi-connected component of the graph. A tree called *BC-tree* of G can represent the blocks and cut vertices of the graph. In the BC-tree of G each block is represented by a *B-node* and each cut vertex is represented by a *C-node*. In Figure 2.15 the BC-tree of graph G in Figure 2.14 is shown, where each B-node is represented by a rectangle and a circle represents each C-node.

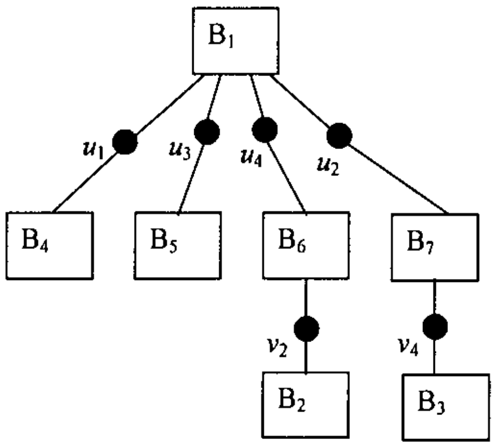


Figure 2.15: BC-tree of the graph G of Figure 2.16.

Chapter 3

Floorplanning and Global Routing

3.1 Introduction

Floorplanning is one of the very important phases of the physical design cycle in automated VLSI design. The input to physical design cycle is a circuit diagram and the output is the layout of the circuit which is later been fabricated, packaged and tested. A chip may contain several million transistors and is normally partitioned by grouping the components into blocks. The output of partitioning is a set of blocks and the interconnection required between the blocks. After the circuit partitioning phase, the area occupied by each block can be calculated and the number of terminals (pins) required by each block is known. A connection between two terminals of two blocks is called a *net* and the set of all nets among all the blocks is called a *netlist*. Given a set of blocks and a netlist the floorplanning problem is to assign each block a rectangular area on a plane such that each net of the netlist could be routed through the space which is not occupied by any block and no two nets intersect each other. Floorplanning is a key step in physical design cycle. A poor floorplanning consumes larger areas and results in performance degradation. It generally leads to a difficult or sometimes impossible routing task.

The floorplanning of blocks occur at three different levels: *system level*, *board level* and *chip level* [4]. At system level, the floorplanning problem is to place all the PCBs together so that area occupied is minimized. At board level, all the chips on a board along with other solid-state devices have to be placed with in a fixed area of the PCB. The PCB technology allows mounting of components on both sides. There is essentially no restriction on the number of routing layers in PCB. The objectives of the board level floorplanning algorithm is two fold: minimization of the number of routing layers and satisfaction of the system performance requirements. The key difference between the board level and chip level floorplanning problem is the limited number of layers that can be used for routing on a chip. In addition, the circuit is fabricated only on one side of the substrate.

Let us formally state the floorplanning problem. Let B_1, B_2, ..., B_n be the blocks to be placed on the chip. Let $N = \{N_1, N_2, ..., N_m\}$ be the set of nets representing the interconnections between different blocks. The floorplanning problem is to find the rectangular region for each block denoted by $R = \{R_1, R_2, ..., R_n\}$ on the plane such that

1. no two rectangles overlap each other, that is, $R_i \cap R_j = \phi$, $1 \leq i, j \leq n$; and
2. floorplanning is routable, that is, Q_i's, $1 \leq i \leq k$, are sufficient to route all the nets, where $Q = \{Q_1, Q_2, ..., Q_k\}$ represents the empty areas allocated for routing between blocks.

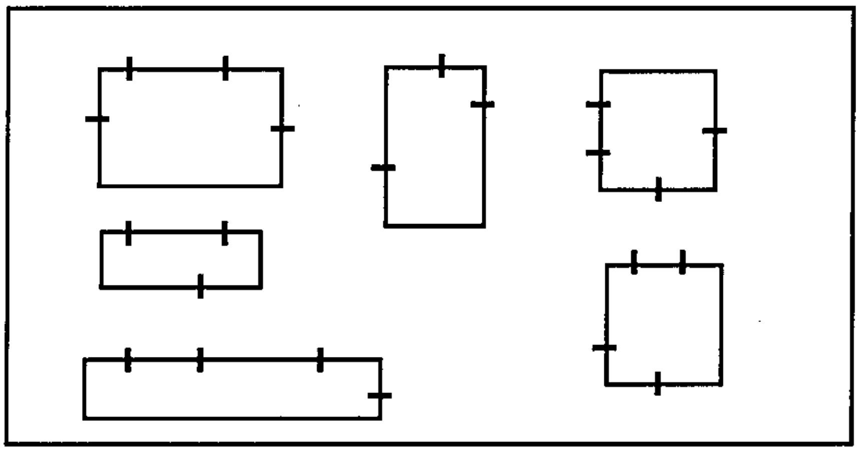


Figure 3.1: Circuit blocks and pins after floorplanning.

Spaces not occupied by the blocks during floorplanning can be viewed as a collection of regions. These regions are used for routing and is called *routing region.* The process of finding the geometric layout of all the nets through the routing region is called *routing.* Nets must be routed within the routing region and also must not be short circuited, that is, nets must not intersect each other. The entire process of routing is divided into two phases: *global routing* and *detailed routing.* If the position of the connection terminals on each block is flexible that is actual position of connection could be floated along the side lines of the block then the routing problem is called global routing problem. It generates a loose route for each net. But if the position of the connection terminals on each block is fixed then the routing problem is called detailed routing problem. Detailed routing finds the actual geometric layout of each net within the routing region. Figure 3.2 and Figure 3.3 show the global routing and detailed routing respectively.

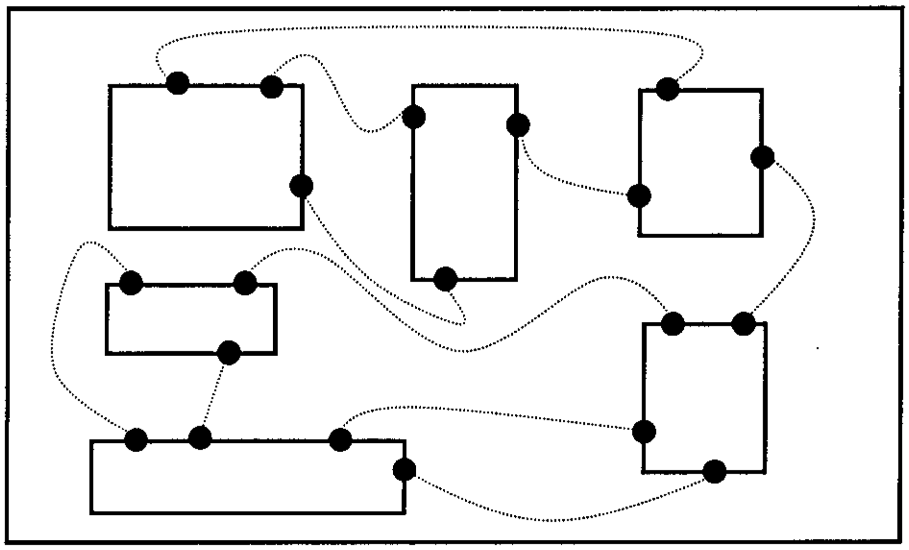


Figure 3.2: Global routing.

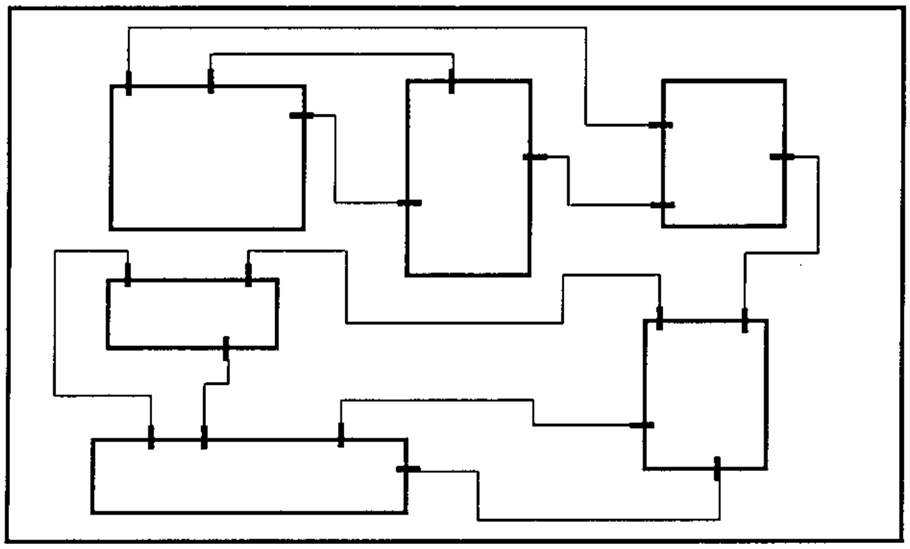


Figure 3.3: Detailed routing.

3.1.1 Traditional Approach of Floorplanning and Routing

A set of blocks and a netlist (as obtained from partitioning) is the input to the floorplanning problem. A floorplanning algorithm generates a floorplan for the given partitioning of the circuit. Then a global routing of the floorplan is determined if there is any feasible global routing exists; otherwise we backtrack for any other alternative floorplan. A feasible global routing is the base of the detailed routing phase. If no feasible detailed routing is possible for a given global routing then we backtrack for alternative global routing. Thus the overall floorplanning and routing process may iterate for several times. The traditional approach of handling floorplanning and routing problems is shown in Figure 3.4.

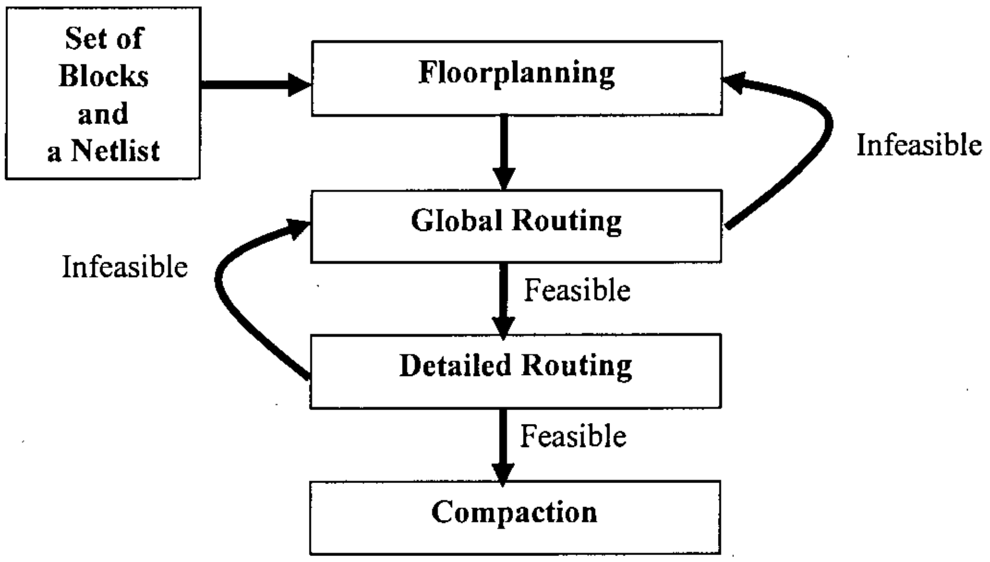


Figure 3.4: Traditional approach of floorplanning and routing.

3.1.2 Our Integrated Approach of Floorplanning and Routing

We handle floorplanning and global routing problems in a single phase. The input to our integrated floorplanning and global routing algorithm is a set of blocks and a netlist and output is the floorplan and the global routing layout (except detail routing) for it. Global routing layout is then used for detailed routing in the next phase. Our integrated algorithm can find floorplan and global routing for a circuit if those exists in a plane. Thus no iteration between floorplanning and routing is required.

In the rest of the chapter, we have presented the integrated approach of solving floorplanning and global routing problems in a single phase. We have used orthogonal graph drawing for this purpose. In Chapter 4, we have developed an algorithm to find detailed routing for the floorplan and we would get global routing from our integrated approach.

Our approach of handling floorplanning and routing is shown in Figure 3.5. This is a linear approach, that is, if there exists a feasible floorplan and global routing in a single plane then our approach could find it without any backtracks.

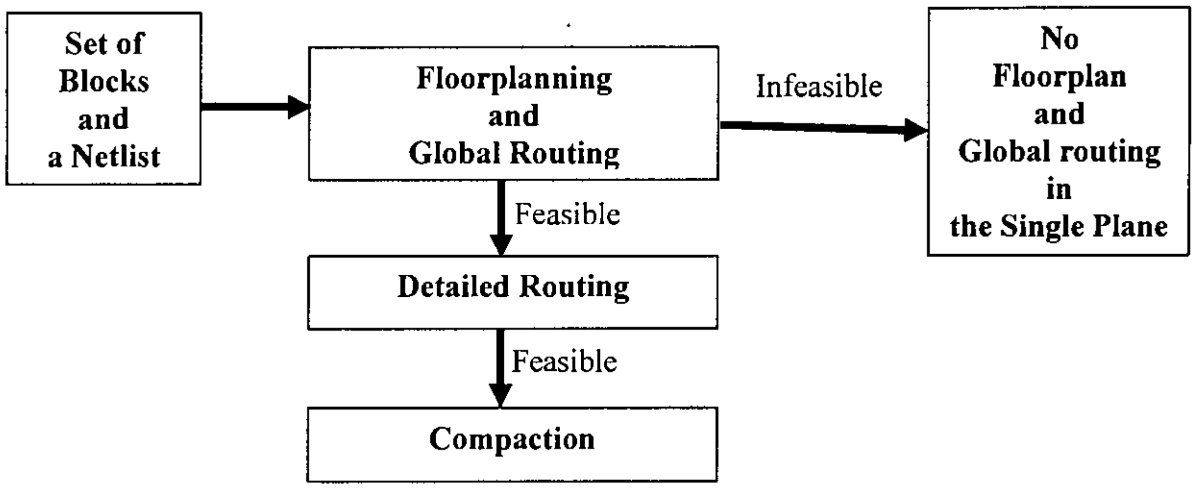

Set of Blocks and a Netlist
Floorplanning and Global Routing
Infeasible
No Floorplan and Global routing in the Single Plane
Feasible
Detailed Routing
Feasible
Compaction

Figure 3.5: Our integrated approach of floorplanning and routing.

3.2 Integrated Floorplanning and Global Routing

Unlike traditional approach we have integrated floorplanning and global routing in a single phase. We have used orthogonal graph drawing for this purpose. The outline for our algorithm is as follows:

Step-1 Represent the circuit C using a multigraph G.

Step-2 Get a 3-graph G' from the graph G.

Step-3 Get an orthogonal drawing D for the 3-graph G'.

Step-4 Get a floorplan and global routing F for the circuit C from the orthogonal drawing D.

3.2.1 Representing a Circuit using Multigraph

We are given a circuit specification consisting of circuit elements (blocks) and the connections between them called netlist. First we represent the circuit using a multigraph *G*, where each block (circuit element) is represented by a vertex and interconnections between blocks by edges. The resulting graph is a multigraph as there may be more than one connection between two same circuit blocks.

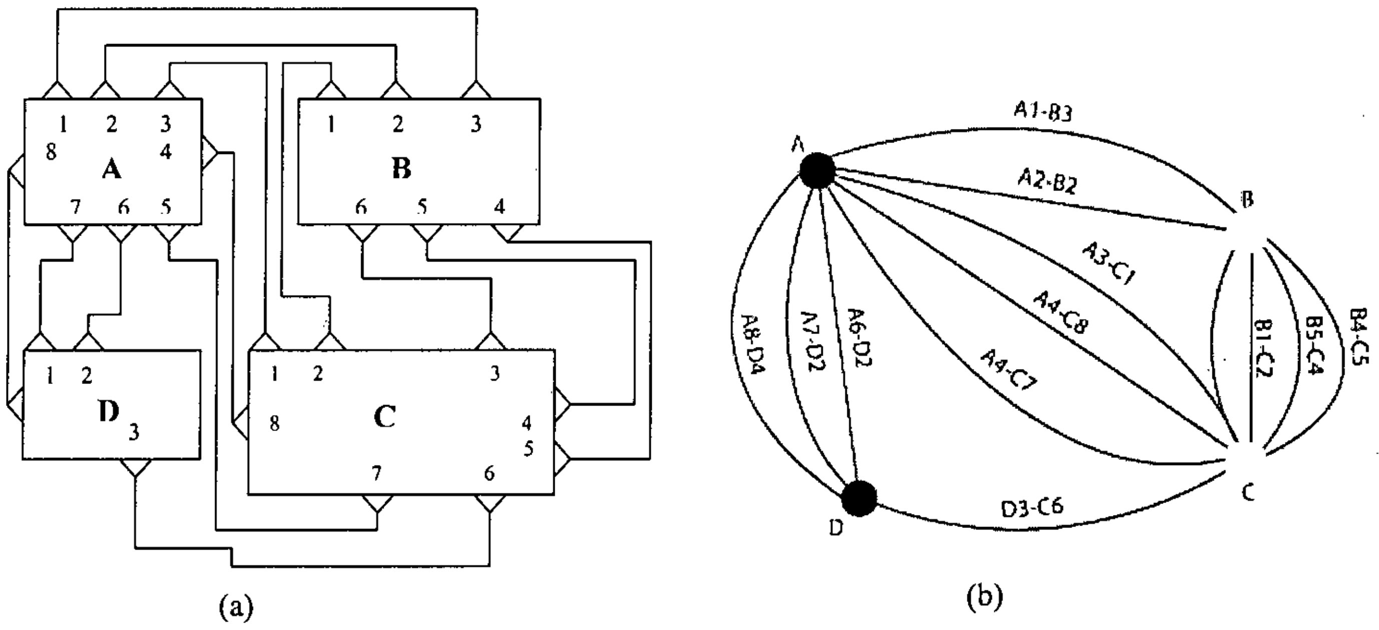


Figure 3.6: (a) A circuit specification, (b) The multigraph *G* for the circuit.

Assume that the graph *G* that obtained from the circuit specification as described above is planar and has a fixed planar embedding. For real and complex circuit, where huge number of circuit elements and the interconnection among them are involved, the graph *G* may not be planar. Actually the routing of the nets is done in multiple layers and the nets those reside in a particular layer do not intersect each other. Thus for a single layer the assumption of being the graph plane does match with real circuits. In a VLSI floorplanning problem, an input is often a plane graph of maximum degree 3 [12, 24, 25]. Distributing the nets in different routing layers is beyond the scope of this thesis.

3.2.2 Converting Multigraph into a 3-Graph

We use orthogonal graph drawing technique to solve the floorplanning and routing problems. Orthogonal drawings have attracted much attention due to their various applications, especially in circuit schematics, entity relationship diagrams, data flow diagrams etc. [5, 28, 30]. If G be a plane graph with maximum degree $\Delta \leq 3$, then G has a bend minimum orthogonal drawing [26, 27]. There is a linear-time algorithm to find such orthogonal drawing of G. It is desired to find an orthogonal drawing with a small number of bends, because a bend corresponds to a "via" or "through-hole", and increase the fabrication cost of VLSI circuit [12, 28]. But the graph we obtained from the circuit specification as described in Section 3.2.1 may consist of vertices with degree one, two, three, or more. Thus we need some conversion of the graph so that a bend minimum orthogonal drawing could be obtained by keeping the meaning of the circuit specification. Here we have described the process of getting a 3-graph G' (a graph with maximum degree $\Delta \leq 3$) from the multigraph G.

Let G be the graph obtained from the circuit specification as described in Section 3.2.1. Then G may have vertices of varying degrees. We replace each vertex with degree three or more with a cycle containing the number of vertices on the cycle equal to the degree of the original vertex. Let u be a vertex of graph G with degree m and $m \geq 3$; then u should be replace by a cycle C_u with vertices $\{u_1, u_2, \ldots, u_m\}$, and let v be a vertex with degree n; then v should be replace by a cycle C_v with vertices $\{v_1, v_2, \ldots, v_n\}$. Now an edge between u and v will be replaced by an edge between u_i and v_j, where $1 \leq i \leq m$ and $1 \leq j \leq n$. The graph G' that obtained in this way from the original graph G is a 3-graph.

A vertex of graph G represents a circuit element and such vertex with degree three or more is replaced by a cycle in the 3-graph. So each vertex on the replaced cycle in the 3-graph G' represents a pin of a circuit element. During this replacement of a vertex with its equivalent cycle, one important feature of circuit specification should be kept: the clockwise orientation of the connecting pins on the circuit element and its equivalent replaced vertices on the corresponding cycle in the 3-graph G' should be the same.

Let the clockwise orientation of the connecting pins of the circuit element (block) B_i is $\{P_{i1}, P_{i2}, \ldots, P_{im}\}$ where m is the number of connecting pins on block B_i and $m \geq 3$ and block B_i has been replaced by cycle C_i and vertex v_{ij} represents pin P_{ij}, where $1 \leq j \leq m$. Then the clock wise orientation of the vertices on cycle C_i starting from vertex v_{i1} should be $\{v_{i1}, v_{i2, \ldots,} v_{im}\}$; which is same as that of the clockwise orientation of the pins on block B_i. This relation is shown in Figure 3.7.

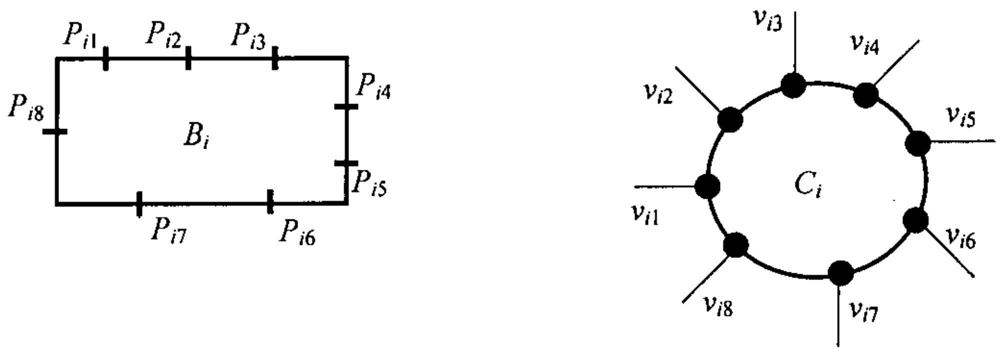


Figure 3.7: Clockwise orientation of pins and vertices.

The 3-graph G' obtained in this way may or may not be bi-connected as original graph G may have vertices of degree one or two also. The conversion of the multigraph G of Figure 3.8(a) to a 3-graph G' has been shown in Figure 3.8(b).

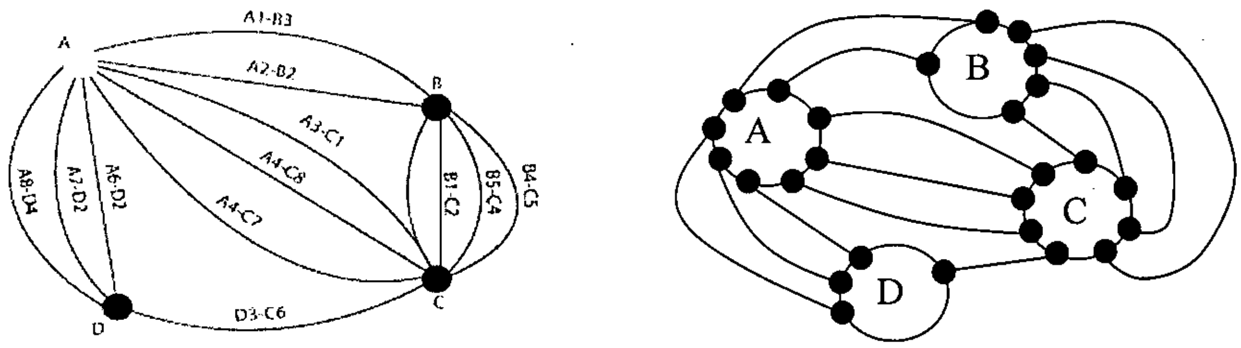


Figure 3.8: (a) Original multi-graph G, (b) 3-graph G'.

The 3-graph G' obtained in Figure 3.8 is bi-connected, but it is not always true. The 3-graph G' obtained in this way may not be bi-connected. Figure 3.9(c) shows such an instance.

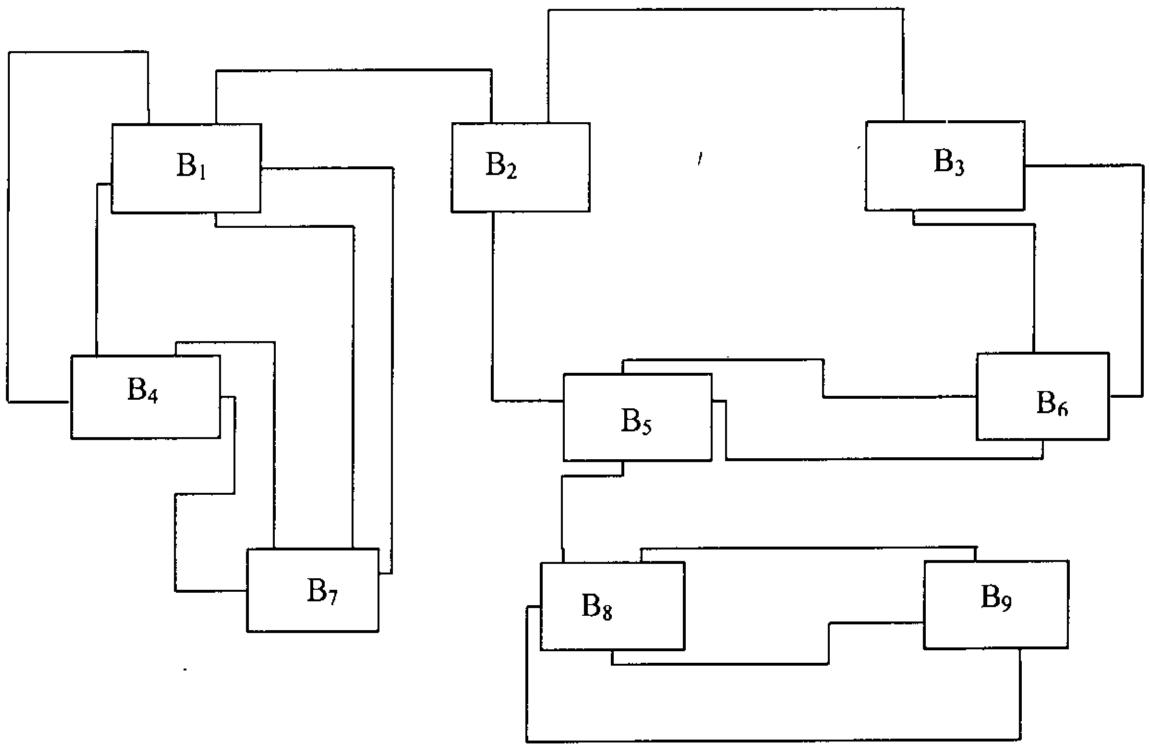


Figure 3.9: (a) A circuit specification.

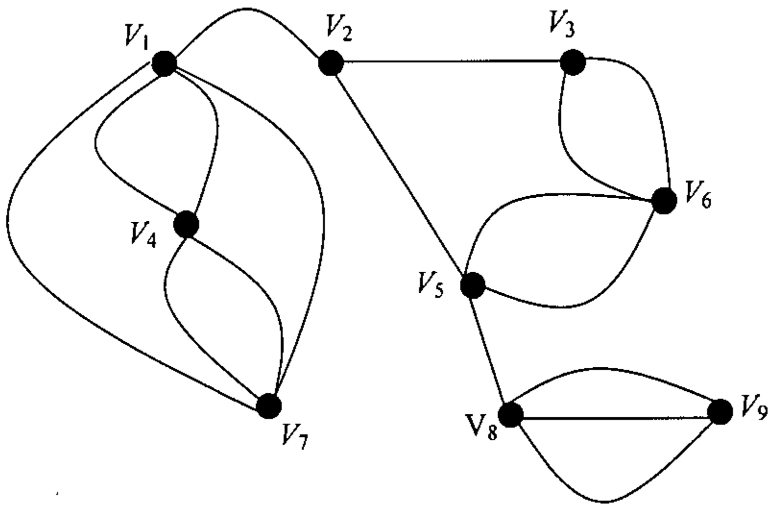


Figure 3.9: (b) Multigraph G for the circuit.

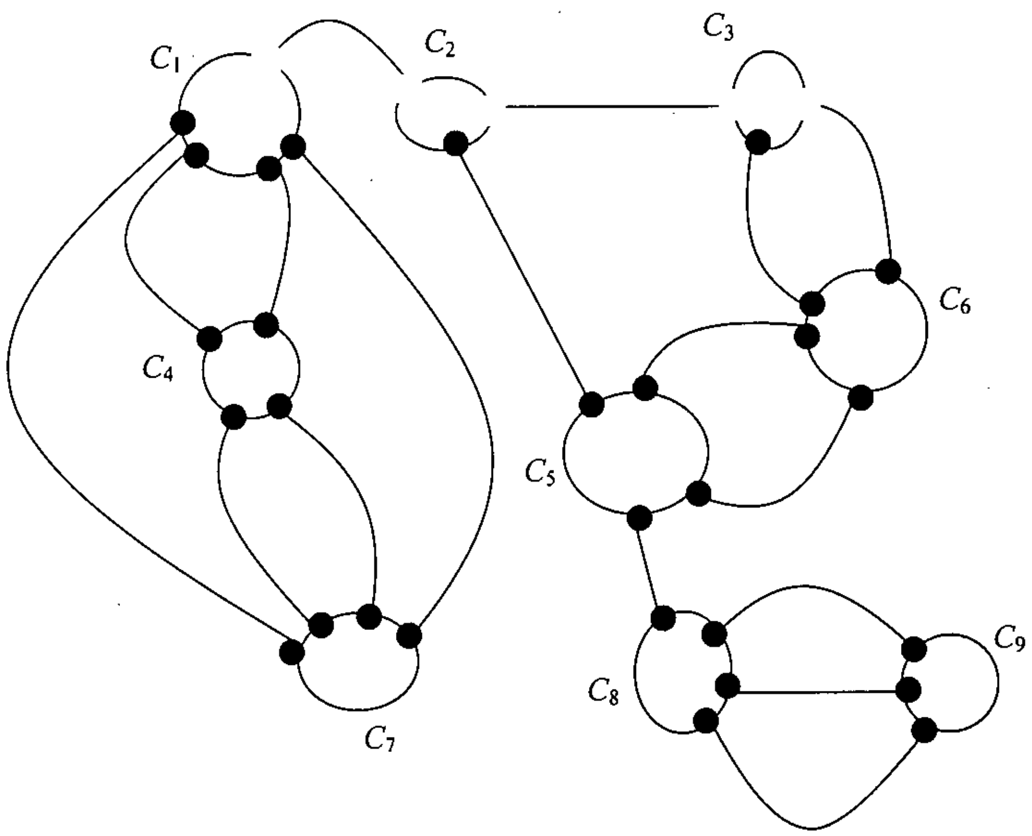


Figure 3.9: (c) A 3-graph G'.

3.2.3 Handling Multi-Terminal Net

The circuit specifications we have considered so far did not have *multi-terminal net*. A net is called multi-terminal net if it has a single source at one end and more than one destination at the other end. A multi-terminal net is shown in Figure 3.10.

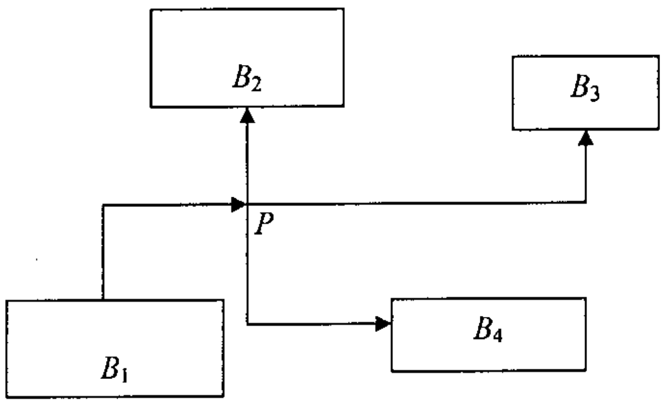


Figure 3.10: A multi-terminal net.

If the circuit containing multi-terminal net is represented by a graph as described in Section 3.2.1, we get a non-planar graph. To avoid this situation we replace the common point P by a vertex u in the multigraph G that represents the circuit. Then u has degree at least 3 and thus later be replaced by a cycle while converting the multi-graph G into a 3-graph. Handling a multi-terminal net is shown in Figure 3.11.

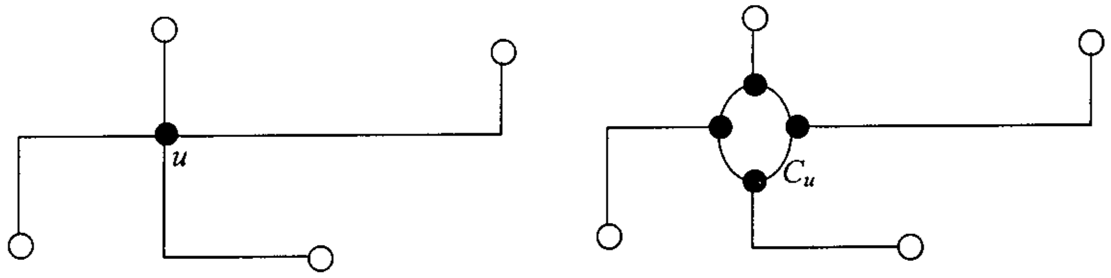


Figure 3.11: Representing multi-terminal net (a) in multigraph, (b) in 3-graph.

3.2.4 Handling Multi-Headed Pin

A *multi-headed pin* is a pin on any circuit block from where more than one connection goes outward. We can replace a multi-headed pin by its equivalent multi-terminal net. Once we get a multi-terminal net, the rest of the process is same as in Section 3.2.3. This process is shown in Figure 3.12.

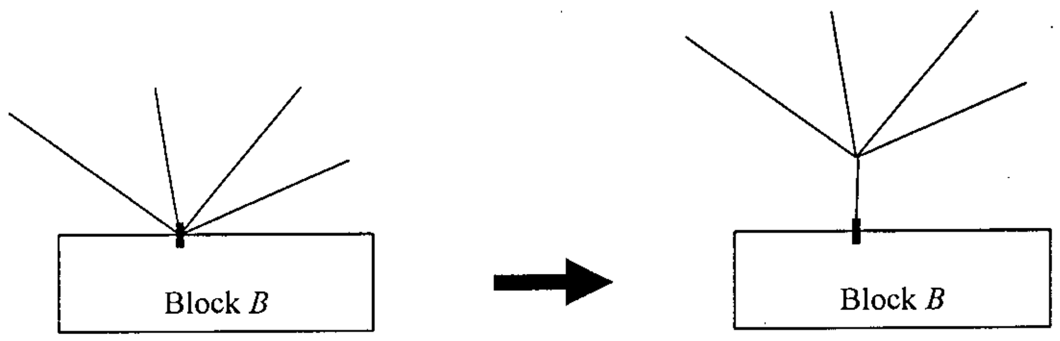


Figure 3.12: Replacing multi-headed pin by a multi-terminal net.

3.2.5 Orthogonal Drawing of a Plane 3-Graph

In this section, we describe how to get an orthogonal drawing for the plane 3-graph as obtained for the circuit specification using the techniques described in Sections 3.2.1, 3.2.2, 3.2.3 and 3.2.4. If G is a plane graph with maximum degree $\Delta \leq 3$, then there is a linear time algorithm to find bend minimum orthogonal drawing of G [24, 25, 27]. We have the following theorem for the orthogonal graph drawing of a plane 3-graph [27]:

Theorem 3.1: *Let G be a connected plane 3-graph and let H be a biconnected component of G. Then there is a bend minimum orthogonal drawing D(H) for H. There is a linear time algorithm to find such orthogonal drawing.*

In the rest of this section, we describe some preliminaries and the approach of getting orthogonal drawing for a plane 3-graph using the results described in [26, 27]. Let C be a cycle in G and let v be a cut vertex of G on C. We call v an *out-cut vertex* for C if v is a leg vertex of C in G; otherwise we call v an *in-cut vertex* for C (Figure 3.13). Any in-cut vertex for C is not a convex corner (having interior angle 90°) of the drawing of C in any orthogonal drawing of G. Similarly, any out-cut vertex of C is not a concave corner (having interior angle 270°) [26].

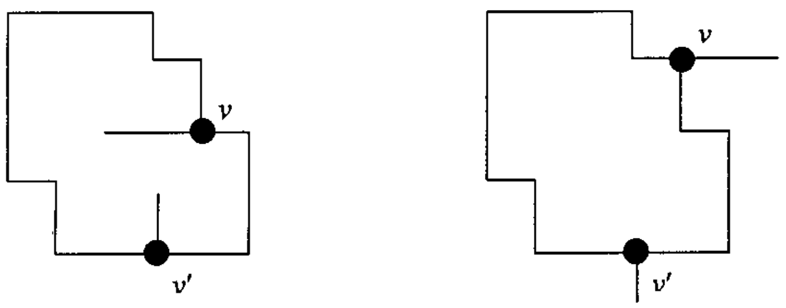


Figure 3.13: (a) In-cut vertices v at a concave corner and v'at a non-corner, (b) Out-cut vertices v at a convex corner and v'at a non-corner.

The plane 3-graph G represents the circuit specification is not necessarily biconnected. So it can be decomposed into a set of biconnected and weakly biconnected components. There is a minimum bend orthogonal drawing for each of the biconnected components using the algorithm in [27]. The orthogonal drawing of a weakly biconnected component is either a horizontal line segment or a vertical line segment. In order to get the orthogonal drawing for the whole plane 3-graph G, we need to merge all the orthogonal drawings of the (weakly) biconnected components.

We construct a BC-tree of G to merge the orthogonal drawings of individual biconnected components. Let B_0, B_1, ..., B_b be the ordering of the blocks following the depth-first search starting from the root B_0 of the BC-tree. Assume that we have already obtained an orthogonal drawing D_i by merging the orthogonal drawings of the blocks B_0, B_1, ..., B_i. Now we are going to obtain an orthogonal drawing D_{i+1}, by merging D_i with the orthogonal drawing of the block B_{i+1}. Let v_t be the cut-vertex corresponding to the C-node which is the parent of B_{i+1} in the BC-tree of G. Let block B_x be the parent of v_t in the BC-tree. Then both B_x and B_{i+1} contain the vertex v_t, and D_i contains the drawing of B_x. We have the following three cases to consider [26]:

Case A: *B_x is a bi-connected component and B_{i+1} is a weakly bi-connected component.*

In this case v_t is either a non-corner or a concave corner. B_{i+1} is an edge and will be drawn inside an inner face of the drawing D_i as shown in the Figure 3.14.

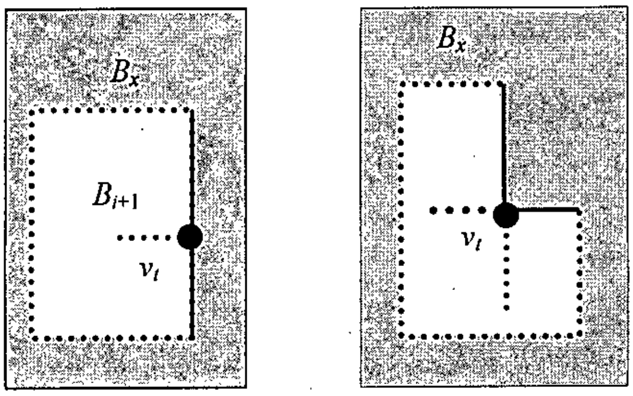


Figure3.14: Embedding of v_t when B_x is a biconnected component and B_{i+1} is a weakly biconnected component.

Case B: *Both B_x and B_{i+1} are weakly bi-connected components.*

In this case v_t is drawn in an inner face of D_i and has degree 1 or 2 in D_i. We first consider the case where v_t has degree one. Then v_t in D_i has the embedding in Figure 3.15(a) or a rotated one. We draw B_i as the dotted line in Figure 3.15(a). We next consider the case where v_l has degree two in D_i. Then v_l has degree 3 in G and let x, y, and z be the three neighbours of v_l in G. We may assume without loss of generality that edges (v_l, x) and (v_l, y) are already drawn in D_i and we now merge the drawing of the edge $(v_l, z)= B_{i+1}$ to D_i. It is evident from the drawing described above that (v_l, x) and (v_l, y) are drawn on a (horizontal or vertical) straight line segment. We draw the edge (v_l, z) as a dotted line as in Figure 3.15(b).

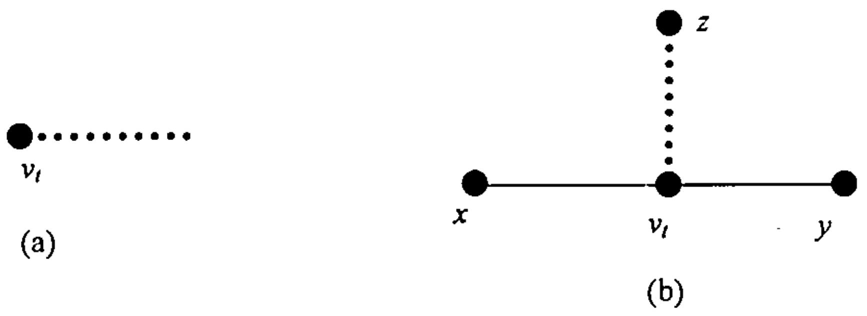


Figure 3.15: Embeddings of v_l when both B_x and B_{i+1} are weakly biconnected components.

Case C: *B_x is a weakly biconnected component and B_{i+1} is a biconnected component.*

In this case v_l is drawn in D_i as the end of a horizontal or vertical line segment inside an inner face of D_i. Vertex v_l has degree 2 in B_{i+1} and is an out-cut vertex for $C_o(B_{i+1})$ and v_l is either a non-corner or a convex corner in $D(B_{i+1})$. Therefore $D(B_{i+1})$ can be easily merged with D_i by rotating $D(B_{i+1})$ by 90° or 180° or 270° and expanding the drawing D_i if necessary.

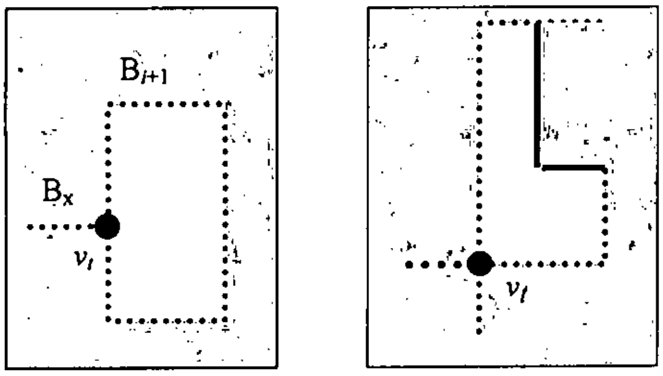


Figure 3.16: Embeddings of v_l when B_x is a weakly biconnected component and B_{i+1} is a biconnected component.

The overall process of getting bend minimum orthogonal drawing of a plane 3-graph is shown in figure 3.17.

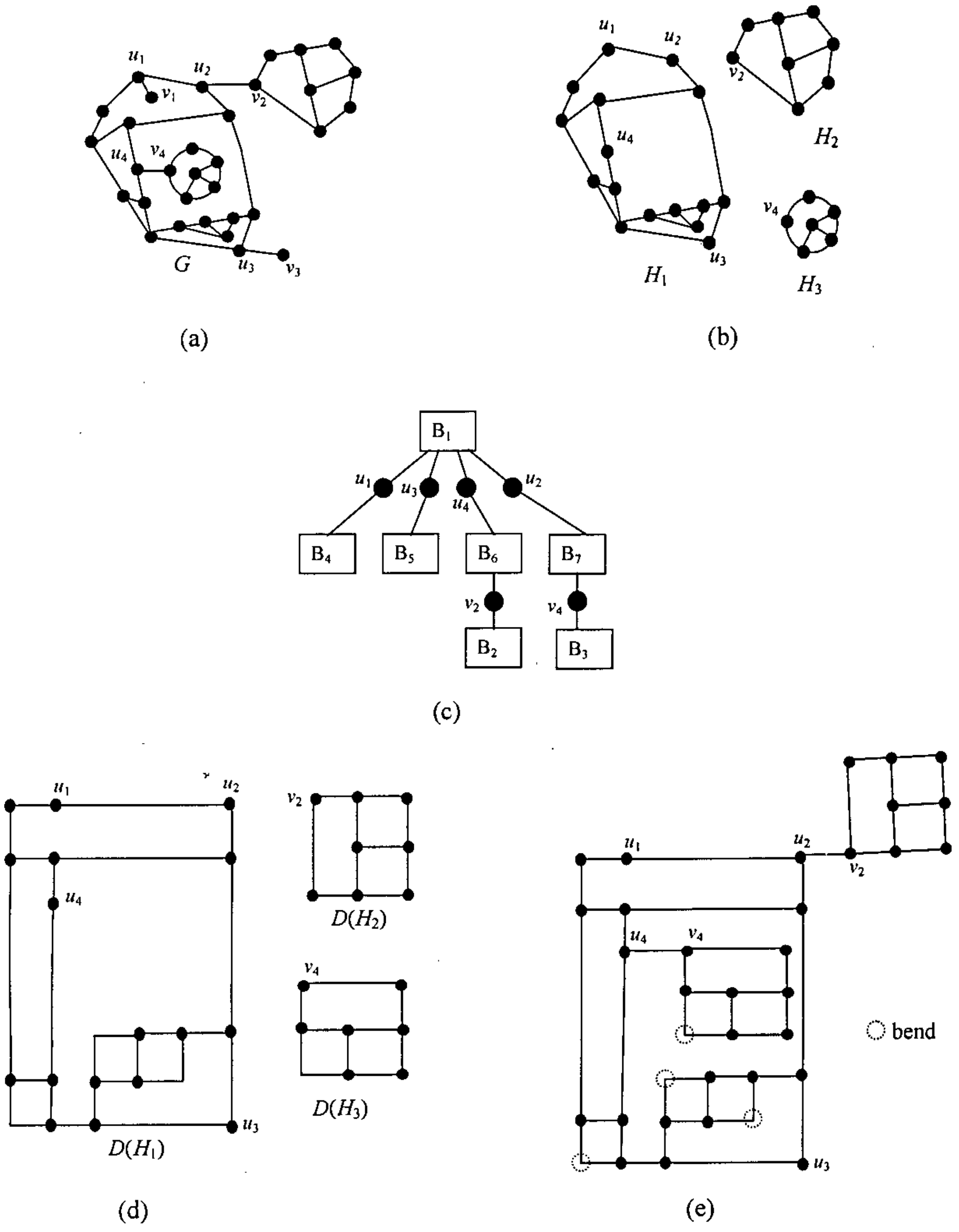


Figure 3.17: (a) A plane 3-graph G, (b) Three biconnected components H_1, H_2 and H_3, (c) BC-tree for G, (d) Minimum bend orthogonal drawing of H_1, H_2 and H_3, (e) Orthogonal drawing of G.

3.2.6 The Overall Process of Getting Floorplan and Global Routing

Orthogonal drawing *D* of plane 3-graph *G′* almost give the floorplan and global routing of the circuit leaving very little processing undone. Each of the cycle of the plane 3-graph *G′* that represents each circuit block or a multi-terminal net's common point is drawn as a rectangle in the orthogonal drawing. We replace each rectangular box in orthogonal drawing *D* that corresponds to the common point of a multi-terminal net by a single point. The overall process of getting floorplan and global routing for a given circuit specification is shown in Figure 3.18.

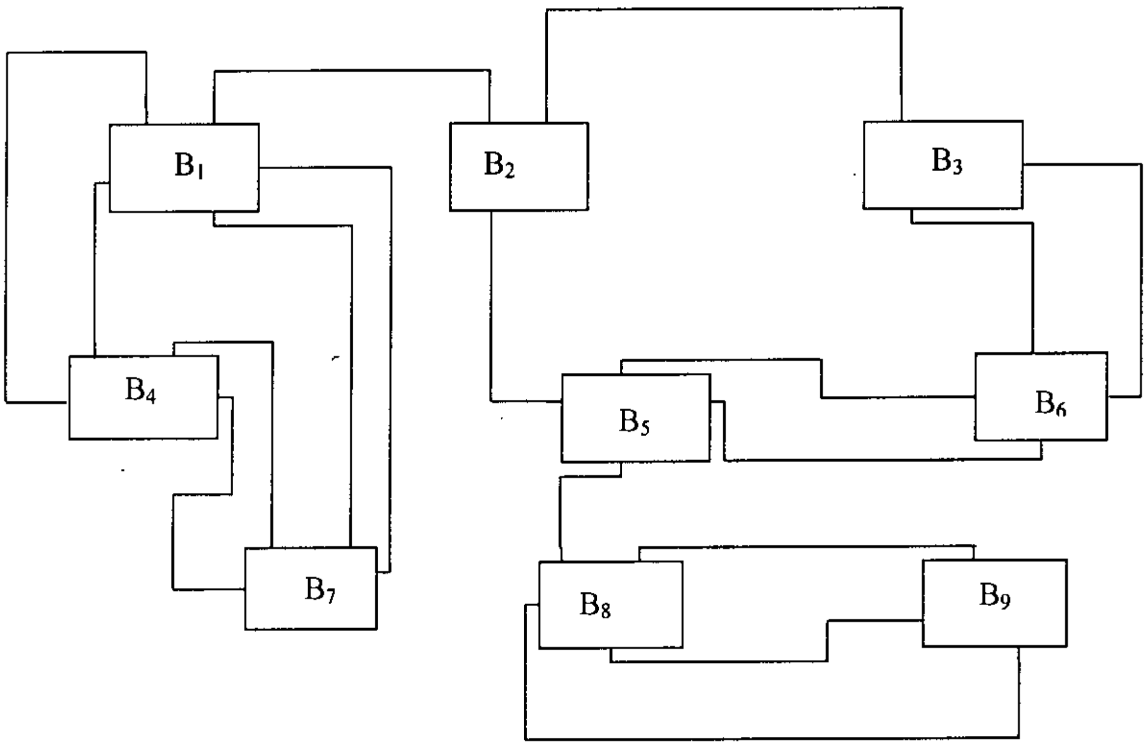


Figure 3.18: (a) A circuit specification.

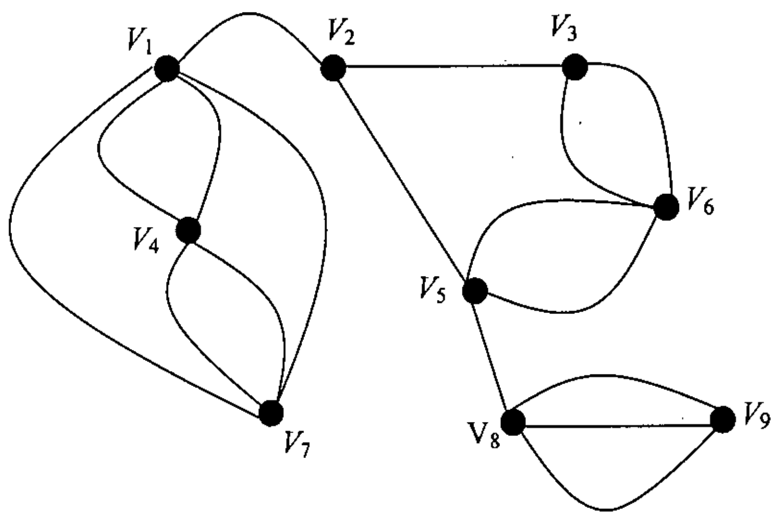


Figure 3.18: (b) Multigraph G for the circuit.

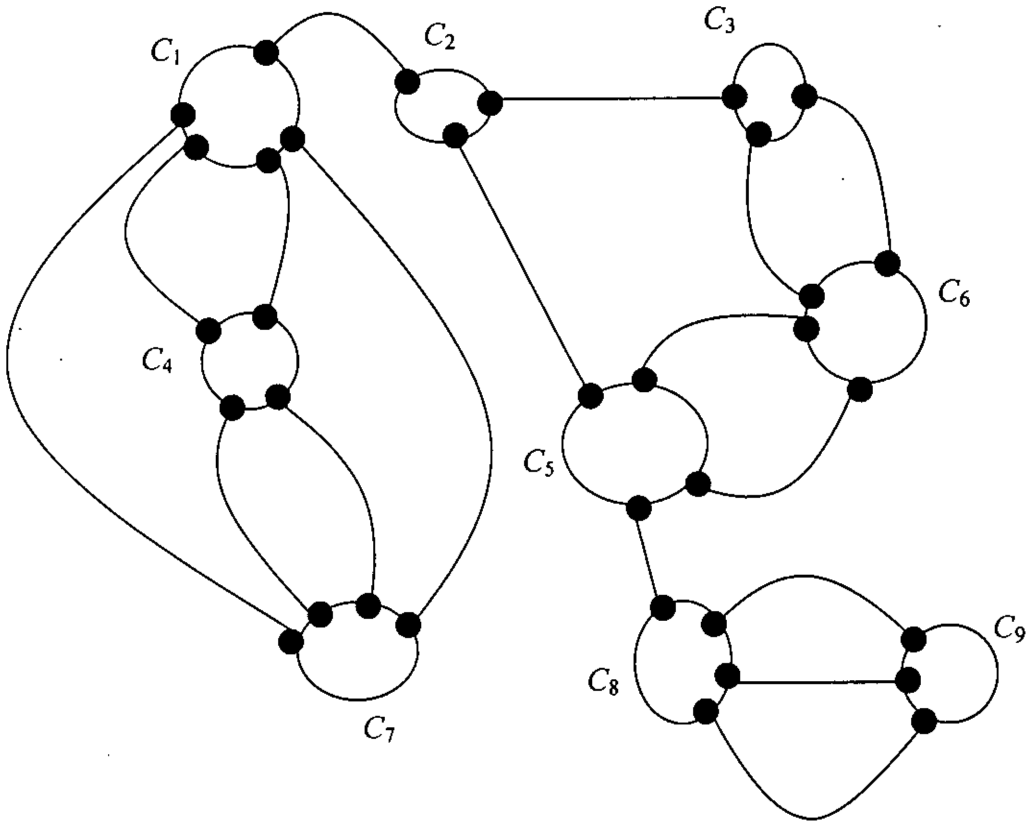


Figure 3.18: (c) A 3-graph G'.

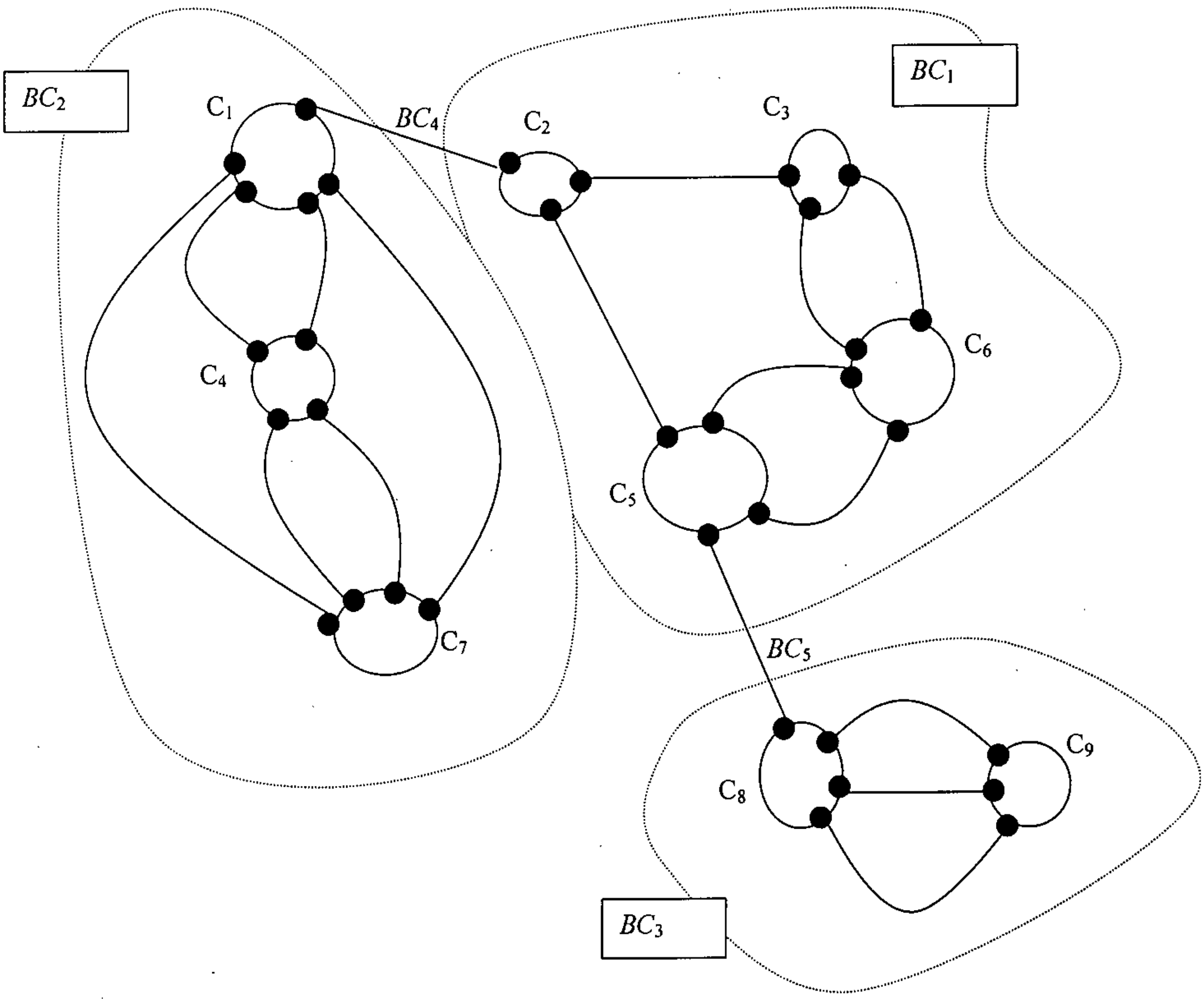


Figure 3.18: (d) Biconnected components BC_1, BC_2, BC_3 and weakly bi-connected components BC_4 and BC_5.

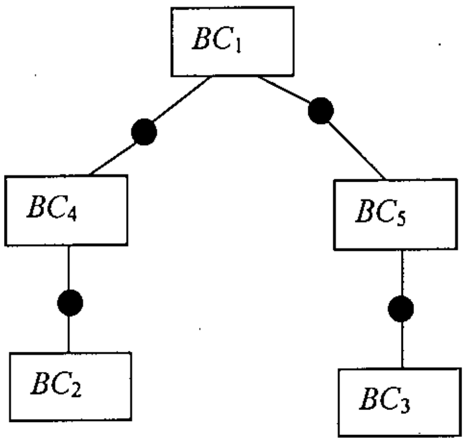


Figure 3.18: (e) BC-tree for the 3-graph G'.

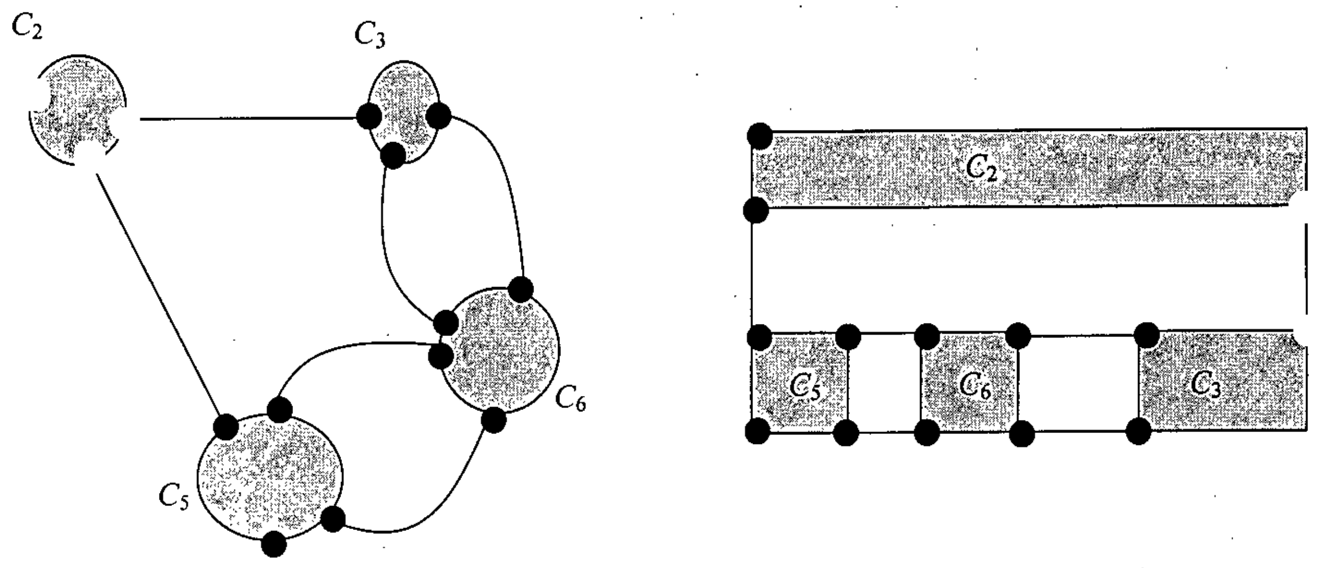


Figure 3.18: (f) Orthogonal drawing of biconnected component BC_1.

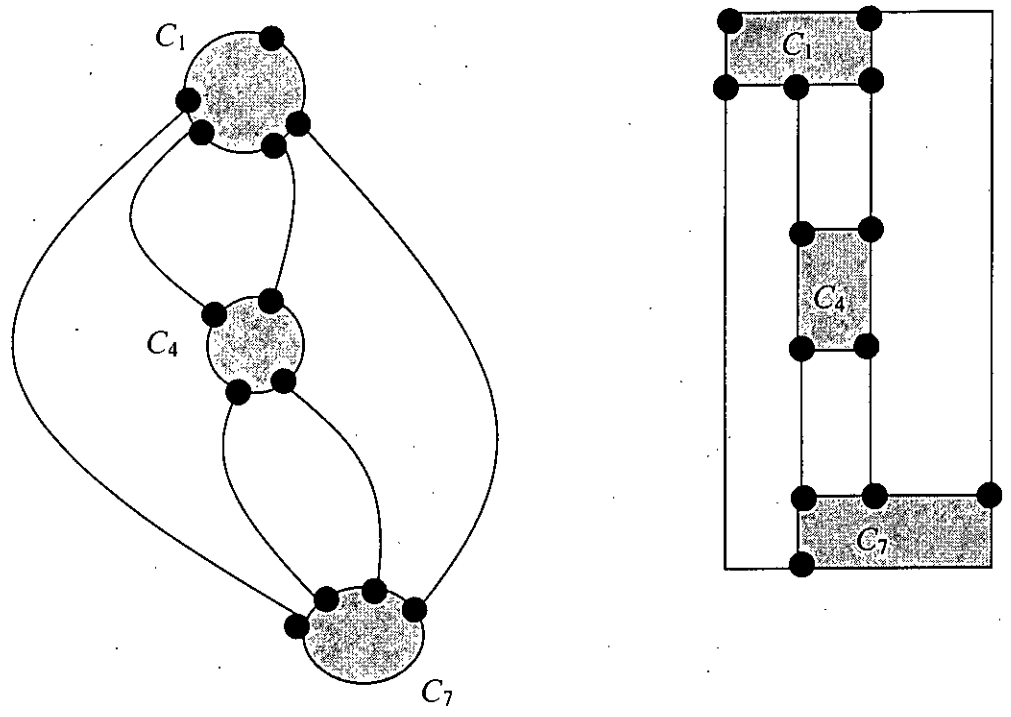
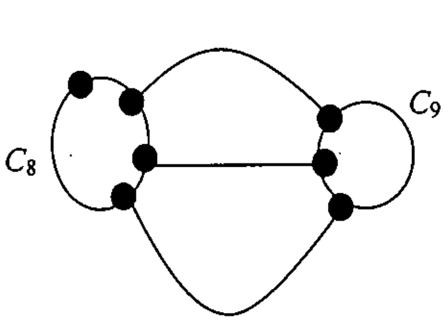


Figure 3.18: (g) Orthogonal drawing of biconnected component BC_2.



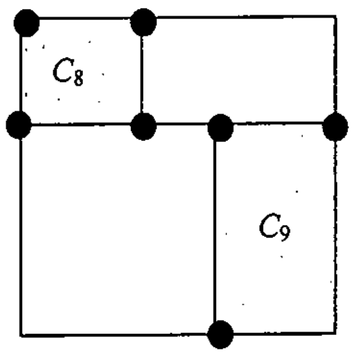


Figure 3.18: (h) Orthogonal drawing of biconnected component BC_3.

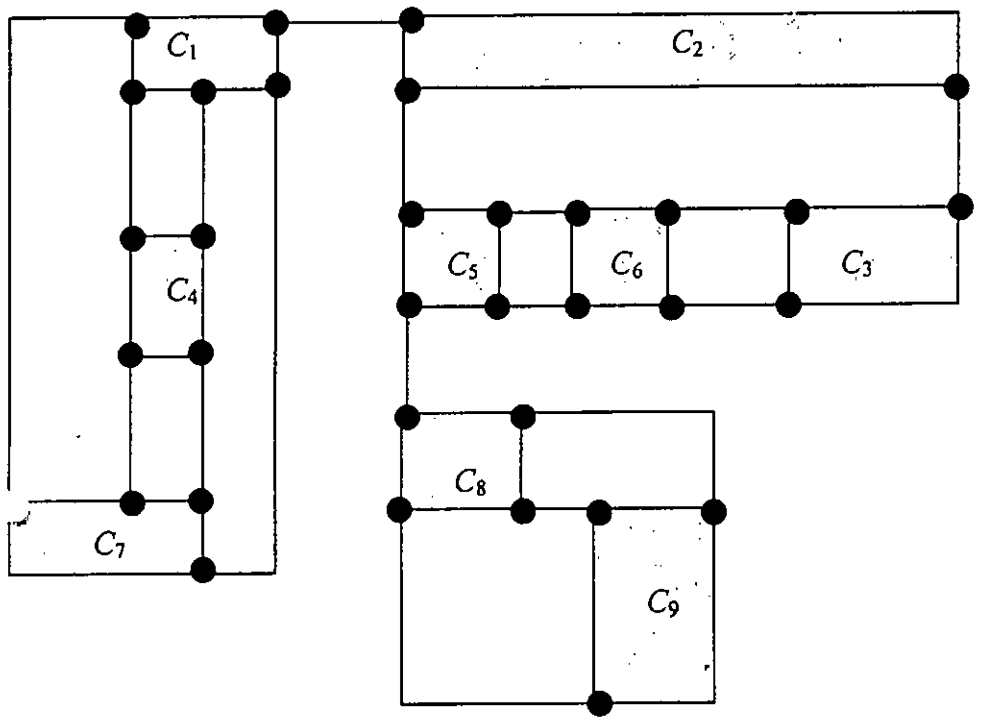


Figure 3.18: (i) Orthogonal drawing for the 3-graph G'.

So we can describe the floorplanning and global routing process using orthogonal graph drawing for a given circuit specification in the following manner:

```
Algorithm FloorPlanning( Circuit $C$)
begin
  Step 1 Represent the circuit $C$ using a multigraph $G$.
          each block (or multi-terminal net's common point) will be represented by a
          vertex and interconnections between blocks by edges;

  Step 2 Convert $G$ into a 3-graph $G'$.
          replace each vertex with degree three or more with a cycle containing the
          number of vertices on the cycle equal to the degree of the original vertex;

          the clockwise orientation of the connecting pins on the circuit element and its
          equivalent vertices on the corresponding cycle in the 3-graph $G'$ should be the
          same;

  Step 3 for each biconnected component $H$ (not weakly bi-connected component) of the
        3-graph $G'$ do
          find a bend minimum orthogonal drawing using algorithm in [27];

  Step 4 Develop a BC-tree $T$ for the 3-graph $G'$.

  Step 5 Visit BC-tree $T$ in depth-first-search order and combine the orthogonal drawings
        of the biconnected components obtained in Step3 with the drawing of weakly
        biconnected components those are either a horizontal or a vertical line segment.
        Thus get an orthogonal drawing $D$ for the 3-graph $G'$.

  Step 6 Replace the rectangular areas obtained for each cycle of 3-graph $G'$ with circuit
        block except rectangular box for those cycles which corresponds to the common
        point of multi-terminal nets.

end
```

3.3 Time Complexity of the Integrated Floorplanning and Global Routing Algorithm

In this section we find the time complexity of the floorplanning and global routing algorithm.

The input to the floorplanning and global routing algorithm is a circuit specification and output is its floorplan together with global routing. The entire process is accomplished with the following phases one after another:

i) To represent the circuit specification using a multigraph G.

ii) To convert the multigraph G into 3-graph G'.

iii) To get orthogonal drawing for each individual biconnected component.

iv) To merge the orthogonal drawings of the biconnected components and hence get the floorplan and global routing.

Thus the cost of the algorithm is the summation of the costs of the above mentioned phases. Here we will define the problem size as N, where N is the number of nets (interconnections between the blocks in the circuit specification).

Let, B_1, B_2, ..., B_b be the blocks after the partitioning of the circuit and $C(B_i)$ be the number of connections going outward of the block B_i, where $1 \leq I \leq b$. A net is made up of two terminal connections. Then

$$\Sigma_{1 \leq i \leq b} C(B_i) = 2N.$$

Let G be the multigraph that represents the circuit and V_1, V_2, ...,V_b are the set of vertices. Then

$$d(V_i) = C(B_i),\ \ 1 \leq i \leq b, \text{ and}$$
$$\Sigma_{1 \leq i \leq b}\, d(V_i) = 2N.$$

In the 3-graph G', each vertex V_i of graph G is replaced by a cycle C_i with $d(V_i)$ vertices, where $d(V_i)$ is the degree of the vertex V_i. So the number of vertices in the 3-graph G' is

$$\Sigma_{1 \leq i \leq b}\, d(V_i) = 2N.$$

The algorithm in [27] finds orthogonal drawing of a biconnected graph in linear time. So the cost of finding orthogonal drawing of the bi-connected component H_i with $O(|V(H_i)|)$ vertices is also $O(|V(H_i)|)$. Thus the total cost of getting orthogonal drawing of all the biconnected components is

$$\Sigma_{1 \leq i \leq k} O(|V(H_i)|) = O(\Sigma_{1 \leq i \leq k} |V(H_i)|) = O(N).$$

Merging of an orthogonal drawing of a biconnected components with the drawing of a weakly biconnected component (a horizontal or vertical line segment) can be done in $O(1)$ time. So merging all the drawings of the biconnected components can be done in $O(N)$ times.

Thus the overall complexity of getting the floorplan and global routing of a circuit with N nets is $O(N)$.

Chapter 4

Detailed Routing

In this chapter we present a linear-time algorithm for detailed routing with emphasis on minimizing the routing distance. In Section 1 we discuss on preliminaries for routing algorithm. Then in Section 2 we present the detailed routing algorithm. In Section 3 we find the time complexity of the algorithm.

4.1 Introduction

Given a floorplan of the circuit and a global routing of the nets as obtained in Chapter 3, we are required to bend the nets so that they connect at the actual pin positions of the circuit elements (or blocks). This operation is called *detailed routing* and the drawing thus obtained after detailed routing is called *routed layout* for the input circuit specification that will later be fabricated in the next phase of a VLSI design cycle.

In Chapter 3, we have seen how floor planning of a circuit can be obtained together with the global routing of the nets using the orthogonal graph drawing. If a vertex v has degree $d(v)$ in the multigraph G representing the circuit specification (see Section 3.2.1), then after floorplanning we will get a box corresponding to the vertex v with same number of connections to other circuit elements and each connections on a box is a pin on that circuit element. During graph representation of the circuit, connection on each vertex of graph obtained from circuit specification are considered clock wise. After floorplanning those connections maintain same clockwise ordering, but the exact pin positions on the actual circuit elements (or blocks) and the positions of the corresponding vertices on the floorplan may not be the same. That is why, the edges connected to the boxes need to bend to position exactly on the pin positions of the circuit elements.

4.2 Detailed Routing Algorithm

In VLSI design, timing consideration is one of the most critical factors and it is very much related to the length of the nets those connect the circuit elements. So routing algorithm should set its sight on minimizing the net length during bending the nets to fit to the actual pin positions of the circuit elements. A net in orthogonal drawing that subject to route to actual pin position, can either be directed clockwise or anti-clockwise to get there. But we should choose the path that ensures minimum distance of routing. In our algorithm, we classify vertices into two classes: *left vertices* and *right vertices* and has been described in Section 4.2.1.

During detailed routing, there may have a number of nets across a particular position and no two nets should intersect each other. In our algorithm, we associate a tuple of three values with each terminal vertex of the net, one for each side of the block that the net should move around, to ensure that nets are not unnecessarily farther from the side lines of the block. The process of determining the 3-tuple for each vertex has been described in Section 4.2.2.

In Section 4.2.3, the way the nets are to be bended has been described. The outline of the detailed routing algorithm is as follows:

```
Algorithm DetailedRouting (Floorplan F)
begin
  Step 1 Classify each vertices into left-vertex or right-vertex;

  Step 2 Determine a 3-tuple for each vertex that would later guide the net to move from
  its vertex position to pin position;

  Step 4 Minimize number tuple for each vertex;

  Step 3 Bend each net according to its leftness/rightness and tuples associated with its
  two terminal vertices;
end
```

4.2.1 Classifying vertices into Left and Right

Depending upon the relative position of vertex (on the box of floorplan) and their exact pin position (on the circuit element) each vertex is classified as either left vertex or right vertex.

Let, the left-top and bottom-right corner of a box (e.g. each circuit element) in floorplan be (x_1, y_1) and (x_2, y_2) respectively. We assume that, X coordinate is increasing horizontally from left to right and Y coordinate is increasing vertically from top to bottom. There are four side lines for each box. Those are top horizontal line L_1, right vertical line L_2, bottom horizontal line L_3 and left vertical line L_4. Let, $P(x_p, y_p)$ be the exact pin position of the vertex positioned at $V(x_v, y_v)$ on box of the floorplan. Then there are following three cases:

Case 1 If $P(x_p, y_p)$ and $V(x_v, y_v)$ are on the same line (L_1 or L_2 or L_3 or L_4) then vertices can be classified into left and right vertex in the following manner:

On L_1: When pin position $P(x_p, y_p)$ and its corresponding vertex position $V(x_v, y_v)$ both are on line L_1 then:

```
if $x_p \geq x_v$ then
    vertex $V(x_v, y_v)$ is a right vertex;
else
    vertex $V(x_v, y_v)$ is a left vertex;
```

On L_2: When pin position $P(x_p, y_p)$ and vertex position $V(x_v, y_v)$ both are on line L_2 then:

```
if $y_p \geq y_v$ then
    vertex $V(x_v, y_v)$ is a right vertex;
else
    vertex $V(x_v, y_v)$ is a left vertex;
```

On L_3: When pin position $P(x_p, y_p)$ and vertex position $V(x_v, y_v)$ both are on line L_3 then:

```
if $x_p \geq x_v$ then
    vertex $V(x_v, y_v)$ is a left vertex;
    else
        vertex $V(x_v, y_v)$ is a right vertex;
```

On L_4: When pin position $P(x_p, y_p)$ and vertex position $V(x_v, y_v)$ both are on line L_2 then:

```
    if $y_p \geq y_v$ then
        vertex $V(x_v, y_v)$ is a left vertex;
    else
        vertex $V(x_v, y_v)$ is a right vertex;
```

Case-1 classification is shown in Figure 4.1. Here a small bold circle on the side line represents a vertex position and a small horizontal or vertical line is a pin position on the box. In Figure 4.1, v_1, v_3, v_5 and v_7 are right vertices and v_2, v_4, v_6 and v_8 are left vertices.

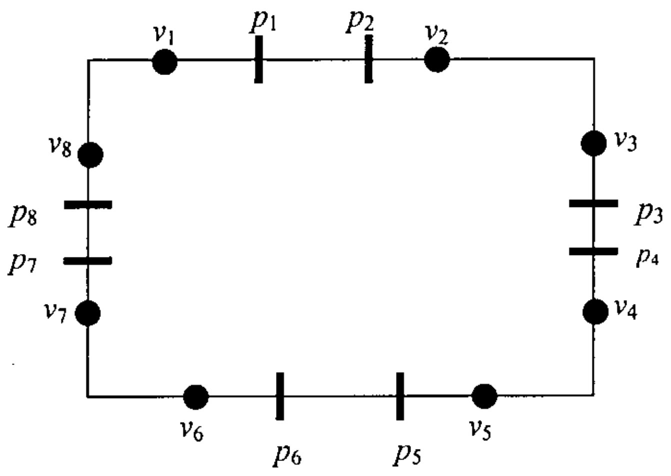


Figure 4.1: Classification of vertices (Case-1).

Case 2 If pin position $P(x_p, y_p)$ and vertex position $V(x_v, y_v)$ are on two neighboring side lines then vertices can be classified into left and right vertex in the following manner:

```
    if $V$ is on $L_1$ and $P$ is on $L_2$ then
        $V$ is a right vertex;
    elseif $V$ is on $L_1$ and $P$ is on $L_4$ then
        $V$ is a left vertex;
    elseif $V$ is on $L_2$ and $P$ is on $L_3$ then
        $V$ is a right vertex;
```

```
elseif $V$ is on $L_2$ and $P$ is on $L_1$ then
    $V$ is a left vertex;
elseif $V$ is on $L_3$ and $P$ is on $L_4$ then
    $V$ is a right vertex;
elseif $V$ is on $L_3$ and $P$ is on $L_2$ then
    $V$ is a left vertex;
elseif $V$ is on $L_4$ and $P$ is on $L_1$ then
    $V$ is a right vertex;
elseif $V$ is on $L_4$ and $P$ is on $L_3$ then
    $V$ is a left vertex;
```

Case-2 classification is shown in Figure 4.2. Here, v_1, v_2, v_3 and v_4 are right vertices and v_5, v_6, v_7 and v_8 are left vertices.

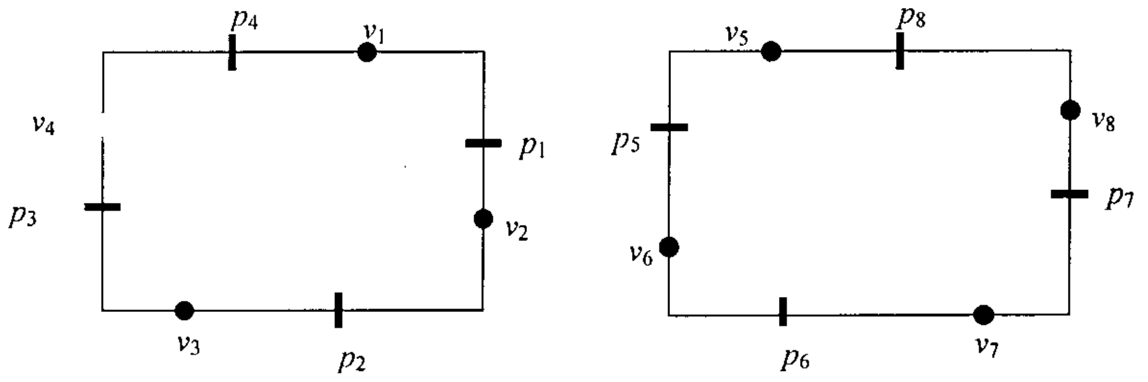


Figure 4.2: Classification of vertices (Case-2).

Case 3 If pin position $P(x_p, y_p)$ and vertex position $V(x_v, y_v)$ are on two opposite side lines then vertices can be classified into left and right vertex in the following manner:

```
(A) if $V$ is on $L1$ and $P$ is on $L_3$ then
        if $x_p-x_1+x_v-x_1 > x_2-x_1$ i.e. $x_p+x_v > x_1+x_2$ then
            $V$ is right vertex;
        else
            $V$ is left vertex;
(B) if $V$ is on $L_3$ and $P$ is on $L_1$ then
        if $x_p-x_1+x_v-x_1 > x_2-x_1$ i.e. $x_p+x_v > x_1+x_2$ then
            $V$ is left vertex;
        else
        V is right vertex;

(C) if V is on $L_2$ and P is on $L_4$ then
        if $y_p$-$y_1$+$y_v$-$y_1$ > $y_2$-$y_1$ i.e. $y_p+y_v > y_1+y_2$ then
            V is right vertex;
        else
            V is left vertex;

(D) if V is on $L_4$ and P is on $L_2$ then
        if $y_p$-$y_1$+$y_v$-$y_1$ > $y_2$-$y_1$ i.e. $y_p+y_v > y_1+y_2$ then
            V is left vertex;
        else
            V is right vertex;
```

Case-3 classification is shown in Figure 4.3. Here, v_1, v_2, v_3 and v_4 are right vertices and v_5, v_6, v_7 and v_8 are left vertices.

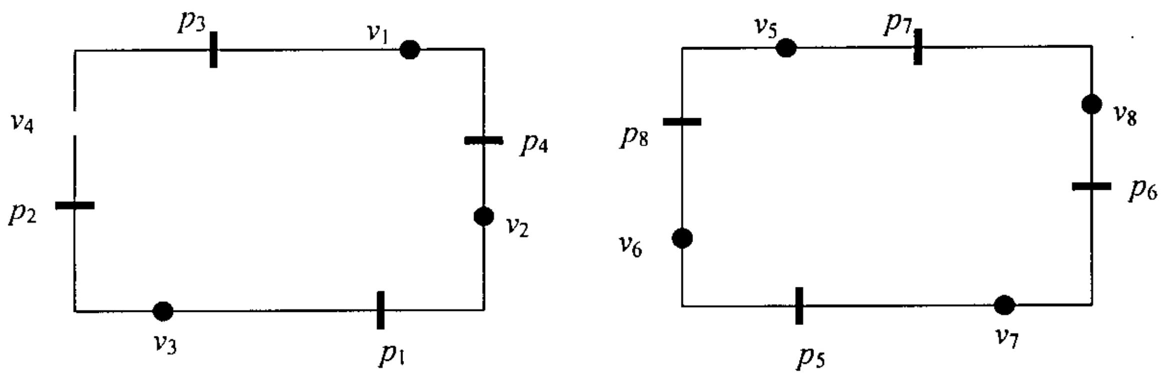


Figure 4.3: Classification of vertices (Case-3).

All three cases of vertex classification are illustrated in Figure 4.4. Here L or R in the parenthesis with each vertex indicates whether the vertex is left or right vertex.

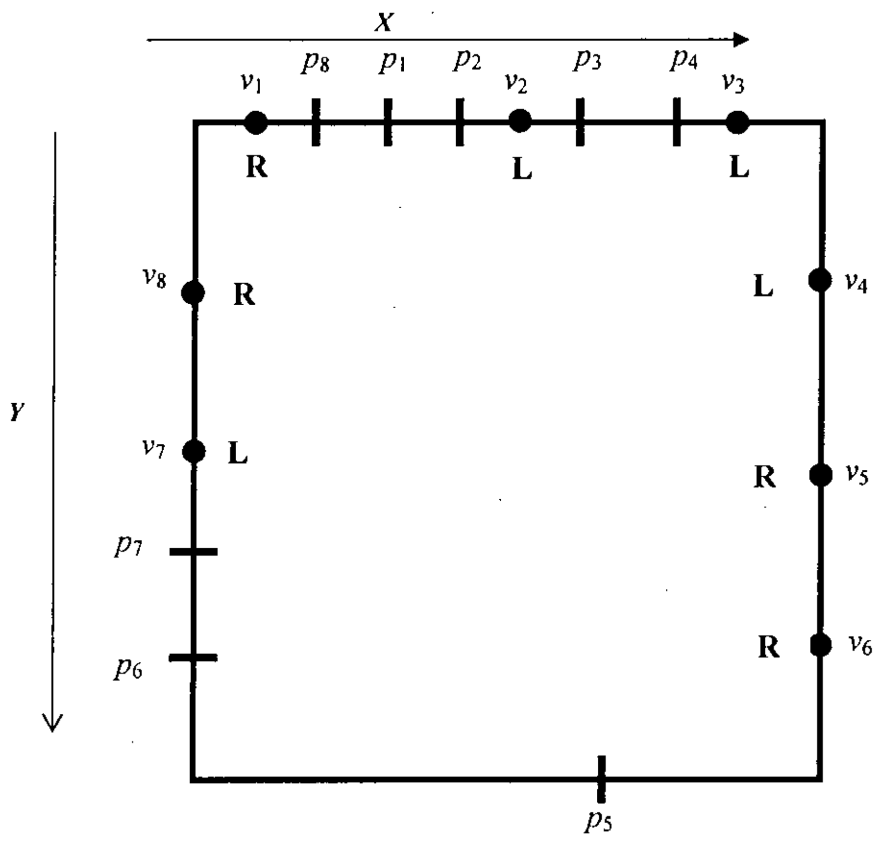


Figure 4.4: Illustration of vertex classification.

4.2.2 Finding Routing Parameters

After classifying the vertices into left and right kinds, we are left with the task of determining the number of grids a net to be away from the side lines of the block to reach from the vertex position to the corresponding actual pin position of the block. We have seen in Section 4.2.1 that a vertex and its corresponding pin may be situated on the same line or on two neighboring lines or on two opposite lines. If a vertex and its actual pin are on the same line then during detailed routing the net needs to move along only one side line of the block to reach from the vertex position to the pin position. A net needs to move along two neighboring side lines of the block if vertex and its corresponding pin are on two neighboring side lines of the block. In the worst case, net needs to move along three neighboring side lines if vertex and its corresponding pin are on two opposite side lines of the block. So we are required to determine a tuple of three values namely (*S*, *N*, *O*); where *S* represents the number

of grids the net need to be away from the line on which the vertex is situated during moving towards the actual pin. *N* and *O* are both zero if the vertex and its actual pin are on the same line of the block. *N* is the number of grid lines the net to be away from the neighboring side line during its move from the vertex to actual pin; if vertex and actual pin is either on neighboring side lines or on opposite side lines of the block. The value of *O* is nonzero only if vertex and actual pin are on two opposite side lines of the block and then it represents the number of grids the net should be away from the opposite side lines during its move from the vertex to actual pin. Figure 4.5 illustrates the (*S*, *N*, *O*) tuple.

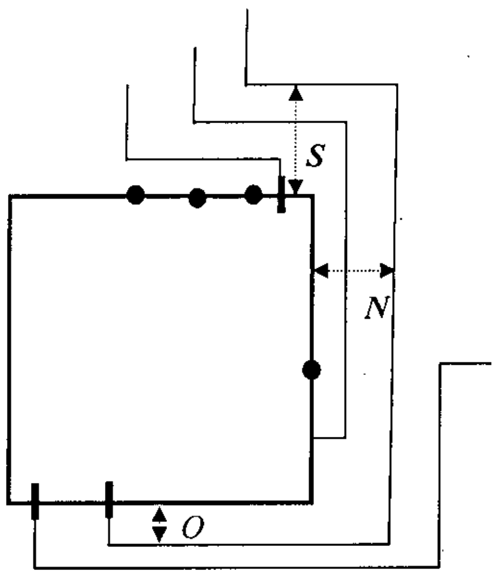


Figure 4.5: Illustration of (*S*, *N*, *O*) tuple.

The determination of the routing parameter (*S*, *N*, *O*) for each vertex of each block of the floorplan is done with emphasis on minimizing the total route. In order to determine the tuples for all vertices of a block or circuit element, we need to move twice around the block. First move is clock wise and it determines tuples for the right vertices only and it is accomplished by the algorithm ***InitialRightRoute*(*B*)** for block *B*. Next move is anti-clock wise to determine number tuples for the left vertices and is done by the algorithm **InitialLeftRoute(*B*).**

4.2.2.1 Finding Routing Parameters for Right Vertices

The clockwise ordering of vertices obtained in the floorplan is same as the clockwise ordering of the pins of the actual circuit block. Thus during clockwise move around the block, the order of occurrence of the right vertices and its correspondence pin will be the same. That is, it is never be the case that a late coming right vertex reaches its pin earlier than an early coming right vertex during the move. We keep the right vertices in a

QUEUE those have not reach their pins yet. During the move it is sure that the first vertex of the queue reaches its pin first thus needs to go away only one grid away from the side lines of the block and next vertex in the queue reaches its pin next and needs to go away two grids away from the side lines and so on. We use a right grid counter (***RGC***) in our algorithm. When a vertex reaches its pin we delete that vertex from the queue and assign number tuple to the vertex based on *RGC* and whether the position of the vertex and pin are on the same line or on the neighboring lines or on the opposite lines of the block. On the clockwise move when we get a left vertex, it is sure that all the pins of the previous right vertices have been reached; because during clockwise move all pins of the early right vertices will be found first before reaching the left vertex. Thus when we encounter a left vertex on this clockwise move we reset *RGC* to zero. Again, when our queue becomes empty; means all the right vertices seen so far has get their pin; we reset the RGC to zero again. The algorithm ***InitialRightRoute(B)*** for block *B* to find routing parameters for right vertices is given below:

```
Algorithm InitialRightRoute(B)
begin
  RGC:=0
  Start from left top corner and find a left vertex by moving clockwise. The next available
  right vertex $u$ will be the starting right vertex.

  Move clockwise around the block starting from $u$ until each right vertex $v$ gets ($S$, $N$,
  $O$) tuple

      if $v$ is a right vertex then
        add $v$ to QUEUE
      else
        RGC:=0
      endif

      if pin $P_f$ for first vertex $v_f$ of QUEUE is reached then
      beginif
        RGC:=RGC+1
        Delete $v_f$ of QUEUE

        if vertex $V_f$ and pin $P_f$ are on same line then
```

```
          Number[$v_f$][$S$]= $RGC$
          Number[$v_f$][$N$]= 0
          Number[$v_f$][$O$]= 0
        **elseif** vertex $v_f$ and pin $P_f$ are on neighboring lines **then**
          Number[$v_f$][$S$]= $RGC$
          Number[$v_f$][$N$]= $RGC$
          Number[$v_f$][$O$]= 0
        **elseif** vertex $v_f$ and pin $P_f$ are on opposite lines **then**
          Number[$v_f$][$S$]= $RGC$
          Number[$v_f$][$N$]= $RGC$
          Number[$v_f$][$O$]= $RGC$
        **endif**
      **endif**
      **if** QUEUE is Empty **then**
        $RGC$:=0
  **endMove**
**end**
```

Figure 4.6 illustrates the use of algorithm *InitialRightRoute*(*B*).

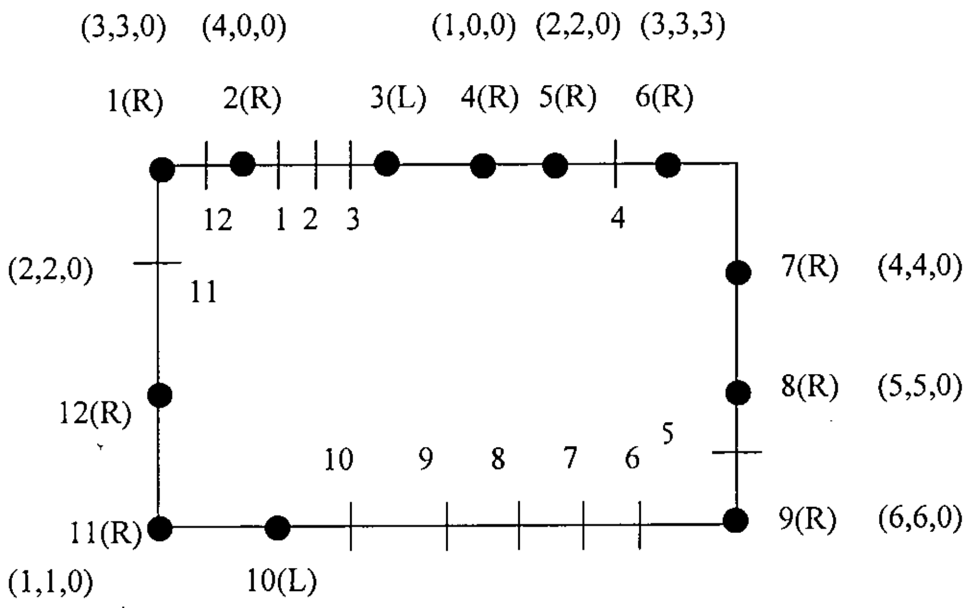


Figure 4.6: Illustration of the *InitialRightRoute*(*B*) algorithm.

4.2.2.2 Finding Routing Parameters for Left Vertices

The anti-clockwise ordering of vertices obtained in the floorplan is same as the anti-clockwise ordering of the pins of the actual circuit block. Thus during anti-clockwise move around the block, the order of occurrence of the left vertices and its correspondence pin will be the same. That is, it is never be the case that a late coming left vertex reaches its pin earlier than an early coming left vertex during the move. We keep the left vertices in a QUEUE those have not reach their pins yet. During the move it is sure that the first vertex of the queue reaches its pin first thus needs to go away only one grid away from the side lines of the block and next vertex in the queue reaches its pin next and needs to go away two grids away from the side lines and so on. We use a left grid counter (***LGC***) in our algorithm. When a vertex reaches its pin we delete that vertex from the queue and assign number tuple to the vertex based on *LGC* and whether the position of the vertex and pin are on the same line or on the neighboring lines or on the opposite lines of the block. On the anti-clockwise move when we get a right vertex, it is sure that all the pins of the previous left vertices have been reached; because during anti-clockwise move all pins of the early left vertices will be found first before reaching the right vertex. Thus when we encounter a right vertex on this anti-clockwise move we reset *LGC* to zero. Again, when our queue becomes empty; means all the left vertices seen so far has get their pin; we reset the *LGC* to zero again. The algorithm ***InitialLeftRoute(B)*** for block *B* to find routing parameters for right vertices is given below:

```
Algorithm InitialLeftRoute(B)
begin
  LGC:=0
  Start from left top corner and find a right vertex by moving clockwise. The next
  available left vertex u will be the starting left vertex.

  Move anti-clockwise around the block starting from u until each left vertex v gets (S,
  N, O) tuple

      if v is a left vertex then
        add v to QUEUE
```

```
    else
      $LGC$:=0
    endif


    if pin $P_f$ for first vertex $v_f$ of QUEUE is reached then
    beginif
      $LGC$:= $LGC$ +1
      Delete $v_f$ of QUEUE
      if vertex $V_f$ and pin $P_f$ are on same line then
        Number[$v_f$][$S$]= $LGC$
        Number[$v_f$][$N$]= 0
        Number[$v_f$][$O$]= 0
      elseif vertex $v_f$ and pin $P_f$ are on neighboring lines then
        Number[$v_f$][$S$]= $LGC$
        Number[$v_f$][$N$]= $LGC$
        Number[$v_f$][$O$]= 0
      elseif vertex $v_f$ and pin $P_f$ are on opposite lines then
        Number[$v_f$][$S$]= $LGC$
        Number[$v_f$][$N$]= $LGC$
        Number[$v_f$][$O$]= $LGC$
      endif
    endif


    if QUEUE is Empty then
      $LGC$:=0
  endMove
end
```

Figure 4.7 illustrates the use of algorithm *InitialLeftRoute*(B).

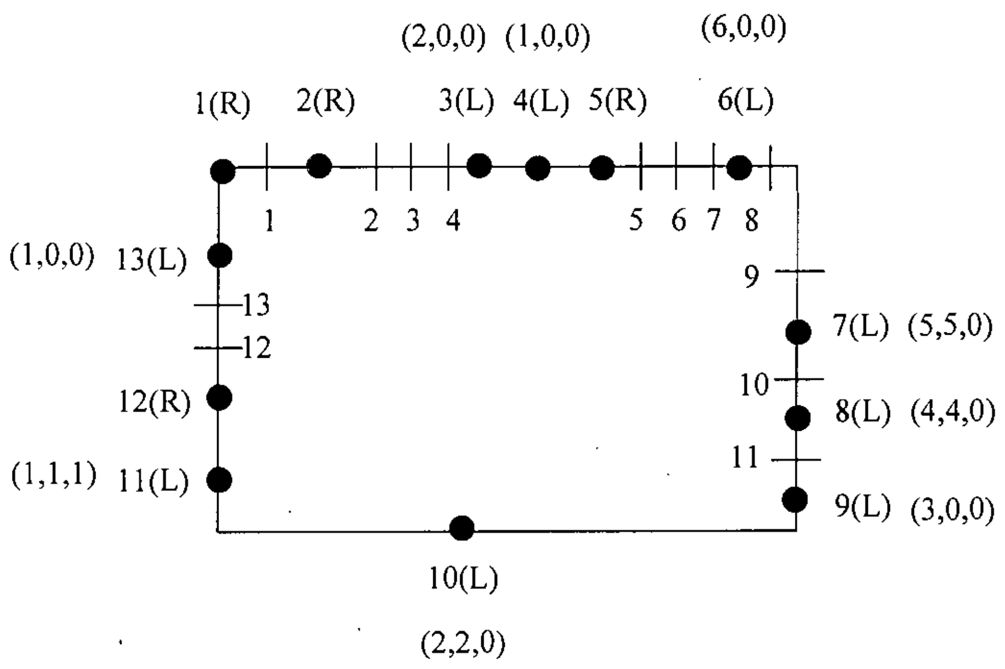


Figure 4.7 Illustration of the InitialLeftRoute (B) algorithm.

4.2.2.3 Minimizing Routing Parameters

In *InitialRightRoute*(*B*) or *InitialLeftRoute*(*B*) algorithm, we have assigned same grid count value (RGC or LGC) to the *S* and *N* if vertex and pin are on the neighboring side lines of the block and same value to *S*, *N* and *O* if vertex and pin are on the opposite side lines of the block. But this is not always necessary. Value of *N* and *O* can be less than that of the value of *S* that means reduction of the number of grids the net should be away from the corresponding side line; thus minimizing the overall layout area. Figure 4.8 illustrates the feasibility of the minimization of routing parameters.

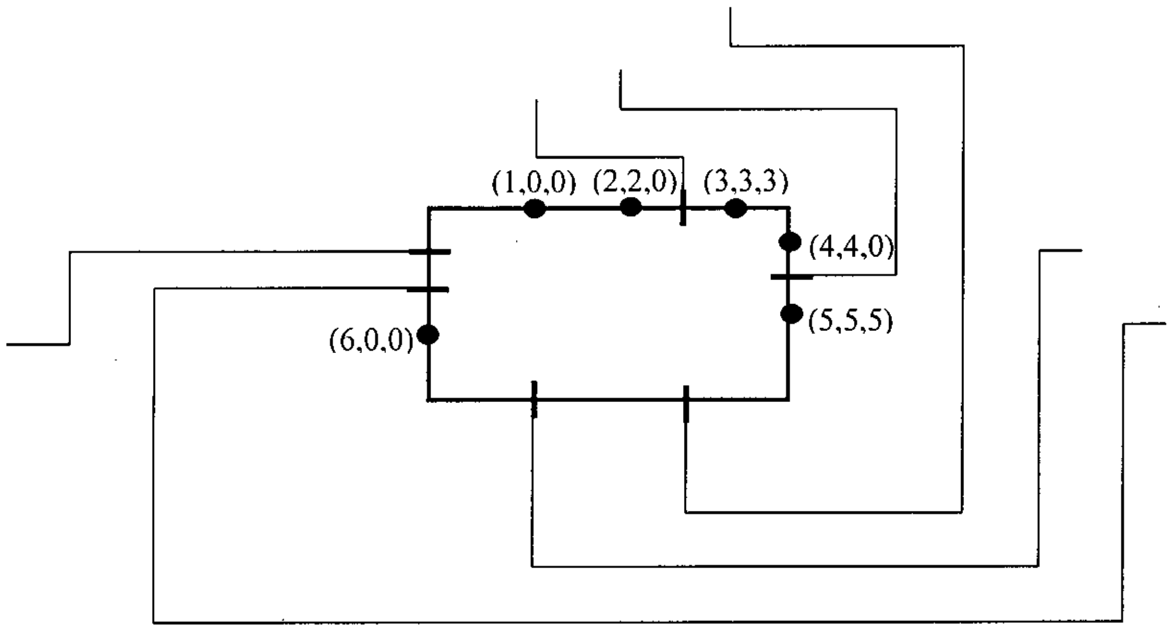


Figure 4.8: (a) Routing parameters before minimization.

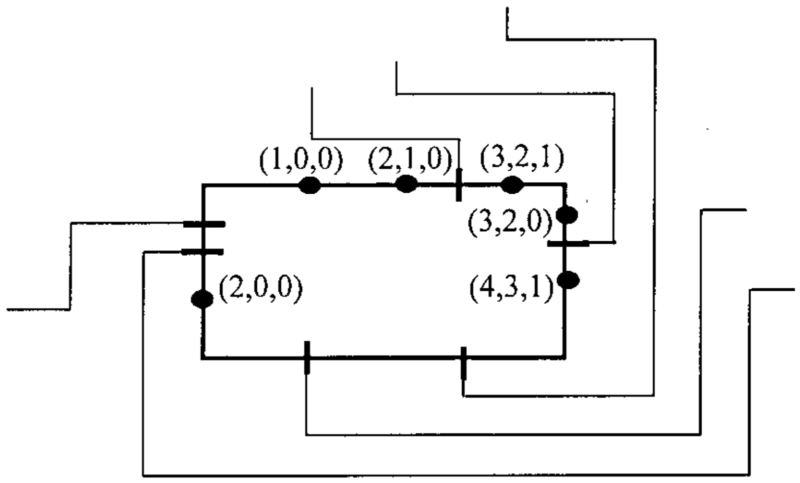


Figure 4.8: (b) Routing parameters after minimization.

In order to minimize routing parameter, we move clockwise for the right vertices. Let v_1 and v_2 are two successive right vertices on a block. Then the following three cases may occur:

Case 1 If v_1 and v_2 are on the same line then Number[v_2][S] will be one greater than Number[v_1][S] and Number[v_2][N] will be one greater than Number[v_1][N] if Number[v_2][N] is non-zero and Number[v_2][O] is one greater than Number[v_1][O] if Number[v_2][O] is non-zero.

Case 2 If v_1 and v_2 are on two neighboring side lines then Number[v_2][S] will be one greater than Number[v_1][N] and Number[v_2][N] will be one greater than

Number[v_1][O] if Number[v_2][N] is non-zero and Number[v_2][O] is one if Number[v_2][O] is non-zero.

Case 3 If v_1 and v_2 are on two opposite side lines then Number[v_2][S] will be one greater than Number[v_1][O] and Number[v_2][N] will be one if Number[v_2][N] is non-zero and Number[v_2][O] is one if Number[v_2][O] is non-zero.

Figure 4.9 shows the minimization of routing parameters for right vertices.

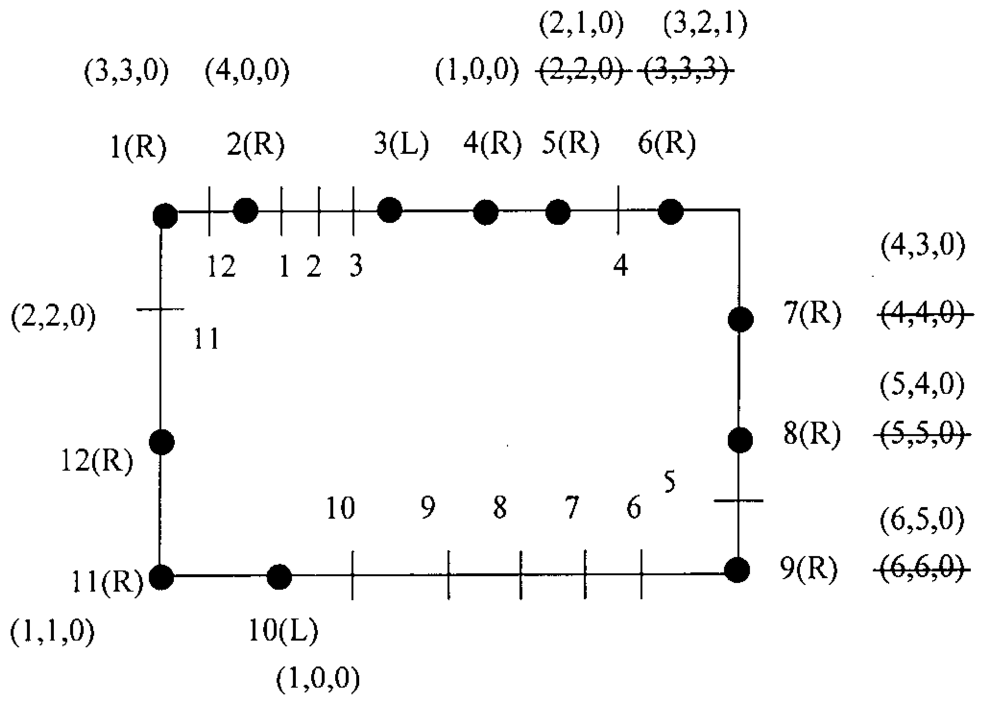


Figure 4.9: Illustration of the *MinimizeRightRoute*(B) algorithm.

The algorithm to minimize the routing parameters for the right vertices is as follows:

```
Algorithm MinimizeRightRoute( B)
begin
  $v_1$ = First Right Vertex as is determined in InitialRightRoute algorithm
  Move clockwise around the block B
    if a Left vertex is reached then
      $v_1$=Next available right vertex
    endif
```

```
        $v_2$ = Next right vertex of $v_1$
        **if** $v_1$ **and** $v_2$ are on the same line **then**
           **if** Number[$v_2$][$N$] > 0 **then**
              Number[$v_2$][$N$] = Number[$v_1$][$N$] +1
           **endif**
           **if** Number[$v_2$][$O$] > 0 **then**
              Number[$v_2$][$O$] = Number[$v_1$][$O$] +1
           **endif**
        **endif**
        **if** $v_1$ **and** $v_2$ are on two neighboring lines **then**
           **if** Number[$v_2$][$S$] > = Number[$v_1$][$N$] +1 **then**
              Number[$v_2$][$S$] = Number[$v_1$][$N$] +1
           **endif**
           **if** Number[$v_2$][$N$] > Number[$v_1$][$O$] +1 **then**
              Number[$v_2$][$N$] = Number[$v_1$][$O$] +1
           **endif**
           **if** Number[$v_2$][$O$] > 0 **then**
              Number[$v_2$][$O$] = 1
           **endif**
        **endif**
        **if** $v_1$ **and** $v_2$ are on two neighboring lines **then**
           **if** Number[$v_2$][$S$] > = Number[$v_1$][$O$] +1 **then**
              Number[$v_2$][$S$] = Number[$v_1$][$O$] +1
           **endif**
           **if** Number[$v_2$][$N$] > 0 **then**
              Number[$v_2$][$N$] = 1
           **endif**
           **if** Number[$v_2$][$O$] > 0 **then**
              Number[$v_2$][$O$] = 1
           **endif**
        **endif**
        $v_1$ = $v_2$
     **endMove**
**end**
```

Similarly, Figure 4.10 shows the minimization of routing parameters for left vertices.

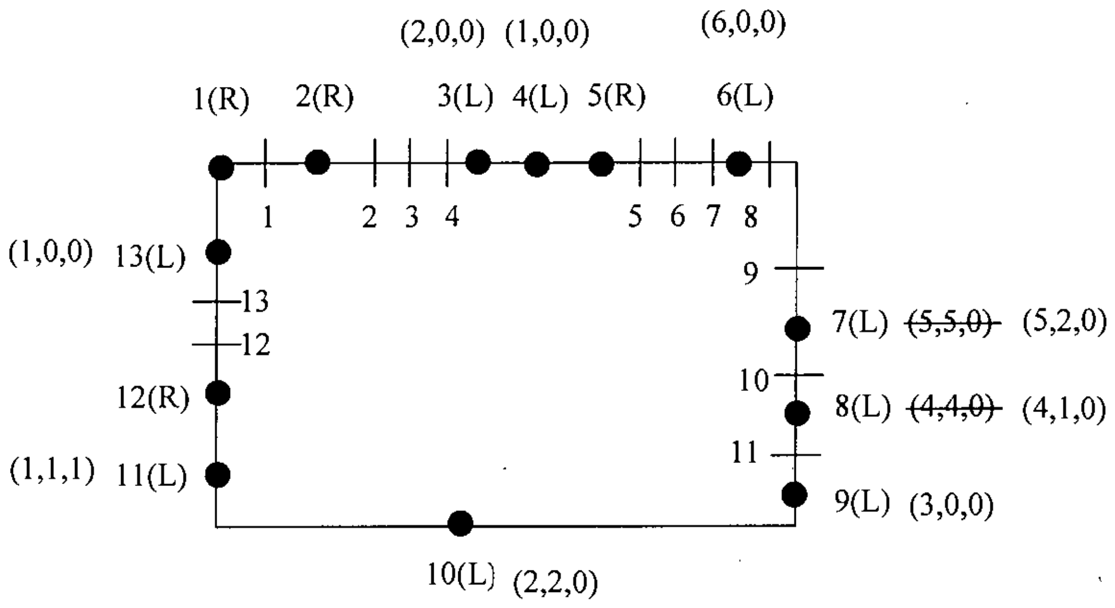


Figure 4.10: Illustration of the *MinimizeLeftRoute*(*B*) algorithm.

200900

The algorithm to minimize the (*S*, *N*, *O*) values for the left vertices is as follows:

```
Algorithm MinimizeLeftRoute( B)
begin
  $v_1$ = First Left Vertex as is determined in InitialLeftRoute algorithm
  Move anti-clockwise around the block $B$
     if a Right vertex is reached then
       $v_1$=Next available left vertex
     endif

     $v_2$ = Next left vertex of $v_1$
     if $v_1$ and $v_2$ are on the same line then
        if Number[$v_2$][$N$] > 0 then
           Number[$v_2$][$N$] = Number[$v_1$][$N$] +1
        endif
        if Number[$v_2$][$O$] > 0 then
           Number[$v_2$][$O$] = Number[$v_1$][$O$] +1
        endif
     endif
```

```
        if $v_1$ and $v_2$ are on two neighboring lines then
            if Number[$v_2$][$S$] >= Number[$v_1$][$N$] +1 then
                Number[$v_2$][$S$] = Number[$v_1$][$N$] +1
            endif
            if Number[$v_2$][$N$] > Number[$v_1$][$O$] +1 then
                Number[$v_2$][$N$] = Number[$v_1$][$O$] +1
            endif
            if Number[$v_2$][$O$] > 0 then
                Number[$v_2$][$O$] = 1
            endif
        endif
        if $v_1$ and $v_2$ are on two neighboring lines then
            if Number[$v_2$][$S$] >= Number[$v_1$][$O$] +1 then
                Number[$v_2$][$S$] = Number[$v_1$][$O$] +1
            endif
            if Number[$v_2$][$N$] > 0 then
                Number[$v_2$][$N$] = 1
            endif
            if Number[$v_2$][$O$] > 0 then
                Number[$v_2$][$O$] = 1
            endif
        endif
        $v_1 = v_2$
    endMove
end
```

4.2.3 Bending the Edges

After classifying and finding routing parameters for each vertex, we bend the edges so that edges are connected to the box at the actual pin position. We bend the edges using (S, N, O) values as calculated in previous subsection. (S, N, O) values define how many grids the net is needed to go away from different box sidelines and leftness and rightness says in which direction (clockwise or anti-clockwise).

Here we have shown only the bending operation of vertices those are on line L_1. Similarly, we can bend the edges connected to other lines of the box. The process of bending the edges is given below (For line L_1 only):

Let, $V(x_v, y_v)$ is a vertex and its corresponding pin is $P(x_p, y_p)$. For simplicity, we have denoted $V(x_v, y_v)$ as V and $P(x_p, y_p)$ as P in the rest of this section. There arise following distinct cases:

Case 1 V and P both are on line L_1

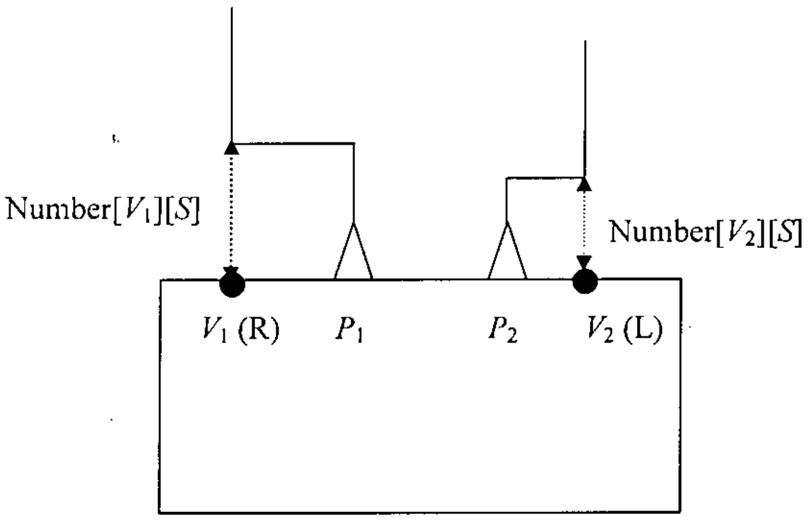


Figure 4.11: V and P both are on line L_l.

If V is a left vertex then edge goes Number[V][S] grids up from line L_1 and move left until the top of pin is reached and then move down to pin. Otherwise V is a right vertex and move Number[V][S] grids up from line L_1 and move to right until the top of pin and then move down to pin. Figure 4.11 illustrates such two bending.

Case 2 V is on L_1 and P is on L_2

Move the edge Number[V][S] grid up from line L_1 and move to the right until Number[V][N] grid away from line L_2, then go down up to the top of pin and connect to the it. Figure 4.12 is an example of V on L_1 and P on L_2.

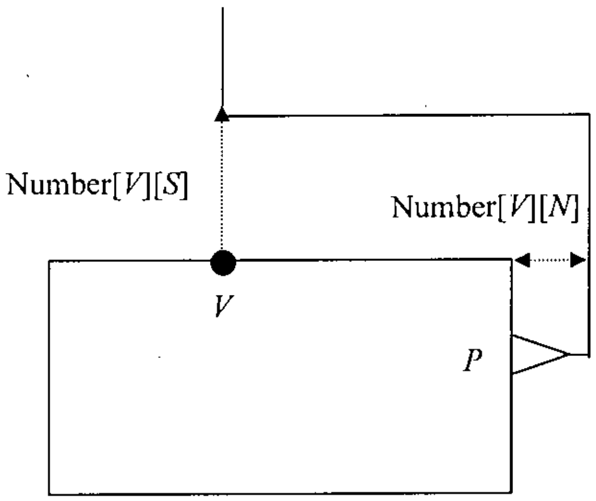


Figure 4.12: V is on L_1 and P is on L_2.

Case 3 V is on L_1 and P is on L_4

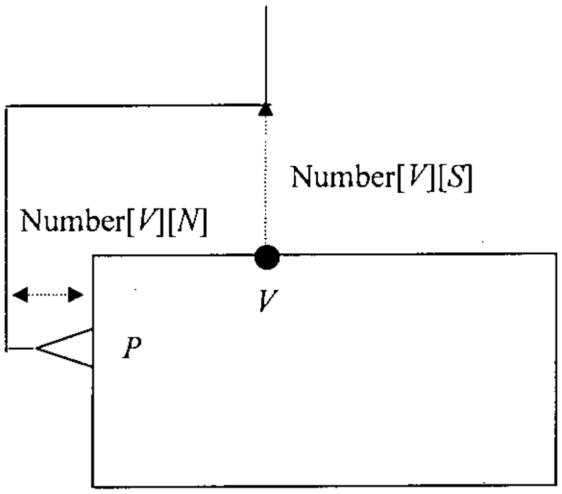


Figure 4.13: V is on L_1 and P is on L_4.

Bend the edge Number[V][S] grid up from line L_1 and move to the left until Number[V][N] grids away from line L_4, then go down along the line up to the top of pin and then connect to it. Figure 4.13 is an example of V on L_1 and P on L_4.

Case 4 V is on L_1 and P is on L_3

Depending on the relative position of V and P on line L_1 and L_3 respectively, vertex V may be a left vertex or a right vertex.

Case 4a) Vertex V is a Right Vertex

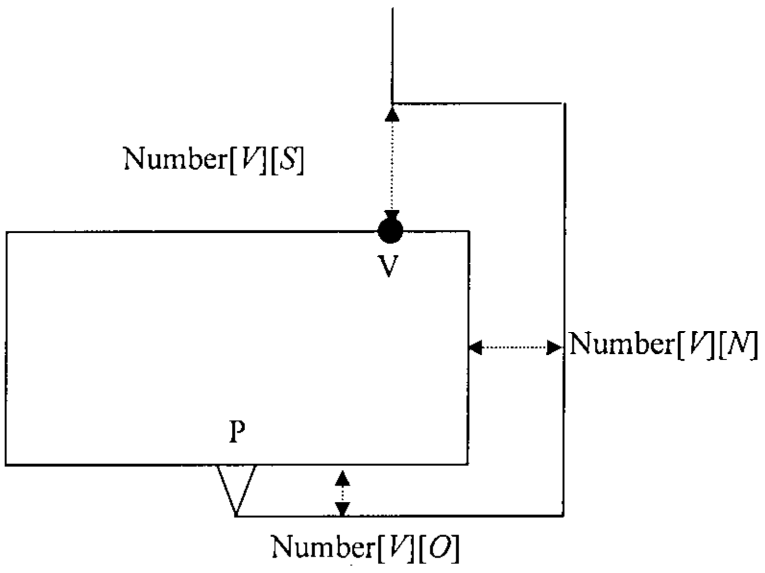


Figure 4.14: V is on L_1, P is on L_3 and V is a right vertex.

Move the edge Number[V][S] grids up from line L_1 and move to right until Numbering[V][N] grids away from line L_2 then move to the down along L_2 and reach Numbering[V][O] grids below from the line L_3 and move to the left up to the pin and connect to it . Figure 4.14 is such an example.

Case 4b) Vertex V is a Left Vertex

Bend the edge Number[V][S] grid up from line L_1 and move to left until Numbering[V][N] grid away from line L_4 then move to the down along L_4 and reach Numbering[V][O] grid below from the line L_3 and move to the right up to the pin and connect to it . Figure 4.15 is such an example.

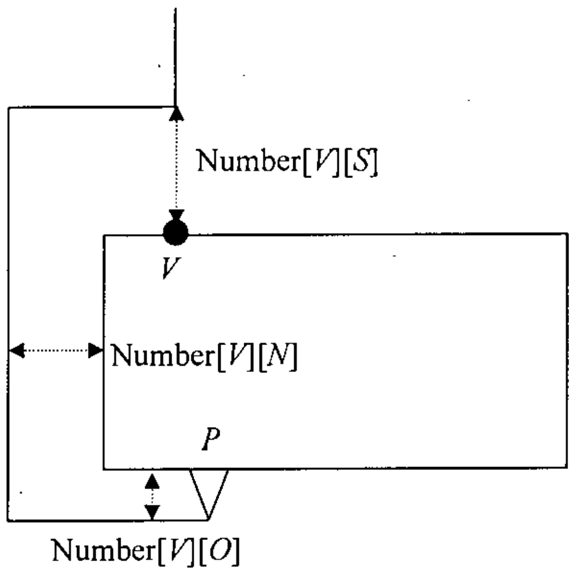


Figure 4.15: V is on L_1 and P is on L_3 and V is left vertex.

4.3 Time Complexity of Detailed Routing Algorithm

In this section we find out the time complexity of the detailed routing algorithm discussed in the Section 4.2. The input to the detailed routing algorithm is the floorplan and global routing of a circuit specification as obtained in Chapter 3 and output is the physical layout for it. The entire process of getting physical layout from a given floorplan is accomplished with the following four phase's one after another:

i) To classify the vertices into left and right vertices.
ii) To assign initial (S, N, O) values to the left vertices using *InitialLeftRoute*(B) and to right vertices using *InitialRightRoute*(B) algorithms.
iii) To assign initial (S, N, O) values to the left vertices using *InitialLeftRoute*(B) and to right vertices using *InitialRightRoute*(B) algorithms.
iv) To minimize (S, N, O) values of the left vertices using *InitialMinimizeRoute*(B) and of right vertices using *MinimizeRightRoute*(B) algorithms.
v) To bend the net using vertex classification (left or right) and the minimized (S, N, O) values of two terminal vertices.

Thus the cost of the detailed routing algorithm is the summation of the costs of the above mentioned phases. Here we define the problem size N as the number of nets (interconnections between the blocks in the circuit specification). Let, M is the number of vertices in the floorplan drawing obtained in Chapter 3; then M is less (multi-terminal nets) or equal to the twice of N.

During classifying the vertices into left and right, we need to compare each vertex coordinate $V(x_v, y_v)$ with its actual pin coordinate on the block $P(x_p, y_p)$ only once. Thus to classify M vertices into left or right we need M such comparisons. So the cost of classifying the vertices into left and right is $O(M)$ and as M=2*N we can say it is $O(N)$.

The next step of our routing algorithm is to get initial routing parameters that is the initial values of (S, N, O) tuples for each left and right vertices using the *InitialLeftRoute*(B) and *InitialRightRoute*(B) algorithms.

Let, L is the number of left vertices and R is the number of right vertices after classification then $M=L+R$.

The only iterative statement in the *InitialLeftRoute*(B_i) algorithm is to move anti-clockwise around the block to calculate (S, N, O) values for left vertices of the block. Within this move statement there is only sequential or conditional statements which has cost $O(1)$. Let, M_i is the number vertices on the block B_i among them L_i is the number of left vertices. Then the cost of getting initial routing parameters for L_i left vertices of block B_i is $O(L_i)$. So the total cost of getting initial routing parameters for all L left vertices of the entire floorplan is $\Sigma(O(L_i)) = O(\Sigma(L_i)) = O(L)$.

Similarly, the only iterative statement in the *InitialrightRoute*(B_i) algorithm is to move clockwise around the block to calculate (S, N, O) values for right vertices of the block. Let, M_i be the number vertices on the block B_i among them R_i is the number of right vertices. Then the cost of getting initial routing parameters for R_i right vertices of block B_i is $O(R_i)$. So the total cost of getting initial routing parameters for all R right vertices of the entire floorplan is $\Sigma(O(R_i)) = O(\Sigma(R_i)) = O(R)$.

So the total cost of getting initial routing parameters of all left and right vertices is $O(L) + O(R) = O(L+R) = O(M)$; which cal also be written as $O(N)$.

MinimizeLeftRoute(B_i) and *MinimizeRightRoute*(B_i) algorithms individually minimizes the routing parameters of the left and right vertices of block B_i respectively. The former one moves anti-clockwise around the block B_i and considers each left vertex once. Thus the cost is $O(L_i)$. The later one moves clockwise around the block B_i and considers each right vertex once. Thus the cost is $O(R_i)$. So the cost of minimization for block B_i is $O(L_i) + O(R_i) = O(L_i + R_i) = O(M_i)$. So the total cost of this minimization for all blocks is $\Sigma(O(M_i)) = O(\Sigma(M_i)) = O(M)$.

After the minimization of the routing parameters routing algorithm starts to bend each net to position from the vertex coordinates to the actual pin coordinates using the left/right classification and the minimized (S, N, O) values of the two terminals of the net as described in Section 4.2.3. The cost of bending each terminal of a net is constant thus has the cost $O(1)$. So to bend about $2*N$ terminals of N nets we have the cost $O(N)$.

As each of the four phases of detailed routing algorithm has the cost $O(N)$ and they operates sequentially one after another so the total cost of this algorithm is $O(N)$. Thus we have the following theorem.

Theorem 4.1 Algorithm *DetailedRouting(Floorplan F)* finds a detailed routing for a given floorplan F in linear time.

Chapter 5

Conclusion

Integrated circuits (ICs) consist of a number of electronic components built by layering several materials in a well defined fashion on a silicon base. The designer of an IC transforms a circuit description into a geometric description (layout). The process of converting the specification of an electrical circuit into a layout is the physical design process. Due to the tight tolerance requirements and the extremely small size of the individual components, physical design is an extremely tedious and error prone process. As a result, all phases of physical design process need automation.

There are four phases in physical design. Those are partitioning, floorplanning, routing (global routing and detailed routing) and compaction. This thesis presents linear time algorithms for the automation of floorplanning and routing phases.

Traditionally, floorplanning and global routing are handled in two successive phases and may need several iterations. Our algorithm handles these two in a single phase and avoids any backtracking from routing to floorplanning phase. The time complexity of our algorithm is linear.

We also develop a linear time algorithm for detailed routing. One important thing of this algorithm is that it always finds a detailed routing for the given result as obtained from our previous integrated algorithm for floorplanning and global routing. Thus we need no backtracking here also.

Scope of Future Work

In this thesis, we use orthogonal drawing of plane graph to get floorplanning and global routing. We first represent a circuit description into a graph. We assume that the graph is planar and the graph has floorplanning and global routing for the circuit in a 2D plane. That is our algorithm solves the single layer floorplanning and routing problems. In the rest of this chapter we describe some open problems related to floorplanning and routing.

If we present a complex circuit using multigraph as described in Chapter 3, we may obtain a nonplanar graph. Then, floorplanning and routing need multiple layers. An efficient algorithm for multi-layer floorplanning and routing in 3D planes may be a good work.

A non-planar graph may be partitioned into a set of planar graphs such that their union will give the original non-planar graph and the planar graphs are edge disjoint [13]. Then the floorplanning and routing of each planar graph can be obtained by using our single layer floorplanning and routing algorithm. But the difficulty is that the orthogonal drawing algorithm, used as the part of our floorplanning and global routing algorithm, do not assume the position of the vertices of the planar graph as fixed. Thus in different layer same vertex may be placed in different position and we need to map them on a single 3D pillar. One can try them in the following two ways:

1. One may try to develop an algorithm that will give intersection free connections between the two places of the same vertex on two successive planes.
2. One may try to develop an efficient orthogonal drawing of a plane graph when the positions of the vertices are fixed.

During detailed routing we introduced a number of bends in order to reach from the vertex position to the actual pin position. Again, the nets move around the side lines of a single block in a way that ensure local area minimization for that block; but it does not ensure the overall area optimization for the total layout. Bend minimization during detail routing can be an interesting research area. Area compaction may be another challenging problem for research.

In our algorithm, we gave same importance to all the blocks and to all the nets. But in practice, some blocks and connections of a circuit are more important than the others. Introducing the unequal priority levels to different blocks and nets and then solving the floorplanning and routing problem to optimize some objective function may be a challenging problem. For example, among all the blocks only a few blocks communicate most of the times during the operations of a circuit. Then giving higher priority to those blocks and the nets among those blocks and bringing them closer during floorplanning and routing will minimize the overall delay of the circuit. Similarly, the width of all nets may not be the same. Taking count of the unequal width of different nets is another challenge in floorplanning and routing problems.

Finally, we can conclude that there is a huge scope of research on the field of automated VLSI layout design, especially on the physical design cycle.

Bibliography

[1] S.B. Akers, *On the use of linear assignment algorithm in module placement*, Proceeding of 18th ACM/IEEE Design Automation Conference, pp.137-144, 1981.

[2] R. Camposano and R. K. Brayton, *Partitioning before logic synthesis*, Proc. of IEEE International Conference on CAD, pp.324-326, 1987.

[3] J. Cong, *Pin assignment with global routing*, Proc. of IEEE International Conference on CAD, pp.302-305, 1989.

[4] J. Cong and B. Preas, *A new algorithm for standard cell global routing*, Proc. of IEEE International Conference on CAD, pp.176-179, 1988.

[5] G. Di Battista, P. Eades, R. Tamassia and I. G. Tollis, *Graph Drawing: Algorithms for the Visualization of Graphs*, Prentice-Hall Inc., Upper Saddle River, New Jersey, 1999.

[6] H. de Fraysseix, J. Pach and R. Pollack, *How to draw a planar graph on a grid*, Combinatorica 10, pp. 41-51, 1990.

[7] A. Grag and R. Tamassia, *On the computational complexity of upward and rectilinear planarity testing*, Proc. Of Graph Drawing'94, Lect. Notes in Computer Science, 894, Springer, pp.286-297, 1995.

[8] A. Grag and R. Tamassia, *A New minimum cost flow algorithm with applications of graph drawing*, Proc. Of Graph Drawing'96, Lect. Notes in Computer Science, Springer, pp.201-226, 1997.

[9] Raia T. Hadsell and Patrick H. Madden, *Improved global routing through congestion estimation*, DAC'03, Anaheim, California, USA, 2003.

[10] D. W. Jepsen and C.D. Gelatt Jr., *Macro planement by monte carlo annealing*, Proceeding of IEEE Design Automation Conference on Computer Design, pp. 175-193, 1983.

[11] Ryan Kastner, Elaheh Bozorgzadeh and Majid Sarrafzadeh, *An exact algorithm for coupling-free routing*, ISPD'01, Sibina, California, USA, 2001.

[12] T. Lengauer, Combinatorial algorithms for integrated circuit layout, Wiley, Chichester, 1990.

[13] Annegret Liebers, Planarizing graphs – A survey and annotated bibliography, Journal of Graph Algorithms and Applications Vol 5, No 1, pp.1-74, 2001.

[14] A. Leblond, *Caf: A Computer-Floorplanning Tool*, Proceeding of 20th Design Automation Conference, pp. 747-753, 1983.

[15] S. Mohan and P. Mazumdar, *Wolverines: Standard cell placement on a network of workstations*, IEEE Transactions on CAD of Integrated Circuits and Systems, 12, pp.1312-1326, 1993.

[16] K. McCullen, J. Thorvaldson, D. Demaris and P. Lampin, *A system for floorplanning with hierarchical placement and wiring*, Proc. of European Design Automation Conference, pp.262-265, 1990.

[17] T. Nishizeki and M. S. Rahman, *Planar graph drawing*, World Scientific, 2004.

[18] A. M. Patel, *Partitioning of VLSI Placement Problems*, Proceeding of 18th ACM/AEEE Design Automation Conference, pp.137-144, 1981.

[19] D. P. La Potin and S. W. Director, *A global floorplanning approach for VLSI design*, IEEE Transaction on Computer Aided Design, pp.477-489, 1986.

[20] P. Pan and C. L. Lin, *Area minimization for floorplans*, IEEE Transactions on CAD of Circuits and System, pp.14:123-132, 1995.

[21] M. Pedram, M. Marek-Sadowska and E.S.kuh, *Floorplanning with pin assignment*, Proceedings of International Conference on Computer-Aided Design, pp.98-101, 1990.

[22] M. S. Rahman, S. Nakano and T. Nishizeki, *Rectangular grid drawings of plane graphs*, Comp. Geom. Theo. Appl. 10 (3), pp. 203-220, 1998.

[23] M. S. Rahman, S. Nakano and T. Nishizeki, *A linear algorithm for bend-optimal orthogonal drawings of triconnected cubic plane graphs*, Journal of Graph Algorithms and Applications Vol 3, No 4, pp.31-62, 1999.

[24] M. S. Rahman, S. Nakano and T. Nishizeki, *Rectangular drawings of plane graphs without designated corners*, Proc. of COCOON'00, LNCS 1858, Springer, pp.85-94, 2000.

[25] M. S. Rahman, S. Nakano and T. Nishizeki, *Box-rectangular drawings of plane graphs*, J. of Algorithms 37, pp.363-398, 2000.

[26] M. S. Rahman, M. Naznin and T. Nishizeki, *Orthogonal drawings of plane graphs without bends*, Proc. Of Graph Drawing'01, Lect. Notes in Computer Science, 2265, pp.392-406, 2002.

[27] M. S. Rahman and T. Nishizeki, *Bend-minimum orthogonal drawing of plane 3-graph*, Proc. of International Workshop on Graph Theoretic Concepts in Computer Science (WG'04), Lect. Notes in Computer Science, Springer, pp. 367-378, 2004.

[28] J. Storer, *On minimum node-cost planar embeddings*, Networks 14, pp.181-212, 1984.

[29] M. Sarrafzadeh and R.D Lou, *Maximum k-coverings of weighted transitive graphs with applications*, Proceedings of IEEE International Symposium on Circuits and Systems, pp. 332-335, 1990.

[30] R. Tamassia, On embedding a graph in the grid with the minimum number of bends, SIAM J.Comput., pp. 421-444, 1987.

[31] X. Yao, M. Yamada and C. L. Liu, *A new approach to the pin assignment problem*. Proc. of 25th ACM/IEEE Design Automation Conference, pp.566-572, 1988.

Index