Chapter 4

Data Structures and Basic Algorithms

VLSI chip design process can be viewed as transformation of data from HDL code in logic design, to schematics in circuit design, to layout data in physical design. In fact, VLSI design is a significant database management problem. The layout information is captured in a symbolic database or a polygon database. In order to fabricate a VLSI chip, it needs to be represented as a collection of several layers of planar geometric elements or polygons. These elements are usually limited to Manhattan features (vertical and horizontal edges) and are not allowed to overlap within the same layer. Each element must be specified with great precision. This precision is necessary since this information has to be communicated to output devices such as plotters, video displays, and pattern-generating machines. Most importantly, the layout information must be specific enough so that it can be sent to the fab for fabrication. Symbolic database captures net and transistor attributes. It allows a designer to rapidly navigate throughout the database and make quick edits while working at a higher level. The symbolic database is converted into a polygon database prior to tapeout. In the polygon database, the higher level relationship between the objects is somewhat lost. This process is analogous to conversion of a higher level programming language (say FORTRAN) code to a lower level programming language (say Assembly) code. While it is easier to work at symbolic level, it cannot be used by the fab directly. In some cases, at late stages of the chip design process, some edits have to be made in the polygon database. The major motivation for use of the symbolic database is technology independence. Since physical dimensions in the symbolic database are only relative, the design can be implemented using any process. However, in practice, complete technology independence has never be reached.

The layouts have historically been drawn by human layout designers to conform to the design rules and to perform the specified functions. A physical design specialist typically converted a small circuit into layout consisting of a set of polygons. These manipulations were time consuming and error prone,

even for small layouts. Rapid advances in fabrication technology in recent years have dramatically increased the size and complexity of VLSI circuits. As a result, a single chip may include several million transistors. These technological advances have made it impossible for layout designers to manipulate the layout databases without sophisticated CAD tools. Several physical design CAD tools have been developed for this purpose, and this field is referred to as Physical Design Automation. Physical design CAD tools require highly specialized algorithms and data structures to effectively manage and manipulate layout information. These tools fall in three categories. The first type of tools help a human designer to manipulate a layout. For example, a layout editor allows designers to add transistors or nets to a layout. The second type of tools are designed to perform some task on the layout automatically. Example of such tools include channel routers and placement tools. It is also possible to invoke a tool of second type from the layout editor. The third type of tools are used for checking and verification. Example of such tools include; DRC (design rule checker) and LVS verifier (layout versus schematics verifier). The bulk of the research of physical design automation has focused on tools of the last two types. However, due to broad range and significant impact the tools of second type have received the most attention. The major accomplishment in that area has been decomposition of the physical design problem into several smaller (and conceptually easier) problems. Unfortunately, even these problems are still computationally very hard. As a result, the major focus has been on development on design and analysis of heuristic algorithms for partitioning, placement, routing and compaction. Many of these algorithms are based on graph theory and computational geometry. As a result, it is important to have a basic understanding of these two fields. In addition, several special classes of graphs are used in physical design. It is important to understand properties and algorithms about these classes of graphs to develop effective algorithms in physical design.

This chapter consists of three parts. First we discuss the basic algorithms and mathematical methods used in VLSI physical design. These algorithms form the basis for many of the algorithms presented later in this book. In the second part of this chapter, we shall study the data structures used in layout editors and the algorithms used to manipulate these data structures. We also discuss the formats used to represent VLSI layouts. In the third part of this chapter, we will focus on special classes of graphs, which play a fundamental role in development of these algorithms. Since most of the algorithms in VLSI physical design are graph theoretic in nature, we devote a large portion of this chapter to graph algorithms. In the following, we will review basic graph theoretic and computation geometry algorithms which play a significant role in many different VLSI design algorithms. Before we discuss the algorithms, we shall review the basic terminology.

4.1 Basic Terminology

A *graph* is a pair of sets $G = (V, E)$, where V is a set of vertices, and E is a set of pairs of distinct vertices called *edges.* We will use $V(G)$ and $E(G)$ to refer to the vertex and edge set of a graph G if it is not clear from the context. A vertex u is adjacent to a vertex v if (u, v) is an edge, i.e., $(u, v) \in E$. The set of vertices adjacent to v is $Adj(v)$. An edge $e = (u, v)$ is *incident* on the vertices u and v, which are the ends of e. The degree of a vertex u is the number of edges incident with the vertex u.

A complete graph on n vertices is a graph in which each vertex is adjacent to every other vertex. We use K_n to denote such a graph. A graph H is called the *complement* of graph $G = (V, E)$ if $H = (V, F)$, where, $F = E(K_{|V|}) - E$.

A graph $G' = (V', E')$ is a *subgraph* of a graph G if and only if $V' \subseteq V$ and $E' \subseteq E$. If $E' = \{(u, v) \mid (u, v) \in E \text{ and } u, v \subseteq V'\}$ then G' is a *vertex induced subgraph* of G. Unless otherwise stated, by subgraph we mean vertex induced subgraph.

A *walk* P of a graph G is defined as a finite alternating sequence $P = v_0, e_1, \ldots, e_k, v_k$ of vertices and edges, beginning and ending with vertices, such that each edge is incident with the vertices preceding and following it.

A *tour* is a walk in which all edges are distinct. A walk is called an *open walk* if the terminal vertices are distinct. A *path* is a open walk in which no vertex appears more than once.

The *length* of a path is the number of edges in it. A path is a (u, v) path if $v_0 = u$ and $v_k = v$. A *cycle* is a path of length $k, k > 2$ where $v_0 = v_k$. A cycle is called *odd* if it's length k is odd, otherwise it is an *even* cycle. Two vertices u and v in G are *connected* if G has a (u, v) path. A graph is connected if all pairs of vertices are connected. A *connected component* of G is a maximal connected subgraph of G. An edge $e \in E$ is called an *cut edge* in G if its removal from G increases the number of connected components of G by at least one. A *tree* is a connected graph with no cycles. A complete subgraph of a graph is called a *clique.*

A *directed graph* is a pair of sets $(V, \vec{E})$, where V is a set of vertices and $\vec{E}$ is a set of ordered pairs of distinct vertices, called *directed edges.* We use the notation $\vec{G}$ for a directed graph, unless it is clear from the context. A directed edge $\vec{e} = (u, v)$ is incident on u and v and the vertices u and v are called the *head* and *tail* of $\vec{e}$, respectively. $\vec{e}$ is an *in-edge* of v and an *out-edge* of u. The *in-degree* of u denoted by $d^-(u)$ is equal to the number of in-edges of u, similarly the *out-degree* of u denoted by $d^+(u)$ is equal to the number of out-edges of u. An *orientation* for a graph $G = (V, E)$ is an assignment of direction for each edge. An orientation is called *transitive* if, for each pair of edges (u, v) and (v, w), there exists an edge (u, w). If such a transitive orientation exists for a graph G, then G is called a *transitively orientable graph.* Definitions of subgraph, path, walk are easily extended to directed graphs. A *directed acyclic graph* is a directed graph with no directed cycles. A vertex u is an *ancestor* of v (and v is a *descendent* of u) if there is a (u, v) directed path in G. A rooted tree (or *directed tree*) is a directed acyclic graph in which all vertices

have in-degree 1 except the *root*, which has in-degree 0. The root of a rooted tree T is denoted by *root(T)*. The *subtree* of tree T rooted at v is the subtree of T induced by the descendents of v. A *leaf* is a vertex in a directed acyclic graph with no descendents.

A *hypergraph* is a pair (V, E), where V is a set of vertices and E is a family of sets of vertices. A *hyperpath* is a sequence $P = v_0, e_1, \ldots, v_{k-1}, e_k, v_k$ of distinct vertices and distinct edges, such that vertices v_{i-1} and v_i are elements of the edge $e_i, 1 \leq i \leq k$. Two vertices u and v are connected in a hypergraph if the hypergraph has a (u, v) hyperpath. A hypergraph is connected if every pair of vertices are connected.

A *bipartite graph* is a graph G whose vertex set can be partitioned into two subsets X and Y, so that each edge has one end in X and one end in Y; such a partition (X, Y) is called *bipartition* of the graph. A *complete bipartite graph* is a bipartite graph with bipartition (X, Y) in which each vertex of X is adjacent to each vertex of Y; if $|X| = m$ and $|Y| = n$, such a graph is denoted by $K_{m,n}$. An important characterization of bipartite graphs is in terms of odd cycles. A graph is bipartite if and only if it does not contain an odd cycle.

A graph is called *planar* if it can be drawn in the plane without any two edges crossing. Notice that there are many different ways of 'drawing' a planar graph. A drawing may be obtained by mapping a vertex to a point in the plane and mapping edges to paths in the plane. Each such drawing is called an *embedding* of G. An embedding divides the plane into finite number of regions. The edges which bound a region define a *face*. The unbounded region is called the *external* or *outside face*. A face is called an odd face if it has odd number of edges. Similarly a face with even number of edges is called an even face. The *dual* of a planar embedding T is a graph $G_T = (V_T, E_T)$, such the $V_T = \{v|$ v is a face in $T\}$ and two vertices share an edge if their corresponding faces share an edge in T.

4.2 Complexity Issues and NP-hardness

Several general algorithms and mathematical techniques are frequently used to develop algorithms for physical design. While the list of mathematical and algorithmic techniques is quite extensive, we will only mention the basic techniques. One should be very familiar with the following techniques to appreciate various algorithms in physical design.

1. Greedy Algorithms
2. Divide and Conquer Algorithms
3. Dynamic Programming Algorithms
4. Network Flow Algorithms
5. Linear/Integer Programming Techniques

Since these techniques and algorithms may be found in a good computer science or graph algorithms text, we omit the discussion of these techniques and refer the reader to an excellent text on this subject by Cormen, Leiserson and Rivest [CLR90].

The algorithmic techniques mentioned above have been applied to various problems in physical design with varying degrees of success. Due to the very large number of components that we must deal with in VLSI physical design automation, all algorithms must have low time and space complexities. For algorithms which must operate on the entire layout ($n \geq 10^6$), even quadratic algorithms may be intolerable. Another issue of great concern is the constants in the time complexity of algorithms. In physical design, the key idea is to develop practical algorithms, not just polynomial time complexity algorithms. As a result, many linear and quadratic algorithms are infeasible in physical design due to large constants in their complexity.

The major cause of concern is absence of polynomial time algorithms for majority of the problems encountered in physical design automation. In fact, there is evidence that suggests that no polynomial time algorithm may exist for many of these problems. The class of solvable problems can be partitioned into two general classes, P and NP. The class P consists of all problems that can be solved by a deterministic turing machine in polynomial time. A conventional computer may be viewed as such a machine. Minimum cost spanning tree, single source shortest path, and graph matching problems belong to class P. The other class called NP, consists of problems that can be solved in polynomial time by a nondeterministic turing machine. This type of turing machine may be viewed as a parallel computer with as many processors as we may need. Essentially, whenever a decision has several different outcomes, several new processors are started up, each pursuing the solution for one of the outcomes. Obviously, such a model is not very realistic. If every problem in class NP can be reduced to a problem **P**, then problem **P** is in class NP-complete. Several thousand problems in computer science, graph theory, combinatorics, operations research, and computational geometry have been proven to be NP-complete. We will not discuss the concept of NP-completeness in detail, instead, we refer the reader to the excellent text by Garey and Johnson on this subject [GJ79]. A problem may be stated in two different versions. For example, we may ask does there exist a subgraph H of a graph G, which has a specific property and has size k or bigger? Or we may simply ask for the largest subgraph of G with a specific property. The former type is the decision version while the latter type is called the optimization version of the problem. The optimization version of a problem **P**, is called NP-hard if the decision version of the problem **P** is NP-complete.

4.2.1 Algorithms for NP-hard Problems

Most optimization problems in physical design are NP-hard. If a problem is known to be NP-complete or NP-hard, then it is unlikely that a polynomial time algorithm exists for that problem. However, due to practical nature of the

physical design automation field, there is an urgent need to solve the problem even if it cannot be solved optimally. In such cases, algorithm designers are left with the following four choices.

4.2.1.1 Exponential Algorithms

If the size of the input is small, then algorithms with exponential time complexity may be feasible. In many cases, the solution of a certain problem be critical to the performance of the chip and therefore it is practical to spend extra resources to solve that problem optimally. One such exponential method is integer programming, which has been very successfully used to solve many physical design problems. Algorithms for solving integer programs do not have polynomial time complexity, however they work very efficiently on moderate size problems, while worst case is still exponential. For large problems, algorithms with exponential time complexity may be used to solve small sub-cases, which are then combined using other algorithmic techniques to obtain the global solution.

4.2.1.2 Special Case Algorithms

It may be possible to simplify a general problem by applying some restrictions to the problem. In many cases, the restricted problem may be solvable in polynomial time. For example, the graph coloring problem is NP-complete for general graphs, however it is solvable in polynomial time for many classes of graphs which are pertinent to physical design.

Layout problems are easier for simplified VLSI design styles such as standard cell, which allow usage of special case algorithms. Conceptually, it is much easier to place cells of equal heights in rows, rather than placing arbitrary sized rectangles in a plane. The clock routing problem, which is rather hard for full-custom designs, can be solved in $O(n)$ time for symmetric structures such as gate arrays. Another example may be the Steiner tree problem (see Section 4.3.1.6). Although the general Steiner tree problem is NP-hard, a special case of the Steiner tree problem, called the single trunk steiner tree problem (see exercise 4.7), can be solved in $O(n)$ time.

4.2.1.3 Approximation Algorithms

When exponential algorithms are computationally infeasible due to the size of the input and special case algorithms are ruled out due to absence of any restriction that may be used, designers face a real challenge. If optimality is not necessary and near-optimality is sufficient, then designers attempt to develop an approximation algorithm. Often in physical design algorithms, near-optimality is good enough. Approximation algorithms produce results with a guarantee. That is, while they may not produce an optimal result, they guarantee that the result would never be worse than a lower bound determined by the *performance ratio* of the algorithm. The performance ratio γ of an algorithm is defined as $\frac{\Phi}{\Phi^*}$, where Φ is the solution produced by the algorithm and Φ^* is the optimal

```
Algorithm AVC
begin
    S = φ;
    E' = E;
    R = φ;
    while E' ≠ φ do
        (* Select an arbitrary edge e = (u, v) from E' *)
        e = Select(E');
        S = S ∪ {u, v};
        R = R ∪ e;
        E' = E' − {(p, q) | (p = u or v) or (q = u or v)};
end.
```

Figure 4.1: Algorithm AVC.

solution for the problem. Recently, many algorithms have been developed with γ very close to 1. We will use *vertex cover* as an example to explain the concept of an approximation algorithm.

Vertex cover is basically a problem of covering all edges by using as few vertices as possible. In other words, given an undirected graph $G = (V,E)$, select a subset $V' \subseteq V$, such that for each edge $(u,v) \in E$, either u or v or both are in V' and V' has minimum size among all such sets. The vertex cover problem is known to be NP-complete for general graphs. However, the simple algorithm given in Figure 4.1 achieves near optimal results. The basic idea is to select an arbitrary edge (u,v) and delete it and all edges incident on u and v. Add u and v to the vertex cover set S. Repeat this process on the new graph until all edges are deleted. The selected edges are kept in a set R.

Since no edge is checked more than once, it is easy to see that the algorithm AVC runs in $O(|E|)$ time.

Theorem 1 *Algorithm AVC produces a solution with a performance ratio of* 0.5.

Proof: Note that no two edges in R have a vertex in common, and $|S| = 2 \times |R|$. However, since R is set of vertex disjoint edges, at least $|R|$ vertices are needed to cover all the edges. Thus $\gamma \geq \frac{|R|}{2|R|} = 0.5$.

In Section 4.5.6.2, we will present an approximation algorithm for finding maximum k-partite $(k \geq 2)$ subgraph in circle graphs. That algorithm is used in topological routing, over-the-cell routing,via minimization and several other physical design problems.

4.2.1.4 Heuristic Algorithms

Faced with NP-complete problems, heuristic algorithms are frequently the answer. A heuristic algorithm does produce a solution but does not guaran-

tee the optimality of the solution. Such algorithms must be tested on various benchmark examples to verify their effectiveness. The bulk of research in physical design has concentrated on heuristic algorithms. An effective heuristic algorithm must have low time and space complexity and must produce an optimal or near optimal solution in most realistic problem instances. Such algorithms must also have good average case complexities. Usually good heuristic algorithms are based on optimal algorithms for special cases and are capable of producing optimal solutions in a significant number of cases. A good example of such algorithms are the channel routing algorithms (discussed in chapter 7), which can solve most channel routing problems using one or two tracks more than the optimal solution. Although the channel routing problem in general is NP-complete, from a practical perspective, we can consider the channel routing problem as *solved.*

In many cases, an $O(n)$ time complexity heuristic algorithm has been developed, even if an optimal $O(n^3)$ or $O(n^2)$ time complexity algorithm is known for the problem. One must keep in mind that optimal solutions may be hard to obtain and may be practically insignificant if a solution close to optimal can be produced in a reasonable time. Thus the major focus in physical design has been on the development of practical heuristic algorithms which can produce close to optimal solutions on real world examples.

4.3 Basic Algorithms

Basic algorithms which are frequently used in physical design as subalgorithms can be categorized as: graph algorithms and computational geometry based algorithms. In the following, we review some of the basic algorithms in both of these categories.

4.3.1 Graph Algorithms

Many real-life problems, including VLSI physical design problems, can be modeled using graphs. One significant advantage of using graphs to formulate problems is that the graph problems are well-studied and well-understood. Problems related to graphs include graph search, shortest path, and minimum spanning tree, among others.

4.3.1.1 Graph Search Algorithms

Since many problems in physical design are modeled using graphs, it is important to understand efficient methods for searching graphs. In the following, we briefly discuss the three main search techniques.

1. **Depth-First Search:** In this graph search strategy, graph is searched 'as deeply as possible'. In Depth-First Search (DFS), an edge is selected for further exploration from the most recently visited vertex v. When all the edges of v have been explored, the algorithm back tracks to the

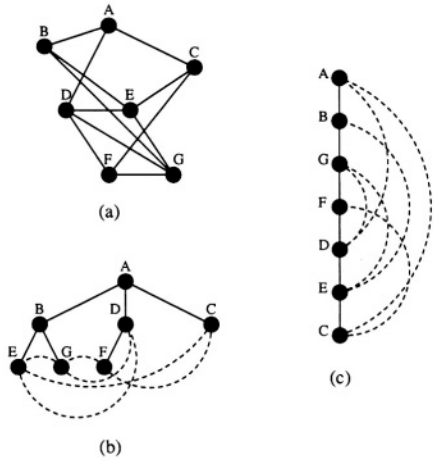


Figure 4.2: Examples of graph search algorithms.

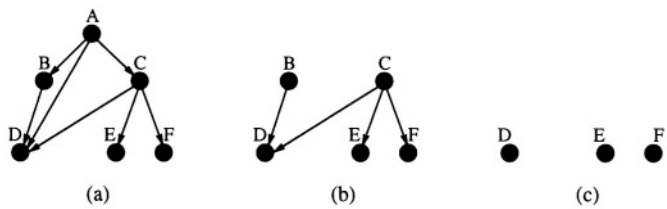


Figure 4.3: Example of the topological search.

previous vertex, which may have an unexplored edge. Figure 4.4 is an outline of a depth-first search algorithm. The algorithm uses an array MARKED (n) which is initialized to zero before calling the algorithm to keep track of all the visited vertices.

It is easy to see that the time complexity of depth-first search is $O(|V| + |E|)$. Figure 4.2(c) shows an example of the depth first search for the graph shown in Figure 4.2(a).

2. **Breadth-First Search:** The basic idea of Breadth-First Search (BFS) is to explore all vertices adjacent to a vertex before exploring any other vertex. Starting with a source vertex v, the BFS first explores all edges of v, puts the reachable vertices in a queue, and marks the vertex v as visited. If a vertex is already marked visited then it is not enqueued. This process is repeated for each vertex in the queue. This process of visiting edges produces a BFS tree. The BFS algorithm can be used to search both directed and undirected graphs. Note that the main difference be-

```
Algorithm DEPTH-FIRST-SEARCH(v)
begin
    MARKED(v) = 1;
    for each vertex u, such that (u, v) ∈ E do
        if MARKED(u) = 0 then
            DEPTH-FIRST-SEARCH(u);
end.
```

Figure 4.4: Algorithm DEPTH-FIRST-SEARCH.

tween the DFS and the BFS is that the DFS uses a stack (recursion is implemented using stacks), while the BFS uses a queue as its data structure. The time complexity of breadth first search is also $O(|V| + |E|)$. Figure 4.2(b) shows an example of the BFS of the graph shown in Figure 4.2(a).

3. **Topological Search:** In a directed acyclic graph, it is very natural to visit the parents, before visiting the children. Thus, if we list the vertices in the topological order, if G contains a directed edge (u, v), then u appears before v in the topological order. Topological search can be done using depth first search and hence it has a time complexity of $O(|V| + |E|)$. Figure 4.3 shows an example of the topological search. Figure 4.3(a) shows an entire graph. First vertex A will be visited since it has no parent. After visiting A, it is deleted (see Figure 4.3(b)) and we get vertices B and C as two new vertices to visit. Thus, one possible topological order would be *A, B, C, D, E, F.*

4.3.1.2 Spanning Tree Algorithms

Many graph problems are subset selection problems, that is, given a graph $G = (V,E)$, select a subset $V' \subseteq V$, such that V' has property $\mathcal{P}$. Some problems are defined in terms of selection of edges rather than vertices. One frequently solved graph problem is that of finding a set of edges which spans all the vertices. The Minimum Spanning Tree (MST) is an edge selection problem. More precisely, given an edge-weighted graph $G = (V, E)$, select a subset of edges $E' \subseteq E$ such that E' induces a tree and the total cost of edges $\sum_{e_i \in E'} wt(e_i)$, is minimum over all such trees, where $wt(e_i)$ is the cost or weight of the edge e_i.

There are basically three algorithms for finding a MST:

1. Boruvka's Algorithm
2. Kruskal's Algorithm
3. Prim's Algorithm.

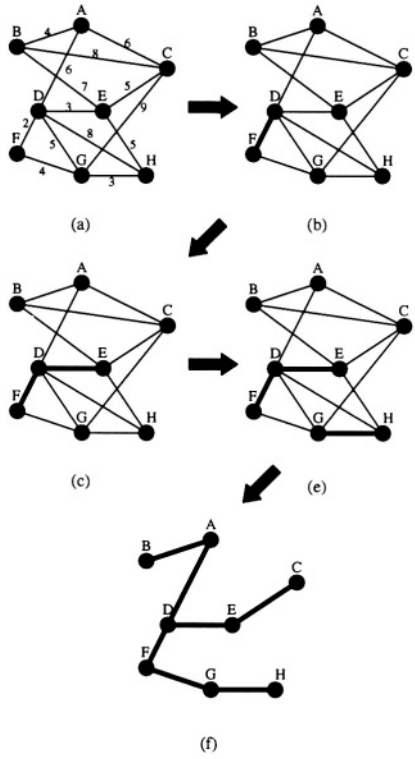


Figure 4.5: Kruskal's algorithm used to find a MST.

We will briefly explain Kruskal's algorithm [Kru56], whereas the details of other algorithms can be found in [Tar83]. Kruskal's algorithm starts by sorting the edges by nondecreasing weight. Each vertex is assigned to a set. Thus at the start, for a graph with n vertices, we have n sets. Each set represents a partial spanning tree, and all the sets together form a spanning forest. For each edge (u, v) from the sorted list, if u and v belong to the same set, the edge is discarded. On the other hand, if u and v belong to disjoint sets, a new set is created by union of these two sets. This edge is added to the spanning tree. In this way, algorithm constructs partial spanning trees and connects them whenever an edge is added to the spanning tree. The running time of Kruskal's algorithm in $O(|E| \log |E|)$. Figure 4.5 shows an example of Kruskal's algorithm. First edge (D, F) is selected since it has the lowest weight (See Figure 4.5(b)). In the next step, there are two choices since there are two edges with weight equal to 2. Since ties are broken arbitrarily, edge (D, E) is selected. The final tree is shown in Figure 4.5(f).

4.3.1.3 Shortest Path Algorithms

Many routing problems in VLSI physical design are in essence shortest path problems in special graphs. Shortest path problems, therefore, play a significant role in global and detailed routing algorithms.

1. **Single Pair Shortest Path:** This problem may be viewed as a vertex or edge selection problem. Precisely stated, given an edge-weighted graph $G = (V, E)$ and two vertices $u, v \in V$, select a set of vertices $P \subseteq V$ including u, v such that P induces a path of minimum cost in G. Let $wt(p, q)$ be the weight of edge (p, q), we assume that $wt(p, q) \geq 0$ for each $(p, q) \in E$.

   Dijkstra [Dij59] developed an $O(n^2)$ algorithm for single pair shortest path, where n is the number of vertices in the graph. In fact, Dijkstra's algorithm finds shortest paths from a given vertex to all the vertices in the graph. See Figure 4.6 for a description of the shortest path algorithm. Figure 4.7(a) shows an edge weighted graph while Figure 4.7(b) shows the shortest path between vertices B and F found by Dijkstra's algorithm.

2. **All Pairs Shortest Paths:** This problem is a variant of SPSP, in which the shortest path is required for all possible pairs in the graph. There are a few variations of all pairs shortest path algorithms for directed graphs. Here we discuss the Floyd-Warshall algorithm which runs in $O(|V|^3)$ time and is based on a dynamic programming technique.

   The algorithm is based on the following observation. Given a directed graph $G = (V, E)$, let $V = \{v_1, v_2, \ldots, v_n\}$. Consider a subset $V' = \{v_1, v_2, \ldots, v_k\} \subseteq V$ for some k. For any pair of vertices $v_i, v_j \in V$, consider all paths from v_i to v_j with intermediate vertices from V' and let p be the the one with minimum weight (an *intermediate vertex* of a path $p = (v_1, v_2, \ldots, v_l)$ is any vertex of p other than v_1 and v_l). The Floyd-Warshall algorithm exploits the relationship between path p and the shortest path from v_i to v_j with intermediate vertices from $\{v_1, v_2, \ldots, v_{k-1}\}$. Let $d_{ij}^{(k)}$ be the weight of a shortest path from v_i to v_j with all intermediate vertices from $\{v_1, v_2, \ldots, v_k\}$. For $k = 0$, a path from v_i to v_j is one with no intermediate vertices, thus having at most one edge, hence $d_{ij}^{(0)} = wt(v_i, v_j)$. A recursive formulation of all pairs shortest path problem can therefore be given as:

$$d_{ij}^{(k)} = \begin{cases} wt(v_i, v_j) & \text{if } k = 0 \\ \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} & \text{if } k \geq 1 \end{cases}$$

   The all pairs shortest path problem allows many paths to share an edge. If we restrict the number of paths that can use a particular edge, then the all pairs shortest path problem becomes NP-hard. The all pairs shortest path problem plays a key role in the global routing phase of physical design.

```
Algorithm SHORTEST-PATH(u)
begin
for i = 1 to n do
    if ((u, i) ∈ E) then D[i] = wt(u, i);
    else D[i] = +∞;
    P[i] = u;
V' = V − u; D[u] = 0;
while (|V'| > 0) do
    Select v such that D[v] = min_{w∈V'} D[w];
    V' = V' − v;
    for w ∈ V' do
        if (D[w] > D[v] + wt(v, w)) then
            D[w] = D[v] + wt(v, w);
            (* D[w] is the length of the shortest path from u to w. *)
            P[w] = v;
            (* P[w] is the parent of w. *)
for w ∈ V do
    (* print the shortest path from w to u. *)
    q = w;
    print q;
    while (q ≠ u) do
        q = P[q];
        print q;
    print q;
end.
```

Figure 4.6: Algorithm SHORTEST-PATH.

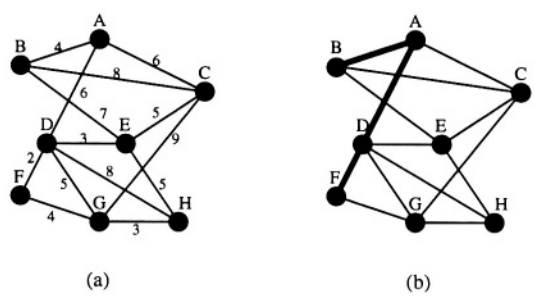


Figure 4.7: Single pair shortest path between vertices B and F.

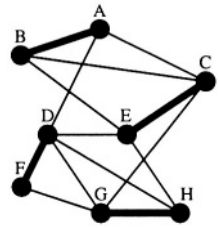


Figure 4.8: Matching.

4.3.1.4 Matching Algorithms

Given an undirected graph $G = (V, E)$, a *matching* is a subset of edges $E' \subseteq E$ such that for all vertices $v \in V$, at most one edge of E' is incident on v. A vertex is said to be *matched* by matching E' if some edge in E' is incident on v; otherwise v is unmatched. A *maximum matching* is a matching with maximum cardinality among all matchings of a graph, i.e., if E' is a maximum matching in G, then for any other matching E'' in G, $|E'| \geq |E''|$. Figure 4.8 shows an example of matching. A matching is called a *bipartite matching* if the underlying graph is a bipartite graph. Both matching and bipartite matching have many important applications in VLSI physical design. The details of different matching algorithms may be found in [PS82].

4.3.1.5 Min-Cut and Max-Cut Algorithms

Min-cut and Max-cut are two frequently used graph problems which are related to partitioning the vertex set of a graph.

The simplest Min-cut problem can be defined as follows: Given a graph $G = (V, E)$, partition V into subsets V_1 and V_2 of equal sizes, such that the number of edges $E' = \{(u, v) | u \in V_1, v \in V_2\}$ is minimized. The set E' is also referred to as a *cut.* A more general min-cut problem may specify the sizes of subsets, or it may require partitioning V into k different subsets. The min-cut problem is NP-complete [GJ79]. Min-cut and many of its variants have several applications in physical design, including partitioning and placement.

The Max-cut problem can be defined as follows: Given a graph $G = (V, E)$, find the maximum bipartite graph of G. Let $G' = (V, E')$ be the maximum bipartite of G, which is obtained by deleting K edges of G, then G has a max-cut of size $|E| - K$.

Max-cut problem is NP-complete [GJ79]. Hadlock [Had75] presented an algorithm which finds max-cut of a planar graph. The algorithm is formally presented in Figure 4.9. Procedure PLANAR-EMBED finds a planar embedding of G, and CONSTRUCT-DUAL creates a dual graph for the embedding. Procedure CONSTRUCT-WT-GRAPH constructs a complete weighted graph by using only vertices corresponding to odd faces. The weight on the edge (u, v) indicates the length of the shortest path between vertices u and v in G_F.

Algorithm MAXCUT
begin
F = PLANAR-EMBED(G);
Let $f_1, f_2, \ldots, f_k$ be the faces of the F;
G_F = CONSTRUCT-DUAL(F);
Let R be the set of vertices of odd degree in G_F and let
Q be the set of all pairs consisting of vertices in R;
G_W = CONSTRUCT-WT-GRAPH(R, Q);
M = MIN-WT-MATCHING(G_W);
Using M find a set of paths, one for each of the matched pairs
in G_F;
Each path determines a set of edges whose deletion leaves the
graph G bipartite;
end.

Figure 4.9: Algorithm MAXCUT.

Note that the number of odd faces in any planar graph is even. Procedure MIN-WT-MATCHING pairs up the vertices in *R*. Each edge in matching represents a path in *G*. This path actually passes through even faces and connects two odd faces. All edges on the path are deleted. Notice that this operation creates a large even face. This edge deletion procedure is repeated for each matched edge in *M*. In this way, all odd faces are removed. The resulting graph is bipartite.

Consider the example graph shown in Figure 4.10(a). The dual of the graph is shown in Figure 4.10(b). The minimum weight matching of cost 4 is (3, 13) and (5, 10). The edges on the paths corresponding to the matched edges, in *M*, have been deleted and the resultant bipartite graph is shown in Figure 4.10(d).

4.3.1.6 Steiner Tree Algorithms

Minimum cost spanning trees and single pair shortest paths are two edge selection problems which can be solved in polynomial time. Surprisingly, a simple variant of these two problems, called the Steiner minimum tree problem, is computationally hard.

The Steiner Minimum Tree (SMT) problem can be defined as follows: Given an edge weighted graph $G = (V, E)$ and a subset $D \subseteq V$, select a subset $V' \subseteq V$, such that $D \subseteq V'$ and V' induces a tree of minimum cost over all such trees. The set D is referred to as the set of *demand points* and the set $V' - D$ is referred to as *Steiner points*. In terms of VLSI routing the demand points are the net terminals. It is easy to see that if $D = V$, then SMT is equivalent to MST, on the other hand, if $|D| = 2$ then SMT is equivalent to SPSP. Unlike MST and SPSP, SMT and many of its variants are NP-complete [GJ77]. In view of the NP-completeness of the problem, several heuristic algorithms have

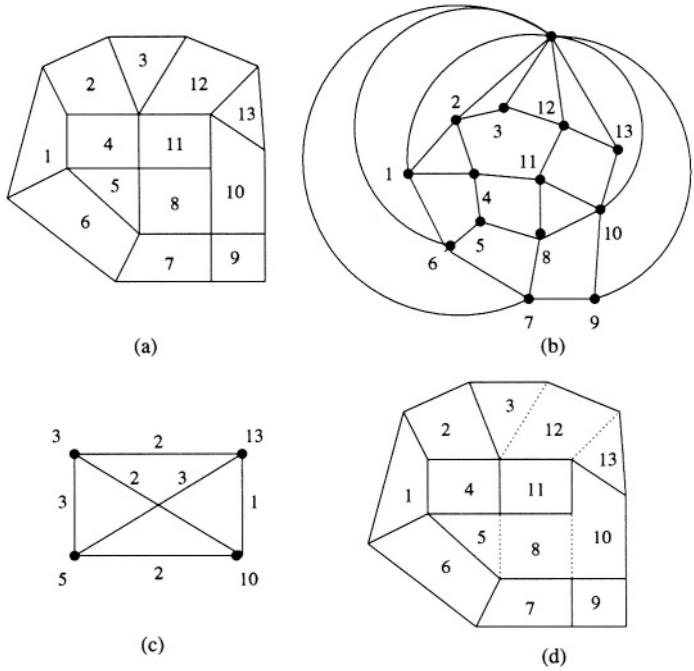


Figure 4.10: An example of finding a max-cut for a planar graph.

been developed.

Steiner trees arise in VLSI physical design in routing of multi-terminal nets. Consider the problem of interconnecting two points in a plane using the shortest path. This problem is similar to the routing problem of a two terminal net. If the net has more than two terminals then the problem is to interconnect all the terminals using minimum amount of wire, which corresponds to the minimization of the total cost of edges in the Steiner tree. The global and detailed routing of multi-terminal nets is an important problem in the layout of VLSI circuits. This problem has traditionally been viewed as a Steiner tree problem [CSW89, HVW85]. Due to their important applications, Steiner trees have been a subject of intensive research [CSW89, GJ77, Han76, HVW85, HVW89, Hwa76b, Hwa79, LSL80, SW90]. Figure 4.11(b) shows a Steiner tree connecting vertices A, I, F, E, and G of Figure 4.11 (a).

The *underlying grid graph* is the graph defined by the intersections of the horizontal and vertical lines drawn through the demand points. The problem is then to connect terminals of a net using the edges of the underlying grid graph. Figure 4.12 shows the underlying grid graph for a set of four points. Therefore Steiner tree problems are defined in the Cartesian plane and edges are restricted to be rectilinear. A Steiner tree whose edges are constrained to rectilinear shapes is called a *Rectilinear Steiner Tree* (RST). A *Rectilinear*

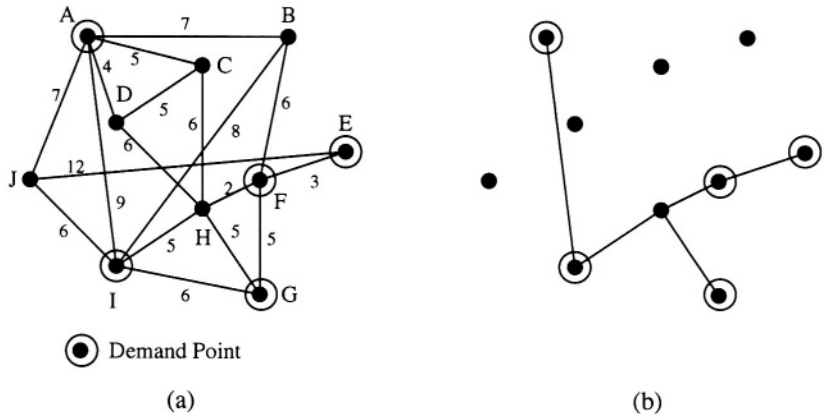


Figure 4.11: A Steiner tree.

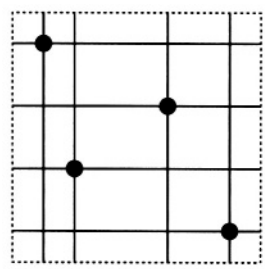


Figure 4.12: The formation of the underlying grid.

Steiner Minimum Tree (RSMT) is an RST with minimum cost among all RSTs. In fact, the MST and RSMT have an interesting relationship. In the MST, the cost of the edges are evaluated in the rectilinear metric. The following theorem was proved by Hwang [Hwa76b].

Theorem 2 *Let $COST_{MST}$ and $COST_{RSMT}$ be the costs of a minimum cost rectilinear spanning tree and rectilinear Steiner minimum tree, respectively. Then*

$$\frac{COST_{MST}}{COST_{RSMT}} \leq \frac{3}{2}$$

As a result, many heuristic algorithms use MST as a starting point and apply local modifications to obtain an RST. In this way, these algorithms can guarantee that weight of the RST is at most $\frac{3}{2}$ of the weight of the optimal tree [HVW85, Hwa76a, Hwa79, LSL80]. Consider the example shown in Figure 4.13. In Figure 4.13(a), we show a minimum spanning tree for the set of

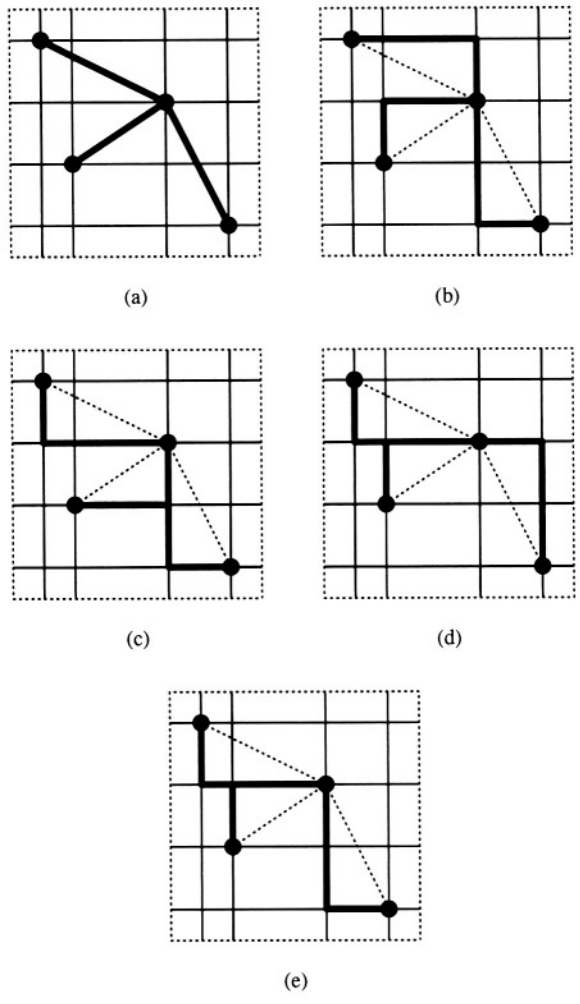


Figure 4.13: Example of different Steiner trees constructed from a minimum cost spanning.

four points. Figure 4.13(b), (c), (d), and (e) show different Steiner trees that can be obtained by using different layouts of edges of spanning tree. Layout of edges should be selected so as to maximize the overlap between layouts and hence minimize the total length of the tree. Figure 4.13(e) shows a minimum cost Steiner tree.

4.3.2 Computational Geometry Algorithms

One of the basic tasks in computational geometry is the computation of the line segment intersections. Investigations to solve this problem have continued for several decades, with the domain expanding from simple segment intersections to intersections between geometric figures. The problem of detecting these types of intersections has practical applications in several areas, including VLSI physical design and motion-planning in robotics.

4.3.2.1 Line Sweep Method

The detailed description of line sweep method and its many variations can be found in [PS85]. A brief description of the line sweep method is given in this section. The n line segments are represented by their $2n$ endpoints, which are sorted by increasing x-coordinate values. An imaginary vertical *sweep line* traverses the endpoint set from left to right, halting at each x-coordinate in the sorted list. This sweep line represents a listing of segments at a given x-coordinate, ordered according to their y-coordinate value. If the point is a left endpoint, the segment is inserted into a data structure which keeps track of the ordering of the segment with respect to the vertical line. The inserted segment is checked with its immediate top and bottom neighbors for an intersection. An intersection is detected when two segments are consecutive in order. If the point is a right endpoint, a check is made to determine if the segments immediately above and below it intersect; then this segment is deleted from the ordering. This algorithm halts when it detects one intersection, or has traversed the entire set of endpoints and no intersection exists. Consider the example shown in Figure 4.15. The intersection between segments A and C will be detected when segment B is deleted and A and C will become consecutive. A description of the line sweep algorithm is given in Figure 4.14.

The sorting of $2n$ endpoints can be done in $O(n \log n)$ time. A balanced tree structure is used for T, which keeps the y-order of the active segments; this allows the operations *INSERT, DELETE, ABOVE,* and *BELOW* to be performed in $O(n \log n)$ time. Since the for loop executes at most $2n$ times, the time complexity of the algorithm is $O(n \log n)$.

4.3.2.2 Extended Line Sweep Method

The line sweep algorithm can be extended to report all K intersecting pairs found among n line segments. The extended line sweep method performs the line sweep with the vertical line, inserting and deleting the segments in the tree R; but when a segment intersection is detected, the point of intersection

```
Algorithm LINE-SWEEP
begin
    Sort the endpoints lexicographically on
        x-coordinates so that Point[1] is leftmost and
        Point[2n] is rightmost;
    for i = 1 to 2n do
        (* Let S be the segment of which P is an endpoint *)
        P = Point[i];
        if P is the left endpoint of S then
            INSERT (S, T);
            A = ABOVE(S, T);
            B = BELOW(S, T);
            if A intersects S then return (A, S);
            if B intersects S then return (B, S);
        else        (* P is the right endpoint of S *)
            A = ABOVE(S, T);
            B = BELOW(S, T);
            if A intersects B then return (A, B);
            Delete(S, T);
end.
```

Figure 4.14: Algorithm LINE-SWEEP.

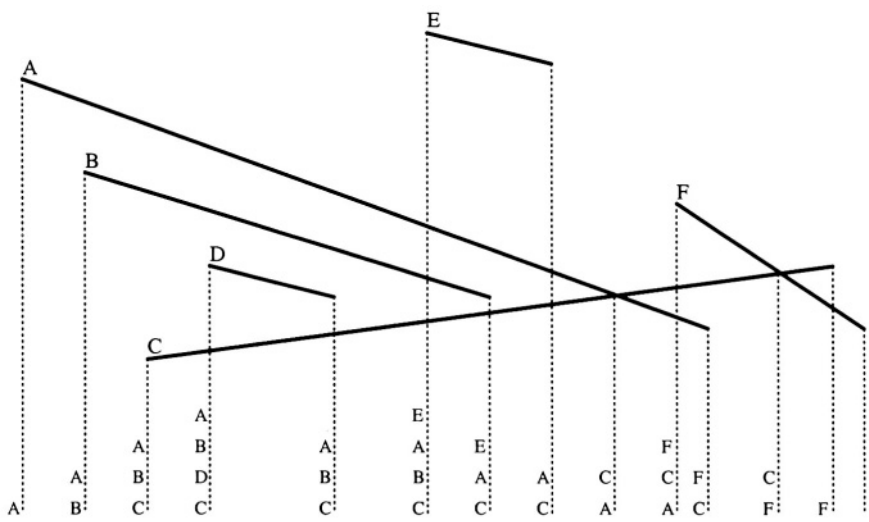


Figure 4.15: Line sweep example.

is inserted into the heap Q of sorted endpoints, in its proper x-coordinate value order. The sweep line will then also halt at this point, the intersection is reported, and the order of the intersecting segments in R is swapped. New intersections between these swapped segments and their nearest neighbors are checked and points inserted into Q if this intersection occurs. This algorithm halts when all endpoints and intersections in Q have traversed by the sweep line, and all intersecting pairs are reported.

The special case in which the line segments are either all horizontal or vertical lines was also discussed. A horizontal line is specified by its y-coordinate and by the x-coordinates of its left and right endpoints. Each vertical line is specified by its x-coordinate and by the y-coordinates of its upper and lower endpoints. These points are sorted in ascending x-coordinates and stored in Q.

The line sweep proceeds from left to right; when it encounters the left endpoint of a horizontal segment S, it inserts the left endpoint into the data structure R. When a vertical line is encountered, we check for intersections with any horizontal segments in R, which lie within the y-interval defined by the vertical line segment.

4.4 Basic Data Structures

A layout editor is a CAD tool which allows a human designer to create and edit a VLSI layout. It may have some semi-automatic features to speed up the layout process. Since layout editors are interactive, their underlying data structures should enable editing operations to be performed within an acceptable response time. The data structures should also have an acceptable space complexity, as the memory on workstations is limited.

A layout can be represented easily if partitioned into a collection of *tiles.* A tile is a rectangular section of the layout within a single layer. The tiles are not allowed to overlap within a layer. The elements of a layout are referred to as *block tiles.* A block tile can be used to represent p-diffusion, n-diffusion, poly segment, etc. For ease of presentation, we will refer to a block tile simply as a *block.* The area within a layout that does not contain a block is referred to as *vacant space.* Figure 4.16 shows a simple layout containing several blocks. Later, in the chapter we will introduce a method that partitions the vacant space into a series of *vacant tiles.*

4.4.1 Atomic Operations for Layout Editors

The basic set of operations that give a designer the freedom to fully manipulate a layout, is referred to as the *Atomic Operations.* The following is the list of atomic operations that a layout editor must support.

1. **Point Finding:** Given the coordinate of a point $p = (x, y)$, determine whether p lies within a block and, if so, identify that block.
2. **Neighbor Finding:** This operation is used to determine all blocks touching a given block B.

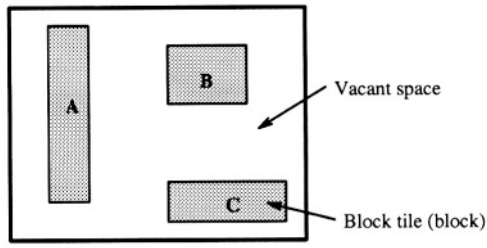


Figure 4.16: Block tile (block) representation in a layout.

3. **Block Visibility:** This operation is used to determine all blocks visible, in the x and y direction, from a given block B. Note that this operation is different from the neighbor finding operation.

4. **Area Searching:** Given a fixed area A, defined by its upper left corner (x,y), the length l, and the width w, determine whether A intersects with any blocks B. This operation is very useful in the placement of blocks. Given a block to be placed in a particular area, we need to check if other blocks are currently residing in that area.

5. **Directed Area Enumeration:** Given a fixed area A, defined by its upper left corner (x,y), length l, and width w, visit each block intersecting A exactly once in sorted order according to its distance from a given side (top, bottom, left, or right) of A.

6. **Block Insertion:** Block insertion refers to the act of inserting a new block B into the layout such that B does not intersect with any existing block.

7. **Block Deletion:** This operation is used to remove an existing block B from the layout. In an iterative approach to placement, blocks are moved from one location to another. This is done by inserting the block into a new location and then deleting the block at its previous location.

8. **Plowing:** Given an area A and a direction d, remove all blocks B_i from A by shifting each B_i in direction d while preserving ordering between the blocks.

9. **Compaction:** Compaction refers to plowing or "compressing" of the entire layout. If compaction is along the x-axis or y-axis then the compaction is called *1-dimensional compaction.* When the compaction is carried out in both x- and y-direction then it is called *2-dimensional compaction.*

10. **Channel Generation:** This operation refers to determining the vacant space in the layout and partitioning it into tiles.

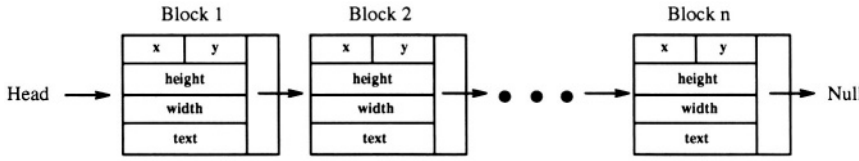


Figure 4.17: Linked list representation.

4.4.2 Linked List of Blocks

The simplest data structure used to store the components of a layout is a linked list, where each node in the list represents a block. The linked list representation is shown in Figure 4.17. Notice that the blocks are not stored in any particular order, as none was specified in the original description of the data structure; however, it is clear that a sorted or self-organizing list will improve average algorithmic complexity for some of the atomic operations. The space complexity of the linked list method is $O(n)$ where n is the number of blocks in the layout.

For illustration purpose, we now present an algorithm for neighbor finding using the linked list data structure. Given a block B, the neighbors of B are all the blocks that share a side with B. Each block is represented in the list by its location (coordinate of upper left corner), height, width, text to describe the block, and a pointer to the next node (block) in the list. The algorithm finds the neighbors on the right side of the given block, however, the algorithm can be easily modified to find all the neighbors of a given block. The input to the algorithm is a specific block B, and the linked list L of all blocks. A formal description of the algorithm for neighbor finding is shown in Figure 4.18.

Linked list data structures are suitable for a hierarchical system since each level of hierarchy contains few blocks. However, this data structure is not suitable for non-hierarchical systems and for hierarchical systems with a large number of blocks in each level. The major disadvantage of this structure is that it does not explicitly represent the vacant space. However, the data structure can be altered to form a new representation of the layout in which the vacant space is stored as a collection of *vacant tiles.* A vacant tile maintains the same geometric restrictions that are implied by the definition of a tile. Converting the vacant space into a collection vacant tiles can be done by extending the upper and lower boundaries of each block horizontally to the left and to the right until it encounters another block or the boundary of the layout as shown in figure 4.19. This partitions the entire area into a collection of tiles (block and vacant), organizing the vacant tiles into *maximal horizontal strips,* thus allowing the entire area of the layout to be represented in the linked list data structure. We call this data structure the *modified linked list.*

```
Algorithm NEIGHBOR-FIND1(B, L)
begin
    neighbor-list = φ;
    x1 = B.x + B.width;
    y11 = B.y;
    y12 = B.y + B.height;
    for each block R ∈ L such that R ≠ B do
        y21 = R.y;
        y22 = R.y + R.height;
        if (x1 = R.x and (y11 ≤ y21 ≤ y12 or
                          y11 ≤ y22 ≤ y12 or
                          y21 < y11 < y12 < y22)) then
                INSERT(R, neighbor-list);
    return neighbor-list;
end.
```

Figure 4.18: Algorithm NEIGHBOR-FIND1.

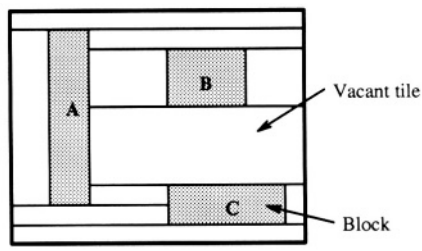


Figure 4.19: Vacant tile creation.

4.4.3 Bin-Based Method

The bin-based data structure does not keep information pertaining to the vacant space and does not create vacant tiles. In the bin-based system, a virtual grid is superimposed on the layout area as shown in Figure 4.20. The grid divides the area into a series of bins which can be represented using a two-dimensional array. Each element in the array ($\mathcal{B}$(row,col)) contains all the blocks that intersect with the corresponding bin. In Figure 4.20, $\mathcal{B}$(2,3) contain blocks D,E,F,G, and H while $\mathcal{B}$(2,4) contain blocks G and H. The space complexity for the bin-based data structure is $O(bn)$, where b denotes the total number of bins and n is the number of blocks.

Clearly, the bin-based data structure can be viewed as an augmented version of the linked list data structure in which a time-space compromise has been made to improve the average case performance on several of the atomic

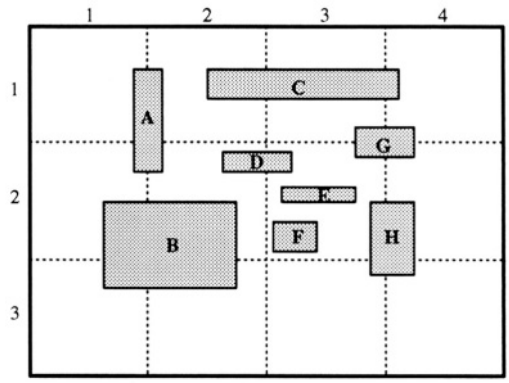


Figure 4.20: Bin-based representation.

operations. However, it is easy to construct pathological examples which cause the worst-case performance of the bin-based structure to degenerate to that of the linked list. This is possible since the bin size is fixed, while the block size may vary. If we insert n blocks into the layout such that all the blocks fall in the same bin, the performance of the bin-based data structure is equivalent to that of the linked list for most atomic operations, and worse than the linked list in the neighbor finding, area searching, and directed area enumeration operations, since bins containing no blocks must be tested. The worst case complexity for these operations is $O(b + n)$.

As in the linked list data structure, we now present an algorithm to find the neighbors of a given block. This algorithm finds the neighbors on the right side of the given block. However, the algorithm can easily be extended to find all the neighbors of a given block. The input to the algorithm is a specific block A and the set of bins $\mathcal{B}$. A formal description of the algorithm is shown in Figure 4.21.

In general, the bin-based data structure is highly sensitive to the time-space tradeoff. For instance, if the bins are small with respect to the average size of a block, the blocks are likely to intersect with more than one bin, thereby increasing storage requirements. Furthermore, many bins may remain empty, creating wasted storage space. If the bins are too large the average case performance will be reduced since the linked lists used to store blocks in each bin will be very long. Obviously, the best case is when each bin contains exactly $\frac{n}{b}$ blocks, and no block is stored in more than one bin.

Even though the bin-based method can be used to locate all blocks within an area (bin), it does not allow for any representation of locality. In order to find the block closest to another, it may be necessary to search other surrounding bins, and in the worst case all the bins. Because the bin-based data structure does not represent the vacant space, operations such as compaction are tedious and time consuming.

```
Algorithm NEIGHBOR-FIND2(A, B)
begin
    neighbor-list = φ;
    x1 = A.x + A.width;
    y11 = A.y;
    y12 = A.y + A.height;
    let B' ⊆ B be set of bins which contain A;
    for all bins X ∈ B' do
        for each block R ∈ X do
            y21 = R.y;
            y22 = R.y + R.height;
            if (x1 = R.x and (y11 ≤ y21 ≤ y12 or
                              y11 ≤ y22 ≤ y12 or
                              y21 < y11 < y12 < y22)) then
                INSERT(R, neighbor-list);
    return neighbor-list;
end.
```

Figure 4.21: Algorithm NEIGHBOR-FIND2.

4.4.4 Neighbor Pointers

Most operations in a layout system require local block information to perform efficiently. Both the linked list and bin-based data structures do not keep local information, such as neighboring blocks. To overcome this limitation, the *neighbor pointer* data structure was developed. The neighbor pointer data structure represents each block by its size (upper left hand corner, length, and width) as well as the pointers to all of its neighbors. The space complexity of the data structure is bounded by $O(n^2)$. Figure 4.22 shows how neighbor pointers are maintained for Block A.

The neighbor pointer data structure is designed to perform well on plowing and compaction operations, unlike the linked list and bin-based structures. Plowing operation can be performed easily since each block directly stores information about its neighbors. In other words, for any block B_i, all blocks affected by moving B_i can be referenced directly. Since compaction is a form of plowing, it can also be performed easily using the neighbor pointer data structure. Figure 4.23, shows how the neighbor pointers of block A are updated when block B is moved.

The primary disadvantage of neighbor pointers is that the data structure is difficult to maintain. A simple modification to the layout may require all the pointers in the data structure be updated. For instance, a plow operation may modify the neighbors of each block B_i, and, in this case, updating the pointers could take as much as $O(n^2)$ time. Furthermore, block insertion and deletion operations each take $O(n)$ time. Since vacant space is not explicitly represented,

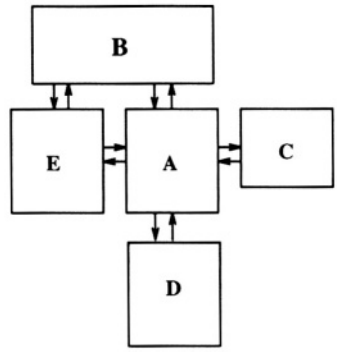


Figure 4.22: Neighbor pointers.

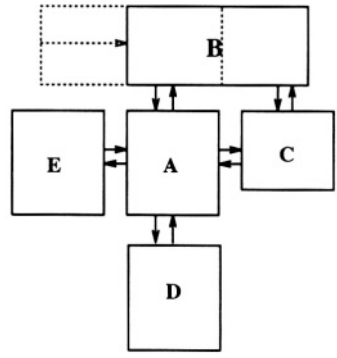


Figure 4.23: Update of neighbor pointers.

channel generation cannot be performed without extensive modification to the data structure.

4.4.5 Corner Stitching

Corner stitching is a radically different data structure used for IC layout editing [Ous84]. Corner stitching is novel in the sense that it is the first data structure to represent both vacant and block tiles in the design. As in the neighbor pointer structure, information about the relative locations of blocks is stored; however, unlike neighbor pointers, the corner stitch data structure can be updated rapidly.

The corner-stitch data structuring technique provides various powerful operations such as stretching, compaction, neighbor-finding, and channel finding. These operations are possible in the order of the number of neighbors, which

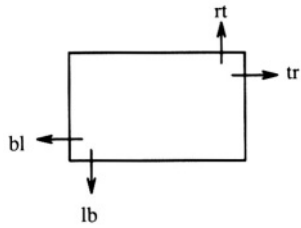


Figure 4.24: Corner stitches.

in the worst case would be the order of the size of the layout; i.e., the number of objects in the layout. The advantage of corner stitch data structure is that it permits easy modification to the layout.

The two main features of the corner stitch data structure pertain to the way in which it keeps track of vacant tiles and how the tiles are linked. To partition the vacant space into a collection of vacant tiles, the vacant space must be divided into maximal horizontal strips (as discussed in section 4.4.2). Hence the whole layout is represented as tiles (vacant and block).

Tiles are linked by a set of pointers called *corner stitches.* Each tile contains four stitches, two at its top right corner and two at the bottom left corner as shown in Figure 4.24. The pointers at these two corners are sufficient to perform all operations. The corner stitch method stores both the vertical and horizontal pointers. Each tile also stores the same number of pointers, irrespective of the number of neighbors it has. In this structure, vacant tiles can assume any size, and this helps in naturally adapting to the variations in the size of the blocks. In other words, a new block, created over a set of a vacant tiles, will result in a number of vacant tiles to be split, thus enabling the layout to be updated easily. The pointers in each of the four directions provide a type of sorting similar to that of the neighbor pointers. Figure 4.25, shows how corner stitches link the tiles of a layout. The stitches exceeding the boundary of the layout have been omitted in the figure, but the data structure represents them as NULL pointers.

The record structure used in this section to represent a tile is the same as the one shown in Figure 4.17 with the exception that the single pointer used to link the tiles will be replaced with the corner stitch pointers (rt, tr, bl, lb). It should also be noted that we define the upper left corner of the layout to be point (0,0).

In the following we present the various operations that can be performed on a layout using the corner stitch data structure.

1. **Point Finding:** a point p_2, the following sequence of steps finds a path through the corner stitches from the current point p_1 to p_2 traversing the minimum number of tiles.

   (1) The first step is to move up or down, using **rt** and **lb** pointers until

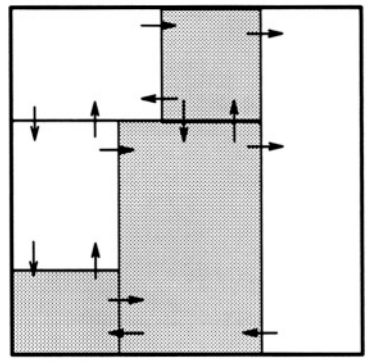


Figure 4.25: An example of the corner stitch layout.

```
Algorithm POINT-FIND(B, x, y)
begin
   do
      while (y < B.y) or (y > B.y + B.height) do
         if y < B.y then B = B.rt; else B = B.lb;
      while (x < B.x) or (x > B.x + B.width) do
         if x < B.x then B = B.bl; else B = B.tr;
   while (y < B.y) or (y > B.y + B.height);
end.
```

Figure 4.26: Algorithm POINT-FIND.

a tile is found whose vertical range contains the destination point.

(2) Then, a tile is found whose horizontal range contains the destination point by moving left or right using **tr** or **bl** pointers.

(3) Whenever there is a misalignment (the search goes out of the vertical range of the tile that contains the destination point) due to the above operations, steps 1 and 2 has to be iterated several times to locate the tile containing the point.

This operation is illustrated in Figure 4.27. In worst case, this algorithm traverses all the tiles in the layout. On an average though, $\sqrt{N}$ tiles will be visited. This algorithm handles the inherent asymmetry in designs by readjusting the misalignments that occur during the search.

Figure 4.26 shows a formal description of the algorithm for point finding. The input to the algorithm is a specific block B, and the coordinates of the desired point (x,y).

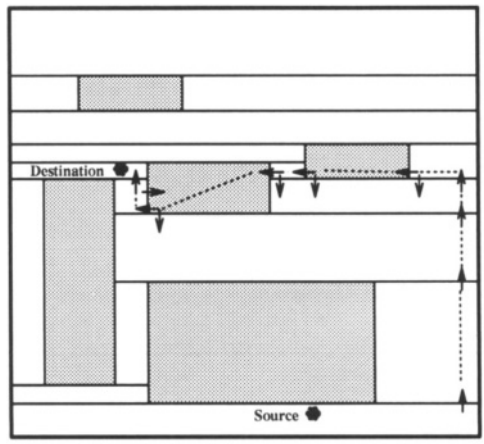


Figure 4.27: Point finding using corner stitches.

2. **Neighbor Finding:** Following algorithm finds all the tiles that touch a given side of a given tile. This is also illustrated in Figure 4.28.

   (1) From the **tr** pointer of the given tile the algorithm starts traversing using the **lb** pointer downwards until it reaches a tile which does not completely lie within the vertical range of the given tile.

3. **Area Search:** Given an area, the following algorithm reports if there are any blocks in the area. This is illustrated in Figure 4.29.

   (1) First the tile in which the upper left corner of the given area is located.

   (2) If the tile corresponding to this corner is a space tile, then if its right edge is within the area of interest, the adjacent tile must be a block.

   (3) If a block was found in step 2, then the search is complete. If no block was found, then the next tile touching the right edge of the area of interest is found, by traversing the **lb** stitches down and then traversing right using the **tr** stitches.

   (4) Steps 2 and 3 are repeated until either the area has been searched or a block has been found.

4. **Enumerate all Tiles:** Given an area, the following algorithm reports all tiles intersecting that area. This is illustrated in Figure 4.30.

   (1) The algorithm first finds the tile in which the upper left corner of the given area is located. Then it steps down through all the tiles along the left edge, using the same technique as in area searching.

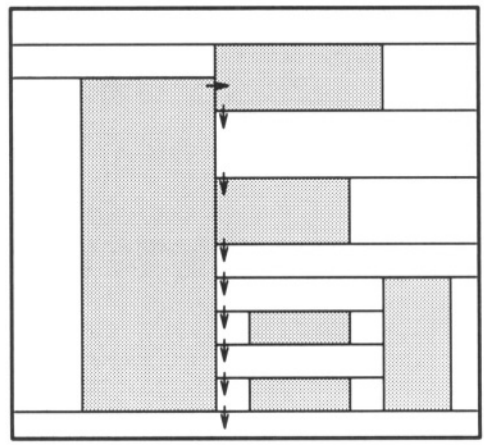


Figure 4.28: Neighbor finding using corner stitches.

(2) The algorithm enumerates all the tiles found in step 1 recursively (one tile at a time) using the procedure given in lines (R1) through (R5).

(R1) Number the current tile (this will generally involve some application specific processing).

(R2) If the right edge of the tile is outside of the search area, then the algorithm returns from the R procedure.

(R3) Otherwise, the algorithm uses the neighbor-finding algorithm to locate all the tiles that touch the right side of the current tile and also intersect the search area.

(R4) For each of these neighbors, if the bottom left corner of the neighbor touches the current tile then it calls R to enumerate the neighbor recursively (for example, this occurs in Figure 4.30 when tile 1 is the current tile and tile 2 is the neighbor).

(R5) Or, if the bottom edge of the search area cuts both the current tile and the neighbor, then it calls R to enumerate the neighbor recursively (in Figure 4.30, this occurs when tile 8 is the current tile and tile 9 is the neighbor).

5. **Block Creation:** The algorithm given below creates a block of given height and width on a certain given location in the plane. An illustration of how the vacant tiles will change when block E is added to the layout is given in Figure 4.31. Notice that the vacant tiles remain in maximal horizontal strips after the block has been added.

(1) First of all the algorithm checks if a block already exists by using the area search algorithm.

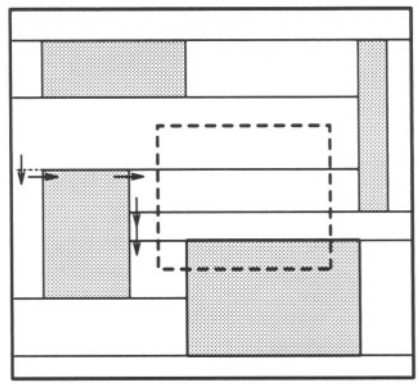


Figure 4.29: Area search using corner stitches.

(2) It then finds the vacant tile containing the top edge of the area occupied by new tile.

(3) The tile found in step (2) is split using the horizontal line along the top edge of the new tile. In addition, the corner stitches of the tiles adjoining the new tile are updated.

(4) The vacant tile containing the bottom edge of the new block is found and split in the same fashion as in step (3) and corner stitches of the tiles adjoining adjoining the new tile are updated.

(5) The algorithm traverses down along the left side and right side of the area of the new tile respectively, using the same technique in step (3) and updates corner stitches of the tiles as necessary.

6. **Block Deletion:** The following algorithm deletes a block at a given location in the plane. An illustration of how the vacant tiles will change when block C is deleted from the layout is given in Figure 4.32. Notice that the vacant tiles remain in maximal horizontal strips after the block has been deleted.

(1) First, the block to be deleted is changed to a vacant tile.

(2) Second, using the neighbor finding algorithm for the right edge of the deleted tile find all the neighbors. For each vacant tile neighbor, the algorithm either splits the deleted tile or the neighbor tile so that the two tiles have the same vertical span and then merges them horizontally.

(3) Third, find all the neighbors for the left edge of the deleted tile. For each vacant tile neighbor the algorithm either splits the deleted tile or the neighbor tile so that the two tiles have the same vertical span and then merges them horizontally. After each horizontal merge, it

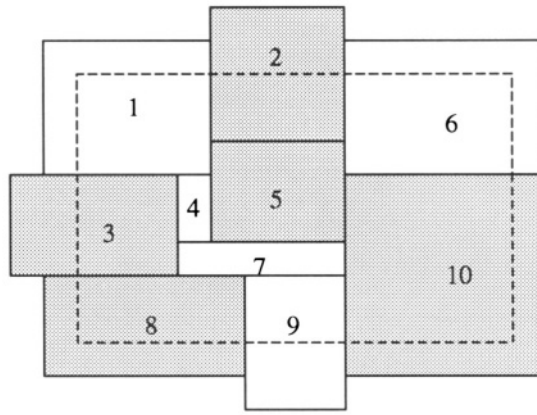


Figure 4.30: Enumerate all tiles using corner stitches.

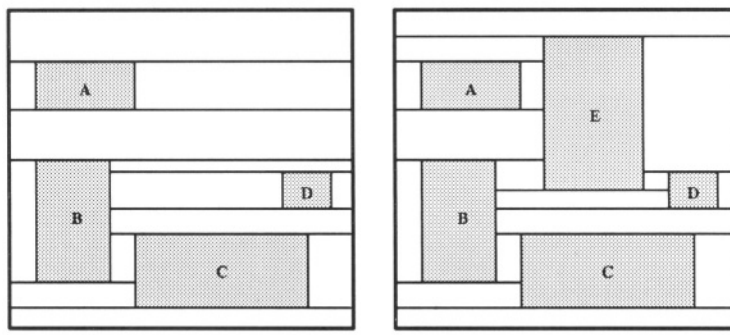


Figure 4.31: Block creation using corner stitches.

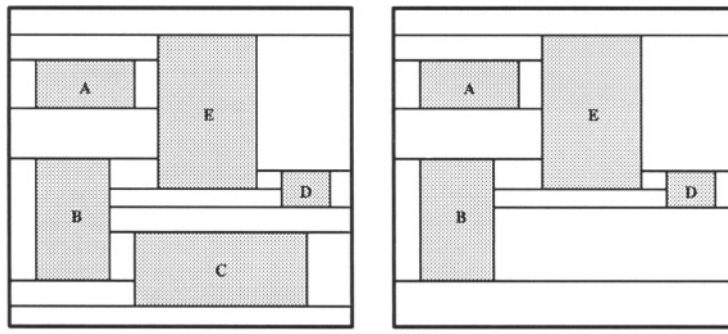


Figure 4.32: Block deletion using corner stitches.

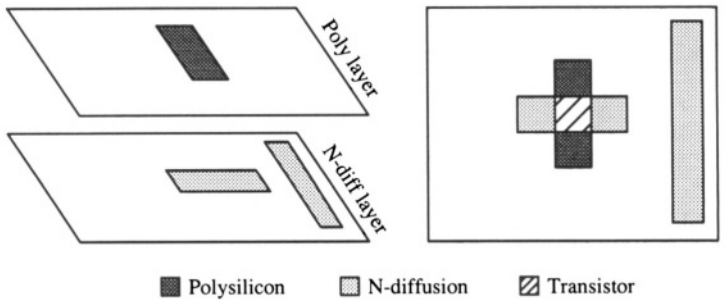


Figure 4.33: A simple two-layer layout.

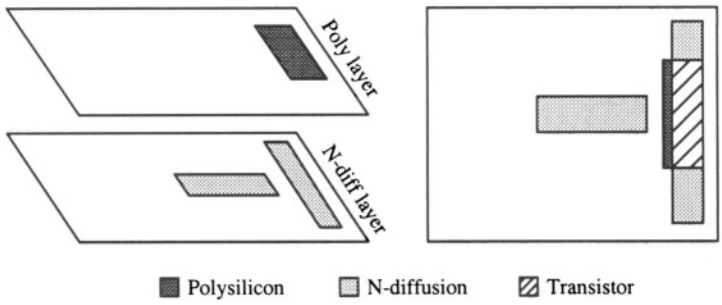


Figure 4.34: Simple two-layer layout plowed along the x-axis.

performs a vertical merge if possible with the tiles above and below it.

4.4.6 Multi-layer Operations

Thus far, the operations have been limited to those that concern only a single layer. It is important to realize that layouts contain many layers. More importantly, the functionality of the layout depends on the relationship between the position of the blocks on different layers. For instance, Figure 4.33 shows a simple transistor formed on two layers (polysilicon and n-diffusion) accompanied by a separate segment on the n-diffusion layer. If each layer is plowed along the positive x-axis, as shown in Figure 4.34, the original function of the layout has been altered as a transistor has been created on another part of the layout.

Design rule checking is also multilayer operation. In Figure 4.34, an illegal transistor has been formed. Even though the design rules were maintained while plowing on a single layer, the overall layout has an obvious design flaw.

If proper design rule checking is to take place, then the position of the blocks on each layer must be taken into consideration.

4.4.7 Limitations of Existing Data Structures

When examining the different data structures described in this chapter, it is easy to see that there is no single data structure that performs all operations with good space and time complexity. For example, the linked list and bin-based data structures are not suitable for use in a system in which a large database needs to be updated very frequently. Although the simpler data structures, like the linked list, are easy to understand, they are not suitable for most layout editors. On the other hand, more advanced data structures, such as the corner stitch, perform well where the simpler data structures become inefficient. Yet, they do not allow for the full flexibility needed in managing complicated layouts.

A limitation that all the data structures discussed share, is that they only work on rectangular objects. For example, the data structures do not support objects that are circular or L-shaped. New data structures, therefore, need to be developed that handle non-rectangular objects. Also as parallel computation is becoming more popular, new data structures need to be developed that can adapt to a parallel computation environment.

4.4.8 Layout Specification Languages

A common and simple method of producing system layouts is to draw them manually using a layout editor. This is done on one lambda grid using familiar color codes to identify various systems layers. Once the layout has been drawn, it can then be digitized or translated into machine-readable form by encoding it into a symbolic layout language. The function of a symbolic layout language, in its simplest form, is similar to that of macro-assembler. The user defines *symbols* (macros) that describe the layout of basic system cells. The function of the assembler for such a language is to scan and decode the statements and translate them into design files in intermediate form. The effectiveness of such languages could be further increased by constructing an assembler capable of handling nested symbols. Through the use of nested symbols, system layouts may be described in a hierarchical manner, leading to very compact descriptions of structured designs.

Caltech Intermediate Form (CIF) is one of the popular intermediate forms of layout description. Its purpose is to serve as a standard machine-readable representation from which other forms can be constructed for specific output devices such as plotters, video displays, and pattern-generation machines. CIF provides participating design groups easy access to output devices other than their own, enables the sharing of designs, and allows combining several designs to form a larger chip. A CIF file is composed of a sequence of commands, each being separated by a semi-colon (;). Each command is made up of a sequence of characters which are from a fixed character set. Table 4.1 lists the command

Command	Representation
B length width center direction	A Box of length and width(specified by integers) with center and direction(specified by points). Default direction is (1,0)
C symbol transformation	Call symbol(specified by integers)
DD symbol	Delete definition denoted by symbol(specified by integers)
DF	Finish definition
DS symbol index-1 index-2	Start definition denoted by Symbol with index-1 and index-2(specified by integers)
E	End of CIF marker
L shortname	Layer name
P path	Polygon with path
R diameter center	Circle of diameter(specified by integer) with center(specified by a point)
W width path	Wire with width(specified by integer) and path
(comments)	comments enclosed in parenthesis
number usercommand	user extension

Table 4.1: Command symbols and their representations.

symbols and their forms.

A more formal listing of the commands is given in Figure 4.35. The syntax for CIF is specified using a recursive language definition as proposed by [Wir77]. The notation used is similar to the one used to express rules in programming languages and is as follows: the production rules use equals (=) to relate identifiers to expressions; vertical bar (|) for *or*; double quotes (" ") around terminal characters; curly braces ({ }) indicate repetition any number of times including zero; square brackets ([ ]) indicate optional factors (i.e., zero or one repetition); parentheses ( ) are used for grouping; rules are terminated by a period (.).

The number of objects in a design and the representation of the primitive elements make the size of the CIF file very large. Symbolic definition is a way of reducing the file size by equating a commonly used command to a symbol. Layer names have to be unique, which will ensure the integrity of the design while combining several layers which represent one design.

CIF uses a right-handed coordinate system where x increases to the right and y increases upward. CIF represents the entire layout in the first quadrant. The unit of measurement of the distance is usually a micrometer(μm). Below are several examples of geometric shapes expressed in CIF. Note that the corresponding example correspond to the respective shape in Figure 4.36.

(a) **Boxes:** A box of length 30, width 50, and which is centered at (15,25), in CIF is B 30 50 15 25; (See Figure 4.36(a))

In this form the length of the box is the measurement of the side that is parallel to the x-axis and the width of the box is the measurement of the side that is parallel to the y-axis.

(b) **Polygons:** Polygon with vertices at (0,0), (20,50), (50,30), and (40,0) in

Alphabet	Rules
cifFile	= {{blank}[command]semi} endCommand{blank}
command	= primCommmand | defDeleteCommand | defStart Command semi{{blank}[primCommand]semi}defFinishCommand.
primCommand	= polygonCommand | boxCommand | roundFlashCommand | wireCommand | layerCommand | callCommand | userExtensionCommand | commentCommand.
polygonCommand	= "P" path
boxCommand	= "B" integer sep integer sep point [sep point]
roundFlashCommand	= "R" integer sep point
wireCommand	= "W" integer sep path
layerCommand	= "L" {blank} shortname
defStartCommand	= "D" {blank} "S" integer [sep integer sep integer]
defFinishCommand	= "D" {blank} "F"
defDeleteCommand	= "D" {blank} "D" integer
callCommand	= "C" integer transformation
userExtensionCommand	= digit userText
commentCommand	= "("comment Text")"
endCommand	= "E"
transformation	= {{blank} ("T" point | "M" {blank}"X"|"M" {blank} "Y" | "R" point)}
path	= point {sep point}
point	= sInteger sep sInteger
sInteger	= {sep}["-"] integerD
integer	= {sep} integerD
integerD	= digit {digit}
shortname	= c[c][c][c]
c	= digit | upperChar
userText	= {userChar}
commentText	= {commentChar} | commentText "("commentText")"commentText
semi	= {blank} ";"{blank}
sep	= upperChar | blank
digit	= "0"| "1"| "2"| "3"| "4"| "5"| "6"| "7"| "8"| "9"
upperChar	= "A"| "B"| "C"| "D"| "E"| "F"| "G"| "H"| "I"| "J"| "K"| "L"| "M"| "N"| "O"| "P"| "Q"| "R"| "S"| "T"| "U"| "V"| "W"| "X"| "Y"| "Z"
blank	= any ASCII character except digit, upperChar, "-", "(", ")", or";"
userChar	= any ASCII character except ";"
commentChar	= any ASCII character except "(" or ")"

Figure 4.35: Formal list of commands and their protocols.

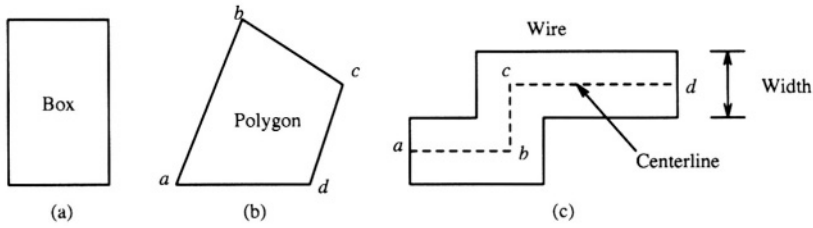


Figure 4.36: CIF terminologies for different geometric shapes.

CIF is P 0 0 20 50 50 30 40 00; (See Figure 4.36(b))

For a polygon with n sides, the coordinates of n vertices must be specified through the path of the edges.

(c) **Wires**: A wire with width 20 and which follows the path specified by the coordinates (0,10), (30,10), (30,30), (80,30) in CIF is W 20 0 10 30 10 30 30 80 30; (See Figure 4.36(c))

For a wire, the width must be given first and then the path of the wire is specified by giving the coordinates of the path along its center. For an object to qualify as a wire, it must have a uniform width.

As shown by the representation of a polygon, CIF will describe shapes that do not have Manhattan or rectilinear features. It is actually possible to represent a box that does not have Manhattan features. This is done using a direction vector. This eliminates the need for any trigonometric functions such as sin, cos, tan, etc. It is also easy to incorporate in the box description. The direction vector is made up two integer components, the first being the component of the direction vector along the x-axis, and the second being the same along the y-axis. The direction vector (1 1) will rotate the box 45° counterclockwise as will (2 2), (50 50), etc. The direction vector pointing to the x-axis can be represented as direction (10). With this new information a new descriptor can be added to box called the direction. Figure 4.37 shows a box with length 25, width 60, center 80,40 and direction -20, 20. When using direction, the length is the measure of the side parallel to the direction vector, and width is the measure of the side perpendicular to the direction vector. The direction vector is optional and if not used defaults to the positive x-axis.

B 25 60 80 40 -20 20;

To maintain the integrity of the layers for these geometric objects they must be labeled with the exact name of the fabrication mask (layer) on which it belongs. Rather than repeating the layers specified for each object, it is specified once and all objects defined after it belong to the same layer.

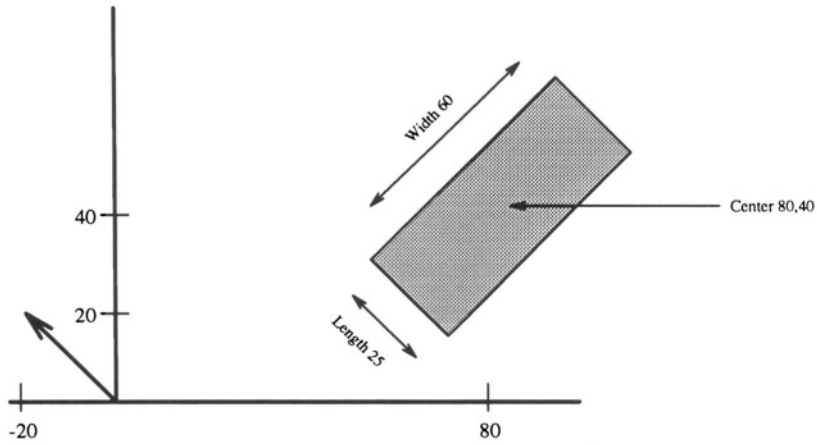


Figure 4.37: Box representation in CIF format.

Using CIF, a cell can be defined as a symbol by using the DS and DF commands. If an instance of a cell is required, the call command for that cell is used. The entire circuit is usually described as a group of nested symbols. A final call command is used to instantiate the circuit.

One of the popular layout description language used in the industry is GDSII.

4.5 Graph Algorithms for Physical design

The basic objects in VLSI design are rectangles and the basic problems in physical design deal with arrangement of these rectangles in a two or three dimensional space. The relationships between these objects, such as overlap and distances, are very critical in development of physical design algorithms. Graphs are a well developed tool used to study relationships between objects. Naturally, graphs are used to model many VLSI physical design problems and they play a very pivotal role in almost all VLSI design algorithms. In this section, we will define various graphs which are used in modeling of physical design problems.

4.5.1 Classes of Graphs in Physical Design

A layout is a collection of rectangles. Rectangles, which are used for routing, are thin and long and the width of these rectangles can be ignored for the sake of simplicity. In VLSI routing problems, such simple models are frequently used where the routing wires are represented as lines. In such cases, one needs to optimally arrange lines in two and three dimensional space. As a result, there are several different graphs which have been defined on lines and their

relationships. Rectangles, which do not allow simplifying assumptions about the width, must also be modeled. For placement and compaction problems, it is common to use a graph which represents a layout as a set of rectangles and their adjacencies and relationships. As a result, a graph may be defined to represent the relationships between the rectangles. Thus we have two types of graphs dealing with lines and rectangles. Complex layouts with non-rectilinear objects require more involved modeling techniques and will not be discussed.

4.5.1.1 Graphs Related to a Set of Lines

Lines can be classified into two types depending upon the alignment with axis. We prefer to use the terminology of line interval or simply interval for lines which are aligned to axis. An interval I_i is represented by its left and right endpoints, denoted by l_i and r_i, respectively. Given a set of intervals $\mathcal{I} = \{I_1, I_2, \ldots, I_n\}$, we define three graphs on the basis of the different relationships between them.

We define an *overlap graph* $G_O = (V, E_O)$, as

$$V = \{v_i \mid v_i \text{ represents interval } I_i \}$$

$$E_O = \{(v_i, v_j) \mid l_i < l_j < r_i < r_j\}$$

In other words, each vertex in the graph corresponds to an interval and an edge is defined between v_i and v_j if and only if the interval I_i overlaps with I_j but does not completely contain or reside within I_j.

We define a *containment graph* $G_C = (V, E_C)$, where the vertex set V is the same as defined above and E_C a set of edges defined below:

$$E_C = \{(v_i, v_j) \mid l_i < l_j, r_i > r_j\}$$

In other words an edge is defined between v_i and v_j if and only if the interval I_i completely contains the interval I_j.

We also define an interval graph $G_I = (V, E_I)$ where the vertex set V is the same as above, and two vertices are joined by an edge if and only if their corresponding intervals have a non-empty intersection. It is easy to see that $E_I = E_O \cup E_C$. An example of the overlap graph for the intervals in Figure 4.38(a) is shown in Figure 4.38(b) while the containment graph and the interval graph are shown in Figure 4.38(c) and Figure 4.38(d) respectively. Interval graphs form a well known class of graphs and have been studied extensively [Gol80].

Overlap, containment and interval graphs arise in many routing problems, including channel routing, single row routing and over-the-cell routing.

If lines are non-aligned, then it is usually assumed (for example, in channel routing) that all the lines originate at a specific y-location and terminate at a specific y-location. An instance of such a set of lines is shown in Figure 4.39(a). This type of diagram is sometimes called a *matching diagram.*

Permutation graphs are frequently used in routing and can be defined by matching diagram. We define a *permutation graph* $G_P = (V, E_P)$, where the

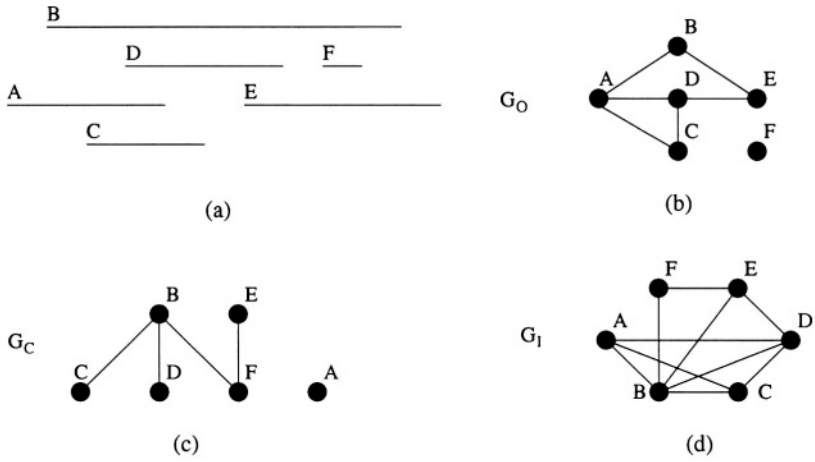


Figure 4.38: Various graphs associated with a set of intervals.

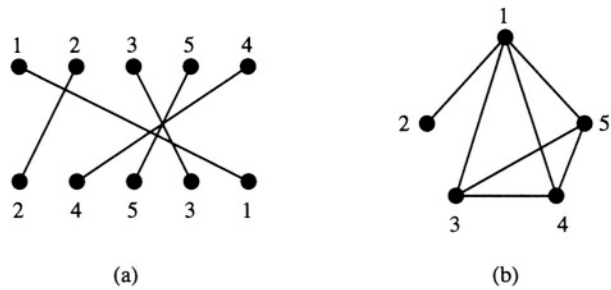


Figure 4.39: Matching diagram and permutation graph.

vertex set V is the same as defined above and E_P a set of edges defined below:

$$E_P = \{(v_i, v_j) \mid \text{ if line } i \text{ intersects line } j \}$$

An example of permutation graph for the matching diagram in Figure 4.39(a) is shown in Figure 4.39(b). It is well known that the class of containment graphs is equivalent to the class of permutation graphs [Gol80].

Two sided box defined above is called a channel. The channel routing problem, which arises rather frequently in VLSI design, uses permutation graphs to model the problem. A more general type of routing problem, called the switchbox routing problem, uses a four-sided box (see Figure 4.40(a)). The graph defined by the intersection of lines in a switchbox is equivalent to a *circle graph* shown in Figure 4.40(b). Overlap graphs are equivalent to circle graphs. Circle graphs were originally defined as the intersection graph of chords of a circle,

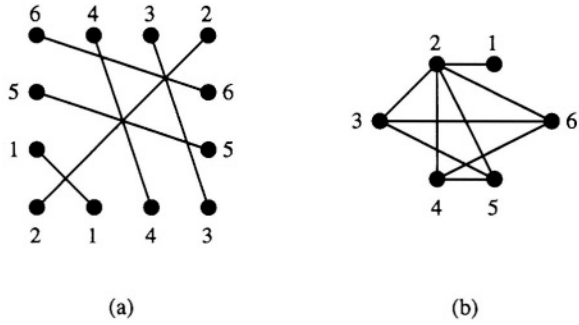


Figure 4.40: Switchbox and its circle graph.

can be recognized in polynomial time [GHS86].

4.5.1.2 Graphs Related to Set of Rectangles

As mentioned before, rectangles are used to represent circuit blocks in a layout design. Note that no two rectangles in a plane are allowed to overlap. Rectangles may share edges, i.e., two rectangles may be neighbors to each other. Given a set of rectangles $R = \{R_1, R_2, \ldots, R_m\}$ corresponding to a layout in a plane, a *neighborhood graph* is a graph $G = (V, E)$, where

$$V = \{v_i | v_i \text{ represents the rectangle } R_i \text{ and}\}$$

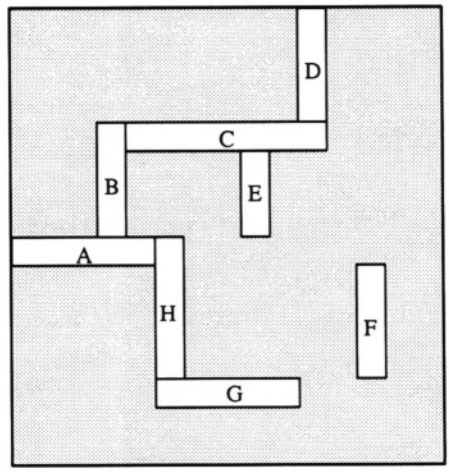
$$E = \{(v_i, v_j) |\ R_i \text{ and } R_j \text{ are neighbors}\}$$

The neighborhood graph is useful in the global routing phase of the design automation cycle where each channel is defined as a rectangle, and two channels are neighbors if they share a boundary. Figure 4.41 gives an example of a neighborhood graph, where for example, rectangles A and B are neighbors in Figure 4.41 (a), and as a result there is an edge between vertices A and B in the corresponding neighborhood graph shown in Figure 4.41 (b).

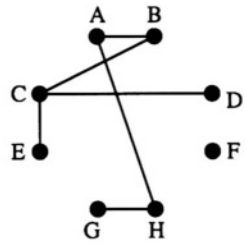
Similarly, given a graph $G = (V, E)$, a rectangular dual of the graph is a set of rectangles $R = \{R_1, R_2, \ldots, R_m\}$ where each vertex $v_i \in V$ corresponds to the rectangle $R_i \in R$ and two rectangles share an edge if their corresponding vertices are adjacent. Figure 4.42(b) shows an example of a rectangular dual of a graph shown in Figure 4.42(a). This graph is particularly important in floorplanning phase of physical design automation. It is important to note that not all graphs have a rectangular dual.

4.5.2 Relationship Between Graph Classes

The classes of graphs used in physical design are related to several well known classes of graphs, such as triangulated graphs, comparability graphs, and co-comparability graphs, which are defined below.

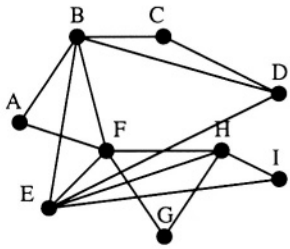


(a)

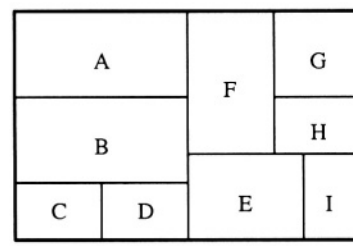


(b)

Figure 4.41: Neighborhood graphs.



(a)



(b)

Figure 4.42: Rectangular duals.

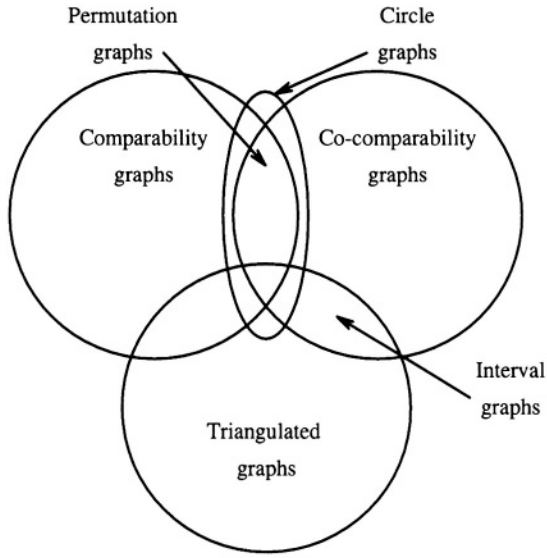


Figure 4.43: Relationship between different classes of graphs.

An interesting class of graphs based on the notion of cycle length is *triangulated graphs.* If $C = v_0, e_1, \ldots, e_{k-1}, v_k$ is a cycle in G, a *chord* of C is an edge e in $E(G)$ connecting vertices v_i and v_j such that $e \neq e_i$ for any $i = 1, \ldots, k$. A graph is *chordal* if every cycle containing at least four vertices has a chord. Chordal graphs are also known as *triangulated graphs.* A graph $G = (V, E)$ is a *comparability graph* if it is *transitively orientable.* A graph is called a *co-comparability graph* if the complement of G is transitively orientable.

Triangulated and comparability graphs can be used to characterize interval graphs. A graph G is called an *interval* graph if and only if G is triangulated and the complement of G is a comparability graph. Similarly, comparability and co-comparability graphs can be used to characterize permutation graphs. A graph G is called a *permutation graph* if and only if G is a comparability graph and the complement of G is also a comparability graph.

The classes of graphs mentioned above are not unrelated, in fact, interval graphs and permutation graphs have a non-empty intersection. Similarly the classes of permutation and bipartite graphs have a non-empty intersection. On the other hand, the class of circle graphs properly contains the class of permutation graphs. In Figure 4.43 shows the relationship between these classes.

4.5.3 Graph Problems in Physical Design

Several interesting problems related to classes of graphs discussed above arise in VLSI physical design. We will briefly state the definitions of these problems.

An extensive list of problems and related results may be found in [GJ79].

Independent Set Problem
Instance: Graph G = (V, E), positive integer $K \leq |V|$.
Question: Does G contain an independent set of size K or more, i.e., a subset $V' \subset V$ such that $|V'| \geq K$ and such that no two vertices in V' are joined by an edge in E ?
Maximum Independent Set (MIS) problem is the optimization version of the Independent Set problem. The problem is NP-complete for general graphs and remains NP-complete for many special classes of graphs [GJ79, GJS76, MS77, Pol74, YG78]. The problem is solvable in polynomial time for interval graphs, permutation graphs and circle graphs. Algorithms for maximum independent set for these classes are presented later in this chapter.

An interesting variant of the MIS problem, called the k-MIS, arises in various routing problems. The objective of the k-MIS is to select a subset of vertices, which can be partitioned into k independent sets. That is, the selected subset is k-colorable.

Clique Problem
Instance: Graph $G = (V, E)$, positive integer $K \leq |V|$.
Question: Does G contain a clique of size K or more, i.e., a subset $V' \subset V$ such that $|V'| \geq K$ and such that every two vertices in V' are joined by an edge in E ?
Maximum clique problem is the optimization version of the clique problem. The problem is NP-complete for general graphs and for many special classes of graphs. However, the problem is solvable in polynomial time for chordal graphs [Gav72] and therefore also for interval graphs, comparability graphs [EPL72], and circle graphs [Gav73], and therefore for permutation graphs.

The maximum clique problem for interval graphs arises in the channel routing problem.

Graph K-Colorability
Instance: Graph $G = (V, E)$, positive integer $K \leq |V|$.
Question: Is G K-colorable, i.e., does there exist a function $f : V \rightarrow \{ 1, 2, \ldots, K \}$ such that $f(u) \neq f(v)$ whenever $\{u, v\} \in E$?
The minimization of the above problem is more frequently used in physical design of VLSI. The minimization version asks for the minimum number of colors needed to properly color a given graph. The minimum number of colors needed to color a graph is called the *chromatic number* of the graph. The problem is NP-complete for general graphs and remain so for all fixed $K \geq 3$. It is polynomial for $K = 2$, since that is equivalent to bipartite graph recognition. It also remains NP-complete for $K = 3$ if G is the intersection graph for straight line segments in the plane [EET89]. For arbitrary K, the problem is NP-complete for circle graphs. The general problem can be solved in polynomial time for comparability graphs [EPL72], and for chordal graphs [Gav72].

As discussed earlier, many problems in physical design can be transformed into the problems discussed above. Most commonly, these problems serve as sub-problems and as a result, it is important to understand how these problems are solved. We will review the algorithms for solving these problems for several classes of graphs in the subsequent subsections.

It should be noted that most of the problems have polynomial time complexity algorithms for comparability, co-comparability, and triangulated graphs. This is due to the fact these graphs are *perfect graphs* [Gol80]. A graph $G = (V, E)$ is called perfect, if the size of the maximum clique in G is equal to the chromatic number of G and this is true for all subgraphs H of G. Perfect graphs admit polynomial time complexity algorithms for maximum clique, maximum independent set, among other problems. Note that chromatic number and maximum clique problems are equivalent for perfect graphs.

Interval graphs and permutation graphs are defined by the intersection of different classes of perfect graphs, and are therefore themselves perfect graphs. As a result, many problems which are NP-hard for general graphs are polynomial time solvable for these graphs. On the other hand, circle graphs are not perfect and generally speaking are much harder to deal with as compared to interval and permutation graphs. To see that circle graphs are not perfect, note that an odd cycle of five or more vertices is a circle graph, but it does not satisfy the definition of a perfect graph.

4.5.4 Algorithms for Interval Graphs

Among all classes of graphs defined on a set of lines, interval graphs are perhaps the most well known. It is very structured class of graphs and many algorithms which are NP-hard for general graphs are polynomial for interval graphs [Gol77]. Linear time complexity algorithms are known for recognition, maximum clique, and maximum independent set problems among others for this class of graphs [Gol80]. The maximal cliques of an interval graph can be linearly ordered such that for every vertex $v \in V$, the cliques containing v occur consecutively [GH64]. Such an ordering of maximal cliques is called a *consecutive linear ordering*. An $O(\mid V \mid + \mid E \mid)$ algorithm for interval graph recognition that produces a consecutive linear ordering of maximal cliques is presented in [BL76]. In this section, we review algorithms for finding maximum independent set and maximum clique in an interval graph.

4.5.4.1 Maximum Independent Set

An optimal algorithm for computing maximum independent set of an interval graph was developed by Gupta, Lee, and Leung [GLL82]. The algorithm they presented is greedy in nature and is described below in an informal fashion. The algorithm first sorts the $2n$ end points in ascending order of their values. It then scans this list from left to right (i.e., in ascending order of their values) until it first encounters a right endpoint. It then outputs the interval having this right endpoint as a member of a maximum independent set and deletes

```
Algorithm MAX-CLIQUE(I)
begin
    SORT-INTERVAL(I, A);
    cliq = 0;
    max_cliq = 0;
    for i = 1 to 2n do
        if A[i] = L then cliq = cliq + 1;
            if cliq > max_cliq then max_cliq = cliq;
        else cliq = cliq − 1;
    return max_cliq;
end.
```

Figure 4.44: Algorithm MAX-CLIQUE.

all intervals containing this point. This process is repeated until there is no interval left in the list. It can be easily shown that the algorithm produces a maximum independent set in a interval graph and the time complexity of the algorithm is dominated by sorting the intervals that is $O(n \log n)$. The time complexity of the algorithm is thus $O(n \log n)$, where n is the total number of intervals.

Theorem 3 *Given an interval graph, the MIS can be found in $O(n \log n)$ time, where n is the total number vertices in the graph.*

In [YG87], an optimal algorithm for finding the maximum k-colorable subgraph in an interval graph has been presented. We present an outline of that algorithm.

The set of the interval is processed from left to right in increasing order of endpoints. For a vertex v_i let I_i denote its corresponding interval, having a maximum k-colorable subgraph $G(U')$ for a set of nodes already processed. The next node v is added to U' if $G(U' \cup \{v\})$ contains no clique with more than k nodes, and is discarded otherwise.

It can be easily shown that this greedy algorithm indeed finds the optimal k-colorable independent set in an interval graph. For details, refer to [YG87].

4.5.4.2 Maximum Clique and Minimum Coloring

Since interval graphs are perfect, the cardinality of a minimum coloring is the same as that of maximum clique in interval graphs. The algorithm shown in Figure 4.44 finds a maximum clique in a given interval graph. The input to the algorithm is a set of intervals $\mathcal{I} = \{I_1, I_2, \ldots, I_n\}$ representing an interval graph. Each interval I_i is represented by its left end point l_i and right end point r_i.

In the algorithm shown in Figure 4.44, SORT-INTERVAL sorts the list of end points of all the intervals and generates an array $A[i]$ to denote whether

the endpoint at the position i in the sorted list is a left endpoint or right endpoint. $A[i]$ = L if the corresponding end point is a left endpoint. Note that the algorithm finds the size of the maximum clique in a given interval graph. However, the algorithm can easily be extended to find the maximum clique. It is easy to see that the worst case complexity of the algorithm is $O(n^2)$, where n is the total number of intervals. The time complexity of the algorithm can be reduced to O(n log n) by keeping track of the minimum of the right end points of all the intervals. The left edge algorithm (LEA) described in detailed routing (chapter 7) is a simple variation of the algorithm in Figure 4.44.

4.5.5 Algorithms for Permutation Graphs

The class of permutation graphs was introduced by Pnnueli, Lempel, and Even [PLE71]. They also showed that the class of permutation graphs is transitive and introduced an $O(n^2)$ algorithm to find the maximum clique [EPL72]. In [Gol80], Golumbic showed an O(n log n) time complexity algorithm for finding the chromatic number χ in a permutation graph.

Permutation graphs are also a structured class of graphs similar to interval graphs. Most problems, which are polynomial for permutation graphs, are also polynomial for interval graphs. In this section, we present an outline of several important algorithms related to permutation graphs.

4.5.5.1 Maximum Independent Set

The maximum independent set in a permutation graph can be found in O(n log n) time [Kim90]. As mentioned before, permutation graphs can be represented using matching diagrams as shown in Figure 4.39.

The binary insertion technique can be used on a matching diagram to find a maximum independent set in a permutation graph. Given a permutation $P = (P_1, P_2, \ldots, P_n)$ of n numbers N = (1, 2, . . . , n) corresponding to a permutation graph, note that an increasing subsequence of P represents an independent set in the permutation graph. Similarly, a decreasing subsequence of P represents a clique in the permutation graph. Therefore, to find a maximum independent set, we need to find a maximum increasing subsequence of P. It is necessary to know the the relations of the positions of numbers in the permutation. A stack is used to keep track of the relations. The algorithm works as follows:

The sequence N is scanned in increasing order. In the jth iteration, j is placed on the top of the stack i whenever j does not intersect with the front entries of the stack q, but intersects with the front entry of stack r, where $1 \leq i < j$ and $i \leq r \leq m$, and m is the total number of stacks during jth iteration. If j does not intersect with any of the front entries of the stacks 1, 2, . . . , m, then the stack m + 1 is created and j is placed on top stack m + 1. It is easy to see that the stack search and insertion can be done using binary search in O(log n) time.

Once the numbers are placed in stacks, stacks can be scanned from bottom up to get a maximum increasing subsequence. We illustrate the algorithm by

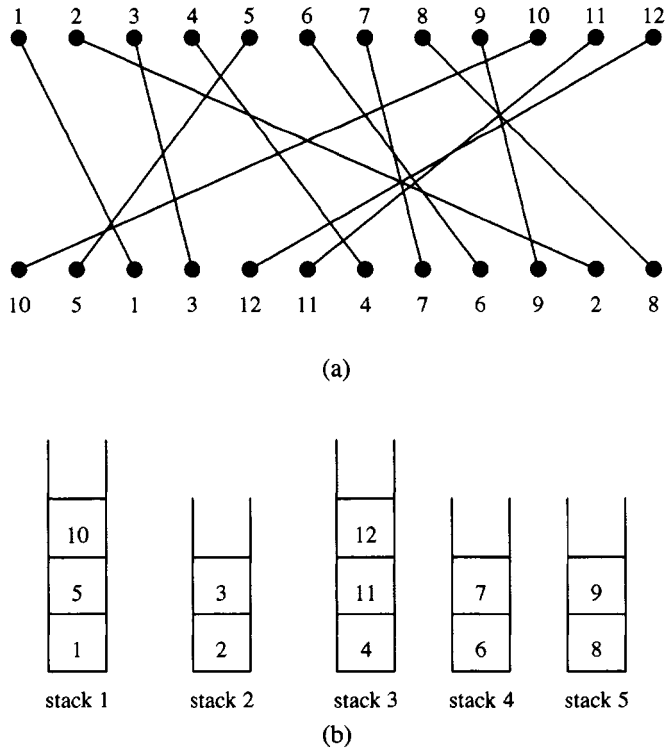


Figure 4.45: Example of Maximum independent set in permutation graph.

means of an example shown in Figure 4.45(a). Initially, the permutation P is given as (10, 5, 1, 3, 12, 11, 4, 7, 6, 9, 2, 8). The top row is processed from left to right. First 1 is placed on the stack 1. Then 2 is placed on stack 2, because 2 does not intersect with 1. After that the 3 goes on top of stack 2, since it intersects with 2 but does not intersect with 1 on top of stack 1. The 4 is placed in front of a new stack 3. Then the 5 intersects with all of the front entries of all the stacks, thus 5 is placed in front of the stack 1. In this way, all the numbers are placed in 5 stacks as shown Figure 4.45(b). Now the stacks are scanned starting from stack 5 to stack 1. One number from each stack is selected so that the numbers are in decreasing order. If 9 from stack 5, 7 from stack 4, 4 from stack 3, 3 from stack 2, and 1 from stack 1 is selected then the generated set {9, 7, 4, 3, 1} is a maximum independent set of the corresponding permutation graph. Note the total number of stacks is equal to the chromatic number of the permutation graph.

In [LSL90], Lou, Sarrafzadeh, and Lee presented a $\Theta(n \log n)$ time complexity algorithm for finding a maximum two-independent set in permutation graphs. Cong and Liu [CL91] presented an $O(n^2 \log n + nm)$ time complexity

algorithm to compute a maximum weighted k-independent set in permutation graphs where m is bounded by n^2. In fact, their algorithm is very general and applicable to any comparability graph.

4.5.5.2 Maximum k-Independent Set

The complement of a permutation graph is a permutation graph. Hence, MKIS problem in graph G is equivalent to maximum k-clique problem in $\bar{G}$. In this section, we discuss an $O(kn^2)$ time algorithm for finding the maximum k-clique in a permutation graph presented by Gavril [Gav87]. In fact, this algorithm is very general and applicable to any comparability graph.

The basic idea of the algorithm is to convert the maximum k-clique problem in a comparability graph into network flow problem. (See [Tar83] for an excellent survey of network flow algorithms.) First a transitive orientation is constructed for a comparability graph $G = (V, E)$, resulting in a directed graph $\vec{G} = (V, \vec{E})$. A directed path in $\vec{G}$ is also called as a chain. Note that each chain in $\vec{G}$ corresponds to a clique in G since G is a comparability graph. Next, each vertex in V is split into two vertices. Assume that $V = \{v_1, v_2, \ldots, v_n\}$. Then each vertex v_i corresponds to two vertices x_i and y_i in a new directed graph $\vec{G_1} = (V_1, \vec{E_1})$. There is a directed edge between x_i and y_i for all $1 \leq i \leq n$. A cost of -1 and capacity of 1 are assigned to the edge (x_i, y_i) for all $1 \leq i \leq n$. In addition, there is a directed edge between y_i and x_j if there exists a directed edge (v_i, v_j) in $\vec{G}$. A cost of 0 and capacity of 1 are assigned to the edge (y_i, x_j). Four new vertices s (source), t (sink), s' and t' are introduced as well as the directed edges (s', x_i) and (y_i, t') for all $i \in V$, (s, s'), and (t', t) are added. A cost of 0 and capacity of 1 are assigned to the edge (s', v_i) and (v_i, t'). A cost of 0 and capacity of k are assigned to the edges (s, s') and (t', t). The graph $\vec{G_1} = (V_1, \vec{E_1})$ so constructed is called a *network* where $V_1 = \{x_i, y_i | 1 \leq i \leq n\} \cup \{s, s', t', t\}$ and $E_1 = \{(x_i, y_i) | 1 \leq i \leq n\} \cup \{(y_i, x_j) | (v_i, v_j) \in \vec{E}\} \cup \{(s', x_i), (y_i, t') | 1 \leq i \leq n\} \cup \{(s, s'), (t', t)\}$.

Then the maximum k-clique problem in the graph G is equivalent to the min-cost max-flow problem in the network $\vec{G_1}$. The flow in a directed graph has to satisfy the following.

1. The flow *f(e)* associated with each edge of the graph, can be assigned a value no more than the capacity of the edge.

2. The net flow that enters a vertex is equal to the net flow that leaves the vertex.

The absolute value of flow that leaves the source, e.g. $|f(s, s')|$, is called the flow of $\vec{G_1}$. The min-cost max-flow problem in the directed graph $\vec{G_1}$ is to find the assignment of *f(e)* for each edge $e \in \vec{E_1}$ such that the flow of $\vec{G_1}$ is maximum and the total cost on the edges that the flows pass is minimum. Notice that the capacity on the directed edge (s, s') is k. Thus, the maximum flow of $\vec{G_1}$ is k. In addition, flow that passes x_i or y_i has value 1, since the capacity on the directed edge (x_i, y_i) is one for each of $1 \leq i \leq n$. Note

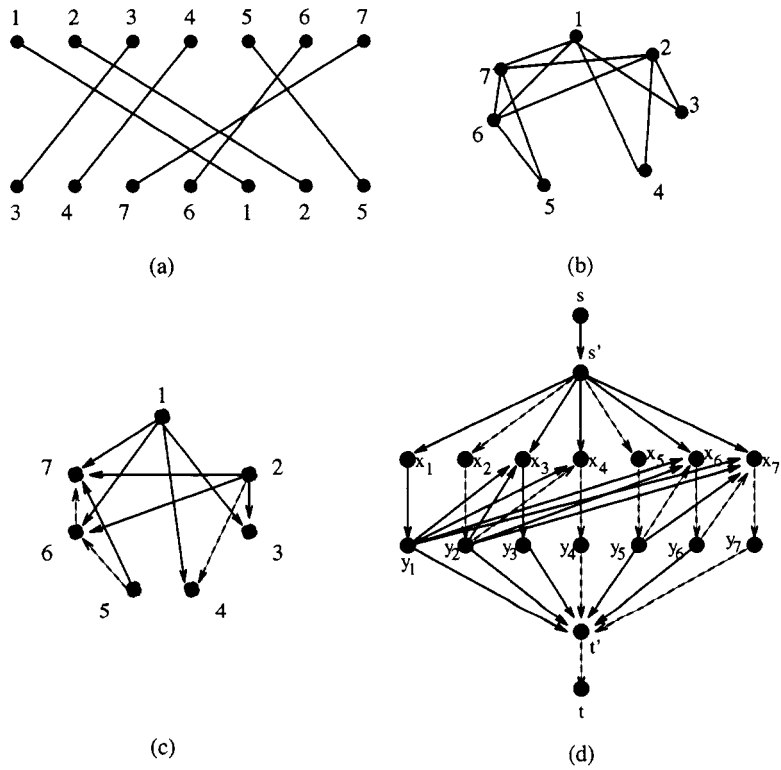


Figure 4.46: (a) Matching diagram (b) Permutation graph (c) Transitive orientation of the graph (d) Network flow graph

that a flow in $\vec{G_1}$ corresponds to a chain in $\vec{G}$. The maximum flow in $\vec{G_1}$ is k, thus the maximum number of the chains in $\vec{G}$ is k, and vice versa. The absolute value of the cost on each flow is equal to the number of vertices on the chain corresponding to the flow. Thus the minimum cost on all flows results in maximum number of vertices in the chains in $\vec{G}$, and hence maximum number of vertices in the cliques in G.

An example of a permutation graph G is given in Figure 4.46(b). The transitive orientation of G is given in Figure 4.46(c) while the network $\vec{G_1}$ is shown in Figure 4.46(d). The min-cost max-flow while $k = 2$ is highlighted in Figure 4.46(d). The chains corresponding to the min-cost max-flow are highlighted in Figure 4.46(c). The maximum 2-clique is $\{5, 6, 7\}$ and $\{2, 4\}$.

The time complexity of the algorithm is dominated by the time complexity of the algorithm to find the min-cost max-flow in a network which is $O(kn^2)$ where n is the number of vertices in the graph [Law76]. The weighted version of the MKIS problem can be solved by $O(kn^2)$ algorithm presented in [SL93].

4.5.6 Algorithms for Circle Graphs

Circle graphs are used for solving certain problems in channel routing and switchbox routing. Circle graphs are not prefect and less structured than interval and permutation graphs. Many problems, such as the maximum bipartite subgraph problem, which are polynomial for interval and permutation graphs are NP-complete for circle graphs. However, there are still many problems that can be solved in polynomial time for circle graphs which are NP-complete for general graphs. For example, polynomial time complexity algorithms are known for maximum clique and maximum independent set problems on circle graphs [Gav73], as well as for the weighted maximum clique problem [Hsu85], but the chromatic number problem for circle graphs remains NP-complete [GJMP78]. In the following, we review the circle graph algorithms used in VLSI design.

4.5.6.1 Maximum Independent Set

The problem of finding maximum independent set in a circle graph can also be solved in polynomial time. In [Sup87], Supowit presented a dynamic programming algorithm of time complexity $O(n^2)$ for finding maximum independent set in a circle graph.

Given is a set C of n chords of a circle, without loss of generality, it is assumed that no two chords share the same endpoint. Number these endpoints of the chords from 0 to 2n – 1 clockwise around the circle. Let $G = (V, E)$ denote a circle graph where $V = \{v_{ab} | a < b, \text{ab is a chord}\}$. Let G_{ij} denote the subgraph of the circle graph $G = (V, E)$, induced by the set of vertices

$$\{v_{ab} \in V : i \leq a, b \leq j\}.$$

Let $M(i, j)$ denote a maximum independent set of G_{ij}. If $i \geq j$, then G_{ij} is the empty graph and, hence $M(i, j) = \phi$. The algorithm is an application of dynamic programming. In particular, $M(i, j)$ is computed for each pair i, j; $M(i, j_1)$ is computed before $M(i, j_2)$ if $j_1 < j_2$. To compute M(i, j), let k be the unique number such that $kj \in C$ or $jk \in C$. If k is not in the range $[i, j-1]$, then $G_{ij} = G_{i,j-1}$ and hence $M(i, j) = M(i, j-1)$. If k is in the range [i, j – 1], then there are two cases to consider:

1. If $v_{kj} \in M(i, j)$, then by definition of an independent set, $M(i, j)$ contains no vertices v_{ab} such that $a \in [i, k-1]$ and $b \in [k+1, j]$.

$$\text{Therefore, } M(i, j) = M(i, k-1) \cup \{v_{kj}\} \cup M(k+1, j-1).$$

2. If $v_{kj} \notin M(i, j)$, then

$$M(i, j) = M(i, j-1).$$

Thus M(i, j) is set to the larger of the two sets M($i, j-1$) and $M(i, k-1) \cup \{v_{kj}\} \cup M(k+1, j-1)$. The algorithm is more formally stated in Figure 4.47.

```
Algorithm MIS(V)
begin
    for j = 0 to 2N − 1 do
        find k such that kj ∈ C or jk ∈ C;
        for i = 0 to j − 1 do
            if i ≤ k ≤ j-1 and |M(i, k − 1)| + 1+
                    |M(k + 1, j − 1)| > |M(i, j − 1)| then
                M(i, j) = M(i, k − 1) ∪ {v_kj}∪
                            M(k + 1, j − 1);
            else M(i, j) = M(i, j − 1)
end.
```

Figure 4.47: Algorithm MIS.

Theorem 4 *The algorithm MIS finds a maximum independent set in a circle graph in time* $O(n^2)$.

4.5.6.2 Maximum k-Independent Set

In general a *k-independent set* can be defined as a set consisting of k disjoint independent sets, and a *maximum k-independent set* (k-MIS) has the maximum number of vertices among all such k-independent sets. Although, a MIS in circle graphs can be found in polynomial time, the problem of finding a k-MIS is NP-complete even for $k = 2$ [SL89a]. Since the problem has many important applications in routing and via minimization described in the later chapters, it is required to develop some provably good approximation algorithm for this problem.

In [CHS93], Cong, Hossain, and Sherwani present an approximation algorithm for a maximum k-independent set in the context of planar routing problem in an arbitrary routing region. The problem is equivalent to finding a maximum k-independent set in a circle graph. The approximation algorithm for $k = 2$, was first presented by Holmes, Sherwani and Sarrafzadeh [HSS93] and later extended to the case of $k = 4$ in [HSS91]. In this section, we present the approximation result in the context of a circle graph.

Given a graph $G_1 = (V_1, E_1)$, the algorithm finds k independent sets one after another denoted by $S_1, S_2, \ldots, S_k$, such that S_1 is a maximum independent set in G_1, and S_i is a maximum independent set in G_i, for $2 \leq i \leq k$, where $G_i = (V_i, E_i)$ is inductively defined as:

$V_i = V_{i-1} - S_i$ and $E_i = E_{i-1} - \{(v_1, v_2) | v_1 \in S_i$ and $v_2 \notin S_i$ and $(v_1, v_2) \in E_{i-1}\}$

Clearly, the algorithm reduces the problem of k-MIS to a series computations of MIS in a circle graph. Since in circle graphs, the complexity of computing 1-MIS is $O(n^2)$, the total time complexity of this approximation algorithm is $O(kn^2)$.

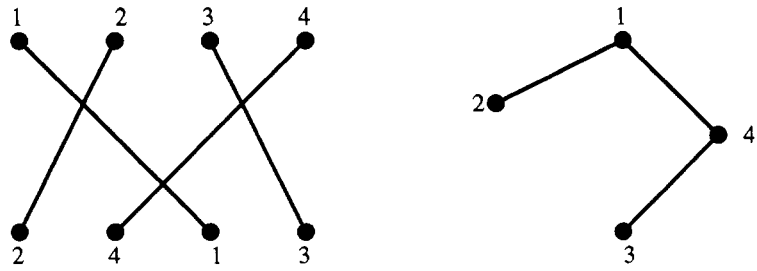


Figure 4.48: Example to show non-optimality of K-MIS algorithm.

Algorithm MKIS $(G(V, E), k)$
begin
1. $V' = V$;
2. **for** $i = 1$ to k **do**
   $S_i = \text{MIS}(V')$;
   $V' = V' - S_i$;
3. return $S_1 \cup S_2 \cup \ldots \cup S_k$;
end.

Figure 4.49: Algorithm MKIS.

Consider the circle graph shown in Figure 4.48. Clearly, the 2-MIS of the graph is {(1,3),(2,4)}. The maximum independent sets in the graph are {(1,3),(2,4),(2,3)}. In the MKIS algorithm, the MIS is chosen randomly, and a *bad* selection of a MIS ({(2, 3)} in this case) may not lead to an optimal 2-MIS for the graph. If {(2,3)} is chosen then either we can choose 1 or 4. Thus, the total number of nets chosen is three while the optimal has four nets. A similar reasoning would show that the algorithm is non-optimal for the k-MIS problem.

The algorithm is formally stated in Figure 4.49.

For any heuristic algorithm H_k for k-MIS, the *performance ratio* of H_k is defined to be $\frac{\Psi_k}{\Psi_k^*}$, where Ψ_k is the size of the k-independent set obtained by the algorithm H_k and Ψ_k^* is the k-MIS in the same graph. The lower bound on the performance ratio is established based on the following theorem.

Theorem 5 *Let γ_k be the performance ratio of the algorithm MKIS for k-MIS. Then,*

$$\gamma_k \geq 1 - (1 - \frac{1}{k})^k$$

Corollary 1 *Given a circle graph G, MKIS can be used to approximate a maximum bipartite set of G with a performance bound of at least* 0.75.

It is easy to see that the function $f(x) = 1 - (1 - \frac{1}{x})^x$ is a decreasing function. Moreover,

$$\lim_{x \to \infty} [1 - (1 - \frac{1}{x})^x] = 1 - e^{-1}$$

where $e \approx 2.718$. Therefore, we have

Corollary 2 *For any integer k, the performance ratio of the algorithm MKIS for k-MIS is at least*

$$\gamma_k \geq 1 - e^{-1} \approx 63.2\%$$

Although the approximation result presented above is for circle graphs, this equally applicable to any class of graphs where the problem of finding MIS is polynomial time solvable.

Another variation of the MIS problem in circle graphs is called *k-density MIS*. Given a set of intervals, the objective of *k*-density MIS is to find an independent set of intervals with respect to overlap property such that the interval graph corresponding to that set has a clique of size at most *k*.

4.5.6.3 Maximum Clique

Given a circle graph $G = (V, E)$, it is easy to show that for every vertex $v \in V$, the induced subgraph $G_v = (V_v, E_v)$ is a permutation graph, where,

$$V_v = \{v\} \cup \{Adj(v)\}$$

$$E_v = \{(u, v) | u \in V_v\}$$

For each G_v, maximum clique can be found using the algorithm presented for maximum clique in a permutation graph. Let C_v be the maximum clique in G_v, then the maximum clique in G is given by $\max\{C_v\}$, for all $v \in V$. It is easy to see that the time complexity of this algorithm is $O(n^2 \log n)$.

4.6 Summary

A VLSI layout is represented as a collection of tiles on several layers. A circuit may consists of millions of such tiles. Since layout editors are interactive, their underlying data structures should enable editing operations to be performed within an acceptable response time. Several data structures have been proposed for layout systems. The most popular data structure among these is corner stitch. However, none of the data structures is equally good for all the operations. The main limitation of all the existing data structures is that they only work on rectangular objects. In other words, the data structures do not support any other shaped objects such as circular, L-shaped. Therefore, development of new data structure is needed to handle different shaped objects.

Also, as parallel computation becomes practical, new data structures need to be developed to adapt to the parallel computation environment.

Due to sheer size of VLSI circuits, low time complexity is necessary for algorithms to be practical. In addition, due to NP-hardness of many problems, heuristic and approximation algorithms play a very important role in physical design automation.

Several special graphs are used to represent VLSI layouts. The study of algorithms of these graphs is essential to development of efficient algorithms for various phases in VLSI physical design cycle.

4.7 Exercises

1. Design an algorithm to insert a block in a given area using the modified linked list data structure. Note that you need to use area searching operation to insert a block and a modified linked list to keep track of the vacant tiles.

2. Design an algorithm to delete a given block from a given set of blocks using modified linked list data structure. Note that once a block is deleted the area occupied by that block becomes vacant tile and the linked list must be updated to take care of this situation.

3. Design algorithms using a linked list data structure to perform the plowing and compaction operations.

4. Solve the problem 3 using a modified linked list.

5. The problem of connectivity extraction is very important in circuit extraction phase of physical design. It is defined as follows. Given a set of blocks $B = \{B_1, B_2, \ldots, B_n\}$ in an area, let us assume that there is a type associated to each circuit. For example, all the blocks can of one type and all the vacant tiles could be of another type. Two blocks B_i and B_j are called connected if there is a sequence of k $(k \leq n)$ distinct blocks of the same type $B_{\pi(1)}, B_{\pi(2)}, \ldots, B_{\pi(k)}$, such that $\pi(i) \leq n$, $B_{\pi(1)} = B_i$, $B_{\pi(k)} = B_j$, and $B_{\pi(1)}$ is a neighbor of $B_{\pi(2)}$, $B_{\pi(2)}$ is a neighbor of $B_{\pi(3)}$, and so on and finally $B_{\pi(k-1)}$ is a neighbor of $B_{\pi(k)}$.

   (a) Design an algorithm using a modified linked list data structure to extract the connectivity of two blocks.

   (b) Design an algorithm using a corner stitch data structure to find the connectivity of two blocks.

†6. The existing data structure can be modified to handle layouts in a multilayer environment. Consider the following data items associated to a tiles in a multilayer environment:

record Tile =

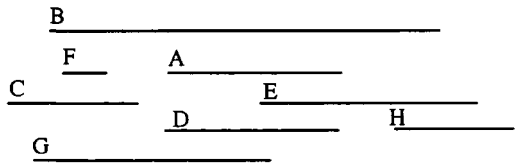


Figure 4.50: A set of intervals.

coordinate;
height;
width;
type;
text;
layer;
end record

(a) Design an algorithm to move the entire layout in one direction.

(b) Following problem 5, find the connectivity of any two blocks in a multilayer environment using corner stitch data structure.

7. Modify algorithm NEIGHBOR-FIND1 to find all the neighbors of a given block using a linked list data structure.

8. Modify algorithm NEIGHBOR-FIND2 to find all the neighbors of a given block using bin-based data structure.

†9. Assume a layout system that allows 45° segments, i.e., the blocks could be 45° angled parallelogram as well as rectangular. Modify the corner stitch data structure to handle this layout system. Are four pointers still sufficient in this situation ?

10. Given a family of sets of segments $\mathcal{S} = \{S_1, S_2, \ldots, S_n\}$, where S_i is the set of segments belonging to net N_i on a layer.

    Determine if there is a connectivity violation by developing an algorithm which finds all such violations.

11. For the set of intervals shown in Figure 4.50, find maximum independent set, maximum clique, and maximum bipartite subgraph in the interval graph defined by the intervals.

12. For the matching diagram shown in Figure 4.51, find its permutation graph. Find maximum independent set, minimum number of colors required to color it, and maximum bipartite subgraph in this permutation graph.

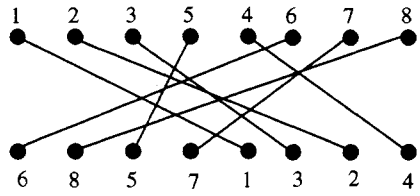


Figure 4.51: Channel problem.

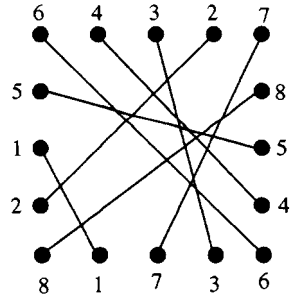


Figure 4.52: Switchbox problem.

13. For the switchbox shown in Figure 4.52, find maximum independent set, maximum clique, and maximum bipartite subgraph in the permutation graph defined by the matching diagram.

†14. Prove that the algorithm MAX-CLIQUE correctly finds the size of the maximum clique in an interval graph.

†15. Improve the time complexity of the algorithm MAX-CLIQUE to $O(n \log n)$. The algorithm should also be able to report a maximum clique.

†16. Prove that the algorithm MIS for finding a maximum independent set in circle graphs does indeed find the optimal solution.

17. Develop a heuristic algorithm for finding a maximum bipartite subgraph in circle graphs.

†18. Implement the approximation algorithm for finding a k-independent set in circle graphs. Experimentally evaluate the performance of the algorithm by implementing an exponential time complexity algorithm for finding a k-independent set.

19. Develop an efficient algorithm to find a k-density MIS in circle graphs.

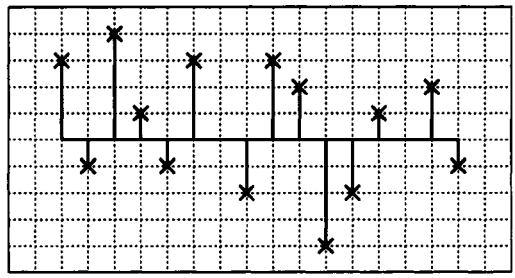


Figure 4.53: A single trunk Steiner tree.

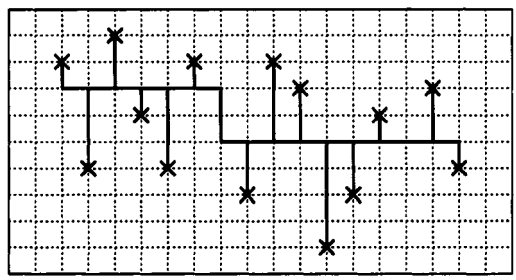


Figure 4.54: A two-trunk Steiner tree.

20. Steiner trees play a key role in global and detail routing problems. Consider the following *Single Trunk Steiner Tree* problem. A single trunk Steiner tree consists of a single horizontal line segment and all the points are joined by short vertical line segments. An example of a single trunk Steiner tree is shown in Figure 4.53.

    Given a set of n points in a plane, develop an $O(n)$ algorithm for the minimum cost single trunk Steiner tree.

†21. Prove that for n = 3, single trunk Steiner tree is indeed an optimal rectilinear Steiner tree.

22. For $n = 4$, give an example which shows that single trunk Steiner tree is not an optimal rectilinear Steiner tree.

23. Single trunk Steiner tree can be easily generalized to k-trunk Steiner tree problem, which consists of k non-overlapping horizontal trunks. An example of a two trunk Steiner tree is shown in Figure 4.54.

    Develop an efficient algorithm for 2-trunk Steiner tree problem.

†24. Does there exist an $O(n^{ck})$ algorithm for the k-trunk Steiner tree problem, for a small constant c?

†25. Implement Hadlock's Algorithm for finding max-cut in a planar graph.

†26. Prove that Hadlock's algorithm is optimal by showing it deletes minimum number of edges.

27. Given a set of rectangles in a plane, develop an efficient algorithm to detect if any two rectangles intersect or contain each other.

†28. Given a switch box, develop an efficient algorithm to find the minimum diameter of rectilinear Steiner trees. The diameter of a tree is the maximum distance between any two of its vertices.

Bibliographic Notes

The paper by John Ousterhout on the corner stitch data structure [Ous84] gives details of different algorithms used to manipulate a layout. The corner stitch data structure has been extended in various ways to account for nonrectilinear shapes and interaction of objects in different layers. In [Meh94] D. P. Mehta presented a technique for estimating the storage requirements of the Rectangular Corner Stitching data structure and the L-shaped Corner Stitching Data Structure on a given circuit by studying the circuit's geometric properties.

However, there are no efficient data structures to express the true three dimensional nature of a VLSI layout. The details of CIF can be found in Mead & Conway [MC79].

Cormen, Leiserson and Rivest [CLR90], present an in depth analysis of graph algorithms. Tarjan [Tar83] provides excellent reference for graph matchings, minimum spanning trees, and network flow algorithms. Computational geometry algorithms are discussed in detail by Preparata and Shamos [PS85]. The theory of NP-completeness is discussed in great detail in Garey and Johnson [GJ79].

General graph concepts have been described in detail in [CL86]. Algorithms and concepts for the perfect graphs, interval graphs, permutation graphs, and circle graphs can be found in [Gol80].