IMPLEMENTATION OF VARIOUS ROUTING AND MAZE ALGORITHMS FOR VLSI ARCHITECTURE

A dissertation submitted
in partial fulfilment of the requirements for the award of the degree of

Master of Engineering
In
Electronics and Communication Engineering

By

Indrajit Sarkar

Roll No. 8727

Department of Electronics and Communication Engineering,
Delhi College of Engineering, University of Delhi
Session 2004-2006

ACKNOWLEDGEMENT

I wish to acknowledge our sincere thanks to my guide **Prof. Asok Bhattacharyya**, H.O.D. (Department of Electronics and Communication Engineering) for his suggestions, excellent guidance and timely advice which have made my thesis work success. I am indebted to the entire faculty and non teaching staff of Electronics and communication department, who had very helpful and cooperative to me at all times.

Indrajit Sarkar (8727)

Department of Electronics and Communication Engineering
Delhi College of Engineering, University of Delhi
Session 2004-2006

DEPARTMENT OF ELECTRONICSANDCOMMUNICATION
DELHI COLLEGE OF ENGINEERING
UNIVERSITY OF DELHI
DELHI

CERTIFICATE

Certified that the thesis work entitled

IMPLEMENTATION OF VARIOUS ROUTING AND MAZE ALGORITHMS FOR VLSI ARCHITECTURE

is bonafied work carried by

Indrajit Sarkar (8727)

In partial fulfilment for the award of degree of Master of Engineering in Electronics and Communication Engineering of the University of Delhi during the year 2004-2006.It is certified that all corrections/suggestions indicated for internal assessment have been in corporate in the report deposited in the Departmental library. The project report has been approved as it satisfied the academic requirements in respect of thesis work prescribed for the Master of Engineering Degree.

Signature of Guide

Prof. Asok Bhattacharyya

Abstract

Field Programmable Gate Array (FPGAs) have emerged as an attractive means of implementing logic circuits providing instant manufacturing turnaround and negligible prototype costs. They hold the promise of replacing much of the VLSI market now held by Mask Programmed Gate Arrays. An important phase in FPGA production is the routing of the cells present inside the arrays for which several CAD tools have been developed.

My project is another attempt to implement these routing algorithms which deals with Global and Detailed Routing problems. Two Maze Routing Algorithms-Lee's and Soukup's Algorithms and also multilayer maze routing algorithm is studied and implemented. These algorithms are a part of Global Routing phase in the design cycle of an FPGA. In other routing algorithm the spanning tree algorithm is also covered. In Detailed Routing, the channel routing problem, the segmented Channel Routing method has been covered, which in turn consists of Mask Programmed, Fully Segmented, One Segment and Two Segment Routing.

Contents

List of Figures

Chapter 1

Introduction

From the beginning of its era in the 1960's, Integrated Circuit(IC) fabrication technology has evolved from being able to integrate a few transistors in Small Scale Integration (SSI) to today's integration of well over a million transistors in Very Large Scale Integration (VLSI). This rapid growth in integration technology has been made possible by the automation of the various steps involved in design and fabrication of VLSI chips.
Integrated circuits consist of a number of electronic components, built by layering several different materials in a well defined fashion on a silicon base called a wafer. The designer of an IC transforms a circuit description into a geometric description, which is known as layout. A layout consists of a set of planar geometric shapes in several layers. The layout is then checked to ensure that it meets all the design requirements. The result is a set of design files in a particular unambiguous representation, known as intermediate form that describes the layout. The design files are then into pattern generator files, which are used to produce patterns called masks by an optical pattern generator. During fabrication, these masks are used to pattern a silicon wafer using a sequence of photolithographic steps. The component formation requires very exacting details about geometric patterns and separation between them. The process of converting the specifications of an electrical circuit into a layout is called physical design.

1.1 VLSI Physical Design Automation

VLSI Physical Design Automation is essentially the study of algorithms and data structures related to the physical design process. The objective is to study optimal arrangements of devices on a plane (or in a three dimensional space) and efficient interconnection schemes between these devices to obtain the desired functionality. Since space on a wafer is very expensive real-estate, algorithms must use the space very efficiently to lower costs and improve yield. In addition, the arrangement of devices plays a key role in determining the performance of a chip. Algorithms for physical design must also ensure that all the rules required by the fabrication are observed and that the layout is within the tolerance limits of the fabrication process. Finally, algorithms must be efficient and should be able to handle very large designs. Efficient algorithms not only lead to fast turn-around time, but also permit designers to iteratively improve the layouts. Since many of the objects in VLSI physical design are very simple geometric objects, such as rectangles and lines, physical design algorithms tend to be very intuitive in nature and significantly overlap with graph theory and combinatorics. As a result, physical design automation is considered by many as the study of graph theoretic and combinatorial algorithms for manipulation of geometric objects in two and three dimensional space.

1.2 VLSI Design Cycle

Our emphasis is on the physical design step of the VLSI design cycle. However to gain a global perspective, we briefly outline all the steps of the VLSI design cycle

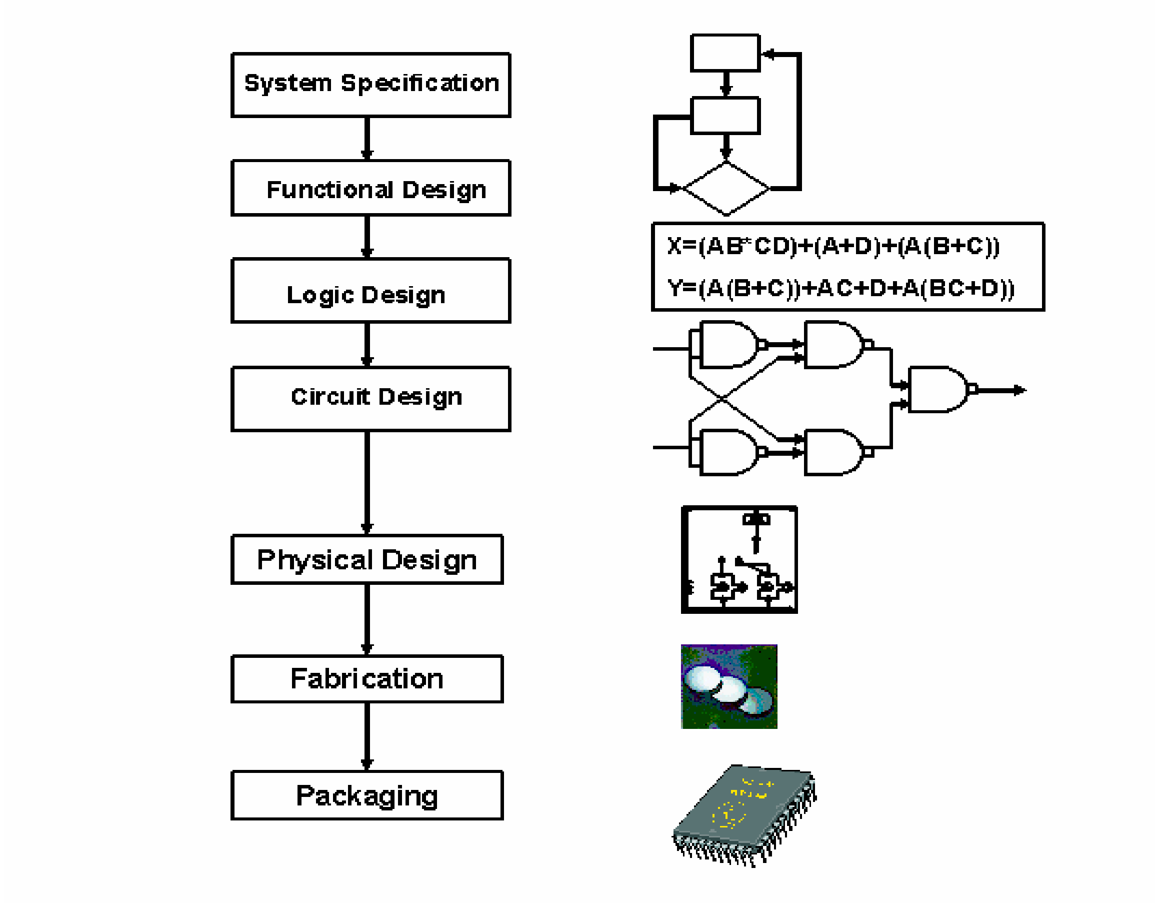


Figure 1.1: Design Process Steps

- **System specification:** As in any design process the first step is to lay down the specifications of the system to be designed. This necessitates creating a high level representation of the system. The factors to be considered in this process include: performance, functionality, and the physical dimensions. The choice of fabrication technology and design techniques is also considered. The end results are specifications for the size, speed, power, and functionality of the VLSI system to be designed.

- **Functional Design:** In this step, behavioral aspects of the system are considered. The outcome is usually a timing diagram or other relationships between sub-units. This information is used to improve the overall design process and to reduce the complexity of the subsequent phases.

- **Logic Design:** In this step, the logic structure that represents the functional design is derived and tested. The achieved design is represented by a textual, schematic or graphic description. Commonly, the logic design is represented by boolean expressions. These expressions are minimized to achieve the smallest logic design which conforms to the functional design. This logic design of the system is simulated and tested to verify its correctness.

- **Circuit Design:** The purpose of circuit design is to develop a circuit representation based on the logic design. The boolean expressions are converted into a circuit representation by taking into consideration the speed and power requirements of the original design. The electrical behaviors of the various components are also considered in this phase. The circuit design is usually expressed in a detailed circuit diagram.

- **Physical Design:** In this step, the circuit representation of each component is converted into a geometric representation. This representation is in fact a set of geometric patterns which perform the intended logic function of the corresponding component. Connections between different components are also expressed as geometric patterns. As stated earlier, this geometric representation of a circuit is called a layout. The exact details of the layout also depend on design rules, which are guidelines based on the limitations of the fabrication process and the electrical properties of the fabrication materials. Physical design is a very complex process; therefore, it is usually broken down into various sub-steps in order to handle the complexity of the problem. In fact, physical design is arguably the most time consuming step in the VLSI design cycle.

- **Design Verification:** The layout is verified in this step to ensure that the layout meets the system specifications and the fabrication requirements. Design verification consists of Design Rule Checking and Circuit Extraction. Design Rule Checking (DRC) is a process which verifies that all geometric patterns meet the design rules imposed by the fabrication process. After checking the layout for design rule violations and removing the design rule violations, the functionality of the layout is verified by circuit extraction. This is a reverse engineering process and generates the circuit representation from the layout. This reverse engineered circuit representation can then be compared with the original circuit representation to verify the correctness of the layout.

- **Fabrication:** After verification, the layout is ready for fabrication. Fabrication process consists of several steps: preparation of wafer, deposition, and diffusion of various materials on the wafer according to layout description.

- **Packaging, Testing and Debugging:** Finally, the wafer is fabricated and diced in a fabrication facility. Each chip is then packaged and tested to ensure that it meets all the design specifications and that it functions properly. Chips used in Printed Circuit Boards(PCBs) are packaged in Dual In-line Package(DIP) or Pin Grid Array(PGA). Chips which are to be used in a Multi-Chip Module(MCM) are not packaged, since MCMs use bare or naked chips.

The VLSI design cycle involves iterations, both within a step and between different steps. The entire design cycle may be viewed as transformations of representation in various steps. In each step, a new representation of the system is created and analyzed. The representation is iteratively improved to meet system specifications. For example, a layout is iteratively improved so that it meets the timing specifications of the system. Another example may be detection of design rule violations during design verification. If such violations are detected, the physical design step needs to be repeated to correct the error.

1.3 Physical Design Cycle

The input to the physical design cycle is a circuit diagram and the output is the layout of the circuit. This is accomplished in several stages such as:

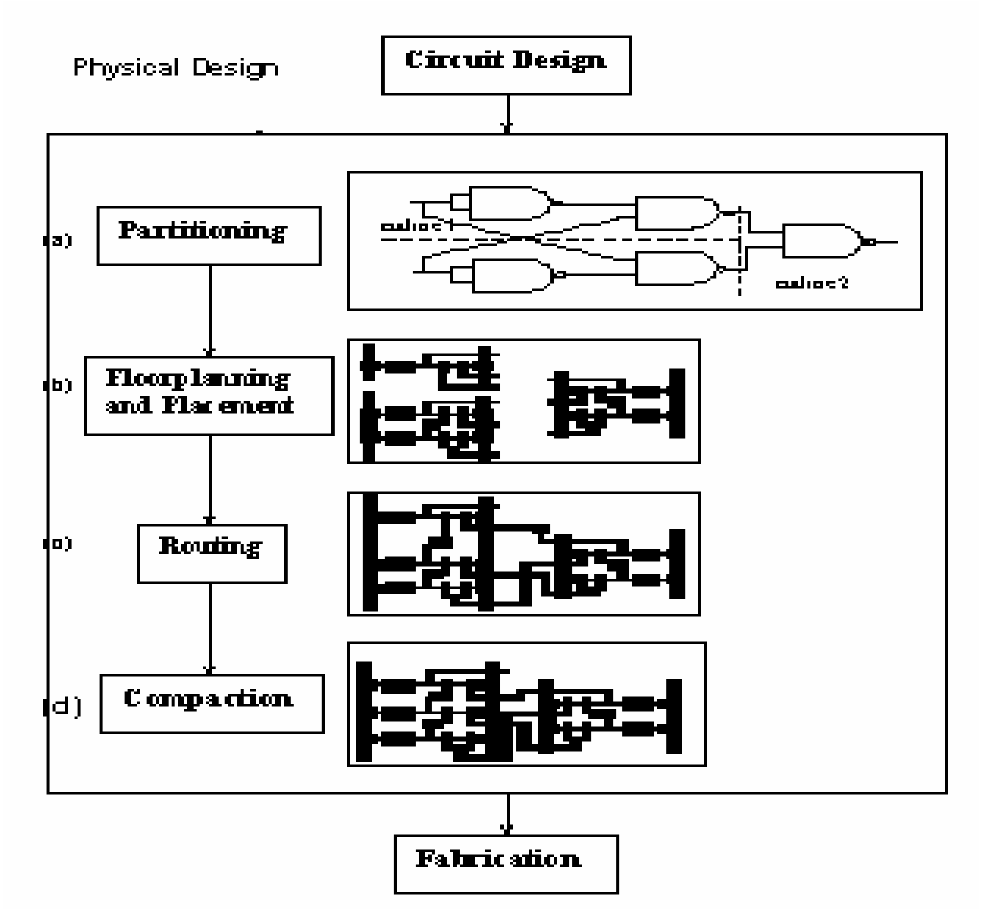


Figure 1.2: Physical Design Cycle

- **Partitioning:** A chip may contain several million transistors. Layout of the entire circuit cannot be handled due to the limitation of memory space as well as computation power available. Therefore, it is normally partitioned by grouping the components into blocks (sub-circuits/modules). The actual partitioning process considers many factors such as, the size of the blocks, number of blocks, and number of interconnections between the blocks. The output of partitioning is a set of blocks along with the interconnections required between blocks. The set of interconnections required is referred to as a netlist. Figure 1.2 (a) shows that the input circuit has been partitioned into three blocks.

- **Floorplanning and Placement:** This step is concerned with selecting good layout alternatives for each block, as well as the entire chip. The area of each chip can be calculated after partitioning and is based approximately on the number and the type of components in that block. The actual rectangular shape of the block, which is determined by the aspect ratio may, however, be varied within a pre-specified range. Floorplanning is a critical step, as it sets up the ground work for a good layout. However, it is computationally quite hard. Very often the task of Floorplan layout is done by a design engineer, rather than a CAD tool.

  During placement, the blocks are exactly positioned on the chip. The goal of placement is to find a minimum area arrangement for the blocks that allows completion of interconnections between the blocks. Placement is typically done in two phases. In the first phase an initial placement is created. In the second phase, the initial placement is evaluated and iterative improvements are made until the layout has minimum area and conforms to design specifications. Figure 1.2 (b) shows that three blocks have been placed. It should be noted that some space between the blocks is intentionally left empty to allow interconnections between blocks.

  The quality of placement will not be evident until the routing phase has been completed. A good routing and circuit performance heavily depend on a good placement algorithm. This is due to the fact that once the position of each block is fixed, very little can be done to improve the routing and the overall circuit performance.

- **Routing:** The objective of the routing phase is to complete the interconnections between blocks according to the specified netlist. First, the space not occupied by the blocks (called the routing space) is partitioned into rectangular regions called channels and switchboxes. The goal of a router is to complete all circuit connections using the shortest possible wire length and using only the channel and switch boxes. This is usually done in two phases, referred to as the Global Routing and Detailed Routing phases. In global routing, connections are completed between the proper blocks of the circuit disregarding the exact geometric details of each wire and pin. For each wire, the global router finds a list of channels which are to be used as a passageway for that wire. In other words, global routing specifies the 'loose route' of a wire through different regions in the routing space. Global routing is followed by detailed routing which completes point-to-point connections between pins on the blocks. Loose routing is converted into exact routing by specifying geometric information such as width of wires and their layer assignments. Detailed routing includes channel routing and switchbox routing. Due to very nature of the routing algorithms, complete routing of all the connections cannot be guaranteed in many cases. As a result, a technique called rip-up and re-route is used, which basically removes troublesome connections and reroutes them in a different order. The routing phase of Figure1.2 (c) shows that all the interconnection between three blocks has been routed.

- **Compaction:** Compaction is simply the task of compressing the layout in all directions such that the total area is reduced. By making the chip smaller, wire lengths are reduced which in turn reduces the signal delay between components of the circuit. At the same time, a smaller area may imply more chips can be produced on a wafer which in turn reduces the cost of manufacturing. The final diagram in Figure 1.2 (d) shows the compacted layout.

Physical design, like VLSI design, is iterative in nature and many steps such as global routing and channel routing are repeated several times to obtain a better layout.

Chapter 2

2.1 Gate Arrays

This design style is a simplification of standard cell design. Unlike standard cell design, all the cells in gate array are identical. In gate design, the entire is prefabricated with an array of identical gates or cells. These cells are separated by both vertical and horizontal spaces called vertical and horizontal channels. The circuit design is modified such that it can be partitioned into a number of identical blocks. Each block must be logically equivalent to a cell on the gate array. The name 'gate array' signifies the fact that each cell may simply be a gate, such as a 3 input NAND gate. Each block in design is mapped or placed onto a prefabricated cell on the wafer, during the partitioning/placement phase, which is reduced to a block to cell assignment problem.

The number of partitioned blocks must be less than or equal to that the total number of cells on the wafer. Once the circuit is partitioned into identical blocks, the task is to make the interconnections between the prefabricated cells on the wafer using horizontal and vertical channels to form the actual circuit. The uncommitted gate array is taken into fabrication facility and routing layers are fabricated on top of the wafer. The completed wafer is also called 'customized wafer'. It should be noted that, the number of tracks allowed for routing in each channel is fixed. As a result, the purpose of routing phase is simply to complete the connections rather than minimize the area. Two layers of interconnections are most common; though one and three layers are also used.

This simplicity of gate array design is gained at the cost of rigidity imposed upon the circuit both by the technology and the prefabricated wafers. The advantage of gate arrays is that the steps involved for creating any prefabricated wafer are the same and only the last few steps in the fabrication process actually depend on the application for which the design will be used.

Hence gate arrays are cheaper and easier to produce than full-custom or standard cell. Similar to standard cell design, gate array architecture is the most restricted form of layout. This also means that it is the simplest for algorithms to work with. For example, the task of routing in gate array is to determine if a given placement is routable.

The routability problem is conceptually simpler as compared to the routing problem in standard cell and full-custom design styles.

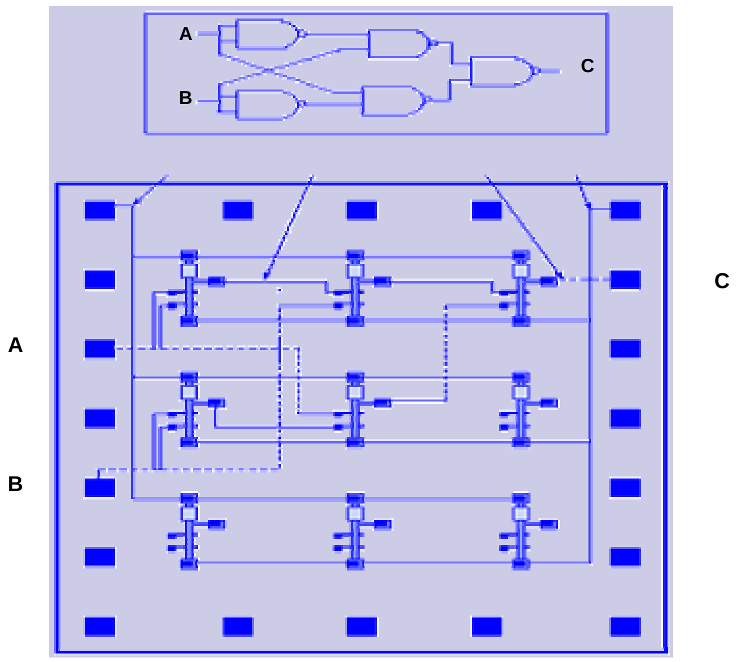


Figure 2.1: Gate Array

2.2 Field Programmable Gate Arrays

The Field Programmable Gate Array (FPGA) is a new approach to ASIC design that can dramatically reduce manufacturing turn-around time and cost. FPGA designs provide large scale integration and user programmability. A FPGA consists of horizontal rows of programmable logic blocks which can be interconnected by a programmable routing network. While, the typical FPGA logic block is more complex than a gate, it is much simpler than a cell in the standard cell designs. In its simplistic form, a logic block is simply a memory block which can be programmed to remember the logic table of a function. Given a certain input, the logic block 'looks up' the corresponding output from the logic table and sets its output line accordingly. Thus by loading different look-up tables, a logic block can be programmed to perform different functions. It is clear that 2^k bits are required in a logic block to represent a k-bit input, 1-bit output combinational logic function. Obviously, logic blocks are only feasible for small values of k. Typically, the value of k is 5 or 6. For multiple outputs and sequential circuits the value of k is even less. The rows of logic blocks are separated by horizontal routing channels. The channels are not simply empty areas in which metal lines can be arranged for a specific design. Rather, they contain predefined wiring ‘segments’ of fixed lengths. Each input and output of a logic block is connected to a dedicated vertical segment. Other vertical segments merely pass through the blocks, serving as feedthroughs between channels. Connection between horizontal segments is provided through anti fuses whereas the connection between a horizontal segment and a vertical segment is provided through a cross fuse.

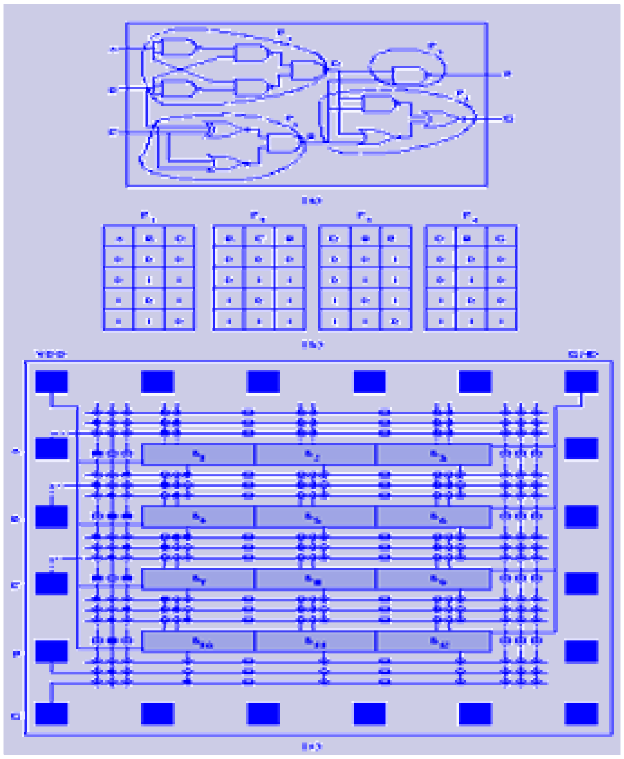


Figure 2.2: A committed FPGA

Since there are no user specific fabrication steps in a FPGA, fabrication process can be set up in a cost effective manner to produce large quantities of generic (un-programmed) FPGAs. The customization (programming) of a FPGA is rather simple. Given a circuit, it is decomposed into smaller sub circuits, such that each sub circuit can be mapped to a logic block. The interconnections between any two sub circuits are achieved by

programming the FPGA interconnects between their corresponding logic blocks. Programming (blowing) one of the fuses (antifuse or crossfuse) provides a low resistance bi-directional connection between two segments. When blown, antifuses connect the two segments to form a longer one. In order to program a fuse, a high voltage is applied across it. FPGAs have special circuitry to program the fuses. The circuitry consists of the wiring segments and control logic at the periphery of the chip. Fuse addresses are shifted into the fuse programming circuitry serially.
The programmable nature of these FPGAs requires CAD algorithms to make effective use of logic and routing resources. The problems involved in customization of a FPGA are somewhat different from those of other design styles; however, many steps are common. For example, the partition problem of FPGAs is different than partitioning problem in all design style while the placement and the routing are similar to gate array approach.

2.3 Sea of Gates

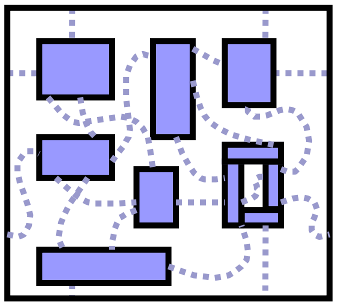
The sea of gates is an improved gate array in which the master is filled completely with transistors. The master of the sea-of-gates has a much higher density of logic implemented on the chip and allow a designer to fabricate complex circuits such as RAMs to be built. In the absence of routing channels, interconnects have to be completed either by routing through gates, or by adding more metal or polysilicon interconnection layers. There are problems associated with either solution. The former reduces the gate utilization; the latter increases the mask count and increases fabrication time and cost.

Chapter 3

3.1 Global Routing

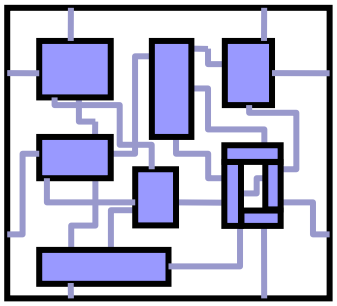
In the placement phase, the exact locations of circuit blocks and pins are determined. A netlist is also generated which specifies the required interconnections. Space not occupied by the blocks can be viewed as a collection of regions. These regions are used for routing and are called as routing regions. The process of finding the geometric layouts of all the nets is called routing. Each routing region has a capacity, which is the maximum number of nets that can pass through that region. The capacity of a region is a function of the design rules and dimensions of the routing regions and wires. Nets must be routed within the routing regions and must not violate the capacity of any routing region. In addition, nets must not short-circuit, that is, nets must not intersect each other. The objective of routing is dependent on the nature of the chip. For general purpose chips, it is sufficient to minimize the total wire length. For high performance chips, total wire length may not be a major concern. Instead, we may want to minimize the longest wire to minimize the delay in the wire and therefore maximize its performance. Usually routing involves special treatment of such nets as clock nets, power and ground nets. In fact, these nets are routed separately by special routers.

There are two types of routing regions: channels and switchboxes. A switchbox is a rectangular area bounded on all sides. A channel is a rectangular area bounded by two opposite sides by the blocks. Capacity of a channel is a function of the number of layers(l), height(h) of the channel, wire width(w) and wire separation(s),i.e., Capacity = (l×h)/(w+s). A VLSI chip may contain several million transistors. As a result, tens of thousands of nets have to be routed to complete the layout. In addition, there may be several hundreds of possible routes for each net. This makes the routing problem computationally hard. In fact, even when the routing problem is restricted to channels, if cannot be solved in polynomial time, i.e., the channel routing problem is NP-complete. Therefore, routing has traditionally been divided into two phases. The first phase is called global routing and generates a ‘loose’ route for each net. In fact it assigns a list of routing regions to each net without specifying the actual geometric layout of wires as shown in the Figure: 3.1(a). The second phase, which is called detailed routing, finds the actual geometric layout of each net within the assigned routing regions as shown in the Figure: 3.1(b). Unlike global routing, which considers the entire layout, a detailed router considers just one region at a time. The exact layout is produced for each wire segment assigned to a region, and vias are inserted to complete the layout. Detailed routing includes channel routing and switchbox routing. Another approach to routing is called Area Routing, which is a single phase routing technique. However, this technique is computationally infeasible for general VLSI circuits and is typically used for specialized problems.



(a)

Global Routing



(b)

Detailed Routing

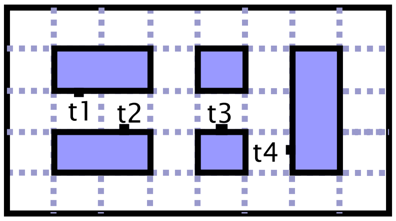
Figure 3.1: Routing

3.2 Problem Formulation

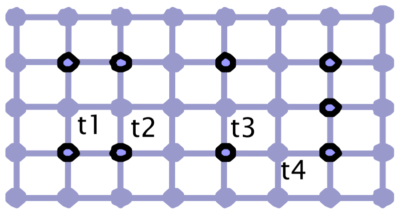
The global routing problem is typically studied as a graph problem. The routing regions and their relationships and capacities are modeled as graphs. However, the design style strongly effects the graph models used and as a result, there are several graph models. Before presenting the problem formulation of global routing, we discuss three different graph models which are commonly used.

The graph models for area routing capture the complete layout information and are used for finding exact route for each net. On the other hand, graph models for global routing capture the adjacencies and routing capacities of routing regions. We discuss three graph models viz; grid graph model, checker board model and the channel intersection graph model. Grid graphs are most suitable for area routing while the channel intersection graphs are most suitable for global routing.

- **Grid Graph Model:** The simplest model for routing is a grid graph. The grid graph $G_1 = (V_1, E_1)$ is a representation of a layout. In this model, a layout is considered to be a collection of unit side square cells arranged in a h×w array. Each cell c_i is represented by a vertex v_i, and there is an edge between two vertices v_i, and v_j, if cells c_i and c_j are adjacent. A terminal in cell c_i is assigned to the corresponding vertex v_i.



(b)

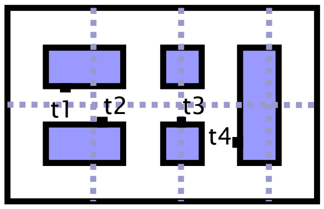


(b)

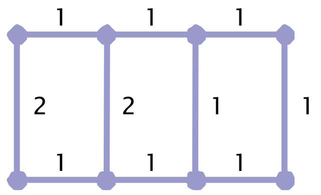
Figure 3.2: Grid Graph Model

The capacity and length of each edge is set equal to one, i.e., $c(e) = 1$, $l(e) = 1$. It is quite natural to represent blocked cells by setting the capacity of the edges incident on the corresponding vertex to zero. Given a grid graph, and a two terminal net, the routing problem is simply to find a path connecting the vertices, corresponding to the terminals, in the grid graph. Whereas, for a multi-terminal net, the problem is to find a Steiner tree in the grid graph. The more general routing problems may consider k-dimensional grid graphs, however, the general techniques for routing essentially remain the same in all grids. In fact, routing in grids, should be considered as area routing, since the actual detailed route of the net is determined.

- **Checker Board Model:** Checker board model is a more general model than the grid model. It approximates the entire layout area as a 'coarse grid' and all terminals located inside a coarse grid cell are assigned that cell number. The checker board graph $G_2 = (V_2, E_2)$ is constructed in a manner analogous to grid graph. The edge capacities are computed based on the actual area available for routing on the cell boundary. Note that the partially blocked edges have unit capacity, whereas, the unblocked edges have a capacity of 2. Given the cell numbers of all terminals of a net, the global routing problem is to find a routing in the coarse grid graph. A checker board graph can also be formed from a cut tree of floorplan. A block b_i in a floorplan is represented by a vertex v_i and there is an edge between vertices v_i and v_j if the corresponding blocks b_i and b_j are adjacent to each other. Note that, unlike the cells in a grid, two adjacent modules in a cut tree of a floorplan may not entirely share a boundary with each other.



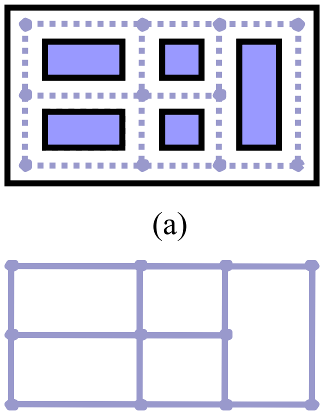
(a)



(b)

Figure 3.3: Checker Board Graph

- **Channel Intersection Graph Model:** The most general and accurate model for global routing is the channel intersection model. Given a layout, we can define a channel intersection graph $G_3 = (V_3, E_3)$, where each vertex v_i Є V_3 represents a channel intersection CI_i .Two vertices v_i and v_j are adjacent in G_3 if there exists a channel between CI_i and CI_j .In other words, the channels appear as edges in G_3. Let c(e) and I(e) be the capacity and length of a channel associated with edge e Є E_3. The channel intersection graph should be extended to include the pins as vertices so that the connections between the pins can be considered in this graph. The global routing problem of two terminal nets is to find path for each net, in the routing graph such that the desired objective function is optimized. In addition, the number of nets using each edge (traffic through the corresponding channel) should not violate the capacity of that edge.


(a)

(b)

Figure 3.4: Channel Intersection Graph

It is obvious that routing of one net at a time causes ordering problem for nets. It is important to note that the overall optimal solution may consist of sub optimal solutions of individual nets. For a net with more than two terminals, the path model discussed above is not appropriate. In fact, global routing of multi-terminal nets can be formulated as a Steiner Tree problem. A Steiner Tree is a tree interconnecting a set of specified points called demand points and some other points called Steiner points. The number of Steiner points is arbitrary. The global routing problem can be viewed as a problem of finding a Steiner tree for each net in the routing graph such that the desired objective function is optimized. In addition, the capacity of the edges must not be violated. As discussed earlier, a typical objective function is to minimize the total length of selected Steiner trees. In high-performance circuits, the objective function is to minimize the maximum wire length of selected Steiner trees. A more precise objective function for high-performance circuits is to minimize the maximum diameter of selected Steiner trees. The diameter of a Steiner tree is defined as the maximum length of a path between any two vertices in the Steiner tree. If there is no feasible solution to an instance of a global routing problem, then the netlist is not routable as the capacity constraints of some edges can not be satisfied. In such cases, the placement phase has to be carried out again. The formal statement of global routing problem is as follows: Given, a netlist $N = \{N_1, N_2, \ldots, N_n\}$, the routing graph $G = (V,E)$, find a Steiner tree T_i for each net N_i, $1 <= i <= n$, such that, the capacity constraints are not violated, i.e., $U(e_j) <= c(e_j)$ for all $e_j \in E$, where $U(e_j) = {}^{n}\sum_{i=1} X_{ij}$ is the number of wires that pass through the channel corresponding to edge e_j ($X_{ij} = 1$ if e_j is inT_i, it is 0 otherwise. A typical objective function is to minimize the total wire length ${}^{n}\sum_{i=1\ L(Ti}$)) where $L(T_i)$ is the length of Steiner tree T_i. In the case of high-performance chips the objective function is to minimize the maximum wire length (${}^{n}nax_{i=1}$ $L(T_i)$). Note that minimization of maximum wire length may not directly reduce the diameter reduce the diameter of the Steiner trees.

3.3 Design Style Specific Global Routing Problems

The objective of global routing in each design style is different. We will discuss the global routing problem for full custom, standard cell and gate array.

- **Full custom:** The global routing problem formulation for full custom design style is similar to the general formulation described above. The only difference is how capacity constraints guide the global routing solution. In the general formulation the edge capacities cannot be violated. In full custom, since channels can be expanded, some violation of capacity constraints is allowed. However, major violation of capacities which leads to significant changes. However, major violation of capacities which leads to significant changes in placement are not allowed. In such case, it may be necessary to carry out the placement again.

- **Standard cell:** In the standard cell design style, at the end of the placement phase, the location of each cell in a row is fixed. In addition, the capacity and location of each feed through is fixed. However, the channel heights are not fixed. They can be changed by varying the distance, between adjacent cell rows to accommodate wires assigned by a global router. As a result, they do not have a predetermined capacity. On the other hand, feed throughs have predetermined capacity. The area of a standard cell layout is determined by the total cell row height and the total channel height, where the total cell row height is the summation of all cell row heights and the total channel height is the summation of all channel heights. As the total cell row height is fixed, the layout area could only be minimized by minimizing the total channel height. As a result, standard cell global routers attempt to minimize the total channel height. Other optimization functions include the minimization of the total wire length and the minimization of the maximum wire length.

  The edge set of G = (V,E) are partitioned into two disjoint sets E^v and E^h i.e., $E = E^v \cup E^h$ Edges in E^v represent feedthroughs, whereas, edges in E^h represent channels. Capacity of each edge $e_j \in E^v$ is equal to the number of wires that can pass through the corresponding feed through.

  If there is no feasible solution for a global routing problem, feedthrough capacities are not sufficient. Additional feed throughs should be inserted in order to allow global routing.

  Recently, a new approach, called over-the-cell routing, has been presented for standard cell design, in which, in addition to the channels and feed throughs the over-the-cell areas are available for routing. Availability of over-cell-cell areas changes the global routing problem.

- **Gate array:** In gate array design style, the size and location of all cells and the routing channels and their capacities are fixed by the architecture. This is the key difference between gate array and other design styles. Unlike the full custom design style and standard cell design style the primary objective of the global routing in gate arrays is to guarantee routability. The secondary objective may be to minimize the total wire length or to minimize the maximum wire length. Other than these objectives, the formulation of global routing problem gate array design style is same as the general global routing formulation. If there is no feasible solution to a given instance of global routing problem, the netlist can not be routed. In this case, the placement phase has to be carried out again as the capacity of routing channels is fixed in gate array design style.

3.4 Classification of Global Routing Algorithms

Basically there are two kinds of approaches to solve global routing problem; the sequential and the concurrent.

❖ **Sequential Approach:** In this approach, as the name suggests, nets are routed one by one. However, once a net has been routed it may block other nets which are yet to be routed. As a result, this approach is very sensitive to the order in which the nets are considered for routing. Usually, the nets are sequenced according to their criticality, perimeter of the bounding rectangle and number of terminals. The criticality of a net is determined by the importance of the nets. For example, clock net may determine the performance of the circuit and therefore it is considered to be a very important net. As a result, it is assigned a high criticality number. The nets on the critical paths are assigned high criticality numbers since they also play a key role in determining the performance of the circuit. The criticality number and other factors can be used to sequence nets. However, sequencing techniques do not solve the net ordering problem satisfactorily. In a practical router, in addition to a net ordering scheme an improvement phase is used to remove blockages when further routing of nets is not possible. However, this also may not overcome the shortcoming of sequential approach. One such improvement phase involves 'rip-up and reroute' technique, while other involves 'shove-aside' technique. In 'rip-up and reroute', the interfering wires are ripped up, and rerouted to allow routing of the affected nets. Whereas, in 'shove-aside' technique, wires that allow completion of failed connections are moved aside without breaking the existing connections. Another approach is to first route simple nets consisting of only two or three terminals since there are few choices for routing such nets. Usually such nets comprise a large portion of the nets (up to 75%) in a typical design. After the simple nets have been routed, a Steiner tree algorithm is used to route intermediate nets. Finally, a maze routing algorithm is used to route the remaining long nets which are not too numerous.

The sequential approach includes:

- Two-terminal algorithms:
  - Maze routing algorithms
  - Line –prove algorithms
  - Shortest path based algorithms
- Multi-terminal algorithm
  - Steiner tree based algorithms

❖ **Concurrent Approach:** This approach avoids the ordering problem by considering routing of all the nets simultaneously. The concurrent approach is computationally hard and no efficient polynomial algorithms are known ever for two-terminal nets. As a result, integer programming methods have been suggested. The corresponding integer program is usually to large to be employed efficiently. Hence, hierarchical methods that work top down are employed to partitioned the problem into smaller subprograms, which can be solved by integer programming.

Chapter 4

4.1 Maze Routing Algorithms

Lee introduced an algorithm for routing a two terminal net on a grid in 1961. Since then, the basic algorithm has been improved for both speed and memory requirements. Lee's algorithm and its various improved versions form the class of maze routing algorithms.

Maze routing algorithms are used to find a path between a pair of points, called the source(s) and the target (t) respectively, in a planar rectangular grid graph. The geometric regularity in the standard cell and gate array design style lead us to model the whole plane as a grid. The areas available for routing are represented as unblocked vertices, whereas, the obstacles are represented as blocked vertices. The objective of a maze routing algorithm is to find a path between the source and the target vertex without using any blocked vertex. The process of finding a path begins with the exploration phase, in which several paths start at the source, and are expanded until one of them reaches the target. Once the target is reached, the vertices need to be retraced to the source to identify the path. The retrace phase can be easily implemented as long as the information about the parentage of each vertex is kept during the exploration phase.

- **Lee's maze routing algorithm:** This algorithm which was developed by Lee is the most widely used algorithm for finding a path between any two vertices on a planar rectangular grid. The key to the popularity of Lee's maze router is its simplicity and its guarantee of finding an optimal solution if one exists.

  The exploration phase of Lee's algorithm is an improved version of the breadth-first search. The search can be visualized as a wave propagating from the source. The source is labeled '0' and the wavefront propagates to all the unblocked vertices adjacent to the source. Every unblocked vertex adjacent to the source is marked with a label '1'. Then, every unblocked vertex adjacent to vertices with a label '1' is marked with a label '2', and so on. This process continues until the target vertex is reached or no further expansion of the wave can be carried out. Due to the breadth-first nature of the search, Lee's maze router is guaranteed to find a path between the source and target, if one exists. In addition, it is guaranteed to be the shortest path between the vertices.

The input to the Lee's Algorithm is an array B, the source(s) and target (t) vertex. B[v], denotes if a vertex v is blocked or unblocked. The algorithm uses an array L, where L[v] denotes the distance from the source to the vertex v. this array will be used in the procedure RETRACE that retraces the vertices to form a path P, which is the output of the Lee's Algorithm. Two linked lists plist (Propagation list) and nlist (Neighbor list) are used to keep track of the vertices on the wavefront and their neighbor vertices respectively. These two lists are always retrieved from tail to head. We also assume that the neighbors of a vertex are visited in counter-clockwise order that is top, left, bottom and then right.

The formal description of the Lee's Algorithm is shown below

```
Algorithm LEE-ROUTER(B,s,t,P)
        Input: B,s,t
        Output: P
Begin
        plist = s;
        nlist = $\Phi$;
        temp = 1;
        path_exists = FALSE;
        while plist != $\Phi$ do
           for each vertex $v_i$ in plist do
             for each vertex $v_j$ neighboring $v_{i\ do}$
             if B[$v_j$] = UNBLOCKED then
               L[$v_j$] = temp;
               INSERT($v_j$ ,nlist);
               if $v_j$ = t then
                  path_exists = TRUE;
                  exit while;
           temp = temp+1;
```

```
            plist = nlist;
            nlist = Φ;
        if path_exists = TRUE then RETRACE(L,P);
        else path does not exists;
end.
```

The time and space complexity of Lee’s algorithm is $O(h \times w)$ for a grid of dimension $h \times w$. The Lee’s routing algorithm requires a large amount of storage space and its performance degrades rapidly when the size of the grid increases. There have been numerous attempts to modify the algorithm to improve its performance and reduce its memory requirements.

Lee’s algorithm requires up to k+1 bits per vertex, where k bits are used to label the vertex during the exploration phase and an additional bit is needed to indicate whether the vertex is blocked. For an $h \times w$ grid, $k = \log_2(h \times w)$.

- **Multilayer Maze Routing:** Multiple layers are necessary in routing when connections must "cross over" each other. The routing surface is represented as a set of stacked grid in which each gridpoint can be the source of a connection, a destination of a connection, or a wire that connects adjacent gridpoints (horizontally or vertically). Here we can make additional connections by clicking to select additional source and destination points. To route connections, follow the prompt and click on the source and destination points. The output will then show the search performed by the Lee Algorithm and the final connection that is made.

- **Soukup’s maze routing algorithm:** Lee’s algorithm explores the grid symmetrically, searching equally in the directions away from target as well as in the directions towards it. Thus, Lee’s algorithm requires a large search time. In order to overcome this limitation, Soukup proposed an iterative algorithm. During each iteration, the algorithm explores in the direction toward the target without changing the direction until it reaches the target or an obstacle, otherwise it goes away from the target. If the target is reached, the exploration phase ends. If the target is not reached, the search is conducted iteratively. If the search goes away from the target, the algorithm simply changes the direction so that it goes towards the target and a new iteration begins. However, if an obstacle is reached, the breadth-first search is employed until a vertex is found which can be used to continue the search in the direction toward the target. Then, a new iteration begins. The formal description of Soukup's Algorithm is shown below:

```
Algorithm SOUKUP-ROUTER(B,s,t,P)
   Input: B,s,t
   Output: P
begin
   plist = s;
   nlist = Φ;
   temp = 1;
   path_exists = FALSE;
   while plist != Φ do
      for each vertex $v_i$ in plist do
         for each vertex $v_j$ neighboring $v_i$ do
            if $v_j$ = t then
               L[$v_j$] = temp;
               Path_exists = TRUE;
                Exit while;
            if B[$v_j$] = UNBLOCKED then
            (*If the direction of the search is toward the target , the search continues
            in this direction*)
            if DIR($v_i$,$v_j$) = TO-TARGET
            then L[$v_j$] = temp;
                temp = temp+1;
                INSERT($v_j$,plist);
                While B[NGHBR-IN-DIR($v_i$,$v_j$)] = UNBLOCKED do
                 Vj = NGHBR-IN-DIR($v_i$,$v_j$);
                 L[$v_j$] = temp;
                  Temp = temp+1;
                INSERT($v_j$,plist);
            else
                L[$v_j$] = temp;
                Temp = temp+1;
                INSERT($v_j$,nlist);
    plist = nlist;
    nlist = Φ;
    if path_exists = TRUE then RETRACE(L,P);
    else path does not exist;
end.
```

The notation used in the algorithm is similar to that used in the Lee's algorithm except for the array L. We use L[v] to denote the order in which the vertex v is visited during the exploration phase in this algorithm. Function DIR(v_1,v_2) returns the direction from v_1 to v_2. function NGHBR-IN-DIR(v_1,v_2) returns the neighbor of v_2 which is in the direction from v_1 to v_2.

The soukup's Algorithm improves the speed of Lee's algorithm by a factor of 10 to 50. it guarantees finding a path if a path between source and target exits. However, this path may not be the shortest one. The search method for this algorithm is a combined breadth-first and depth- first search. The worst case time and space complexities for this algorithm are both O(h×w), for a grid of size h×w.

4.2 Comparison of Maze Routing Algorithms

Maze routing algorithms are grid based methods. The time and space required by these algorithms by these algorithms depend linearly on their search space.

The search in Lee's algorithm is conducted by using a wave propagating from the source. The algorithm searches symmetrically in every direction, using the breath-first search technique. Thus, it guarantees finding a shortest path between any two vertices if such a path exists. However, the worst case happens when the source is located at the center and the target is located at a corner of routing area, in which all the vertices have to be scanned before the target is reached.

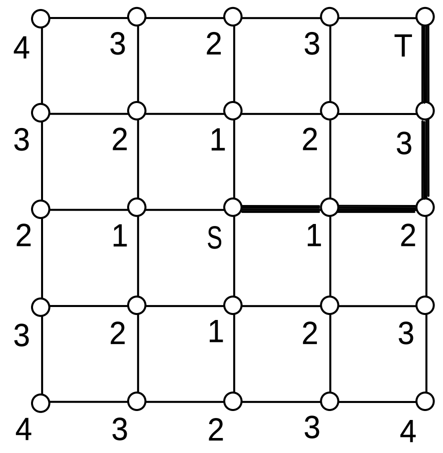


Figure 4.1: Lee’s Algorithm in the Worst Case

The Soukup’s algorithm remedies the shortcoming of the breadth-first search method by using a depth-first search until an obstacle is encountered. If an obstacle is encountered, a breadth-first search method is used to get around the obstacle. The search time in Soukup’s algorithm is usually smaller than the Lee’s algorithm is usually smaller than the Lee’s algorithm due to the nature of depth-first search method. However, this algorithm may not find a shortest path between the source and target. The Soukup’s algorithm explores all the vertices and does not find the shortest path between s and t.

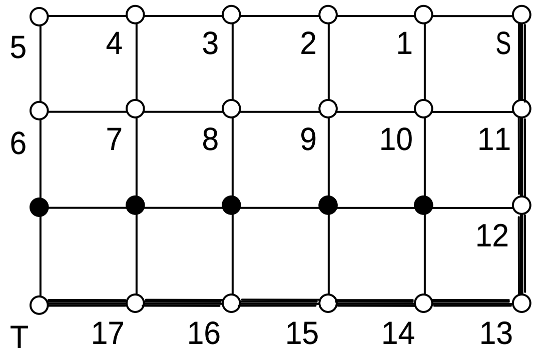


Figure 4.2: Soukup’s Algorithm in the Worst Case

The worst case of Soukup’s algorithm occurs when the search goes in the direction of the target, which is opposite the direction of the passageway through the obstacle. Figure 4.3

shows an example in which soukup's algorithm scans all vertices while finding a path between s and t.

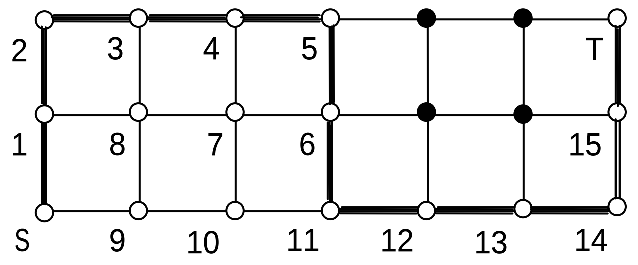


Figure 4.3: Soukup's Algorithm doesn't find the shortest path

Chapter 5

Other Routing Algorithms

5.1 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 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' includes a tree and the total cost of edges $\sum_{e_i \in E'} wt(e_i)$, is minimum over all such tree, where $wt(e_i)$is the cost or weight of the edge e_i.

Minimum Spanning Trees

A spanning tree of a graph is just a subgraph that contains all the vertices and is a tree. A graph may have many spanning trees; for instance the complete graph on four vertices

```
o---o
|\ /|
| X |
|/ \|
o---o
```

has sixteen spanning trees:

```
o---o    o---o    o   o    o---o
|   |    |        |   |        |
|   |    |        |   |        |
|   |    |        |   |        |
o   o    o---o    o---o    o---o

o---o    o   o    o   o    o   o
\ /      |\ /      \ /      \ /|
X        | X        X        X |
/ \      |/ \      / \      / \|
o   o    o   o    o---o    o   o

o   o    o---o    o   o    o---o
|\  |        /    |  /|     \
| \ |       /     | / |      \
|  \|      /      |/  |       \
o   o    o---o    o   o    o---o
```

```
o---o    o   o    o   o    o---o
|\       |  /      \  |       /|
| \      | /        \ |      / |
|  \     |/          \|     /  |
o   o    o---o    o---o    o   o
```

Now suppose the edges of the graph have weights or lengths. The weight of a tree is just the sum of weights of its edges. Obviously, different trees have different lengths. The problem is how to find the minimum length spanning tree.This problem can be solved by many different algorithms. There are several "best" algorithms, depending on the assumptions we make.These algorithms are all quite complicated, and probably not that great in practice unless we are looking at really huge graphs. There are basically three algorithms for finding a MST:

- ❖ Boruvka’s Algorithm
- ❖ Kruskal Algorithm
- ❖ Prim’s Algorithm

The standard application is to a problem like phone network design. You have a business with several offices; you want to lease phone lines to connect them up with each other; and the phone company charges different amounts of money to connect different pairs of cities. You want a set of lines that connects all your offices with a minimum total cost. It should be a spanning tree, since if a network isn't a tree you can always remove some edges and save money. A less obvious application is that the minimum spanning tree can be used to approximately solve the traveling salesman problem. A convenient formal way of defining this problem is to find the shortest path that visits each point at least once. Note that if you have a path visiting all points exactly once, it's a special kind of tree. For instance in the example above, twelve of sixteen spanning trees are actually paths. If you have a path visiting some vertices more than once, you can always drop some edges to get a tree.

How to find minimum spanning tree

The stupid method is to list all spanning trees, and find minimum of list. We already know how to find minima. But there are far too many trees for this to be efficient. It's also not really an algorithm, because you'd still need to know how to list all the trees. A better idea is to find some key property of the MST that lets us be sure that some edge is part of it, and use this property to build up the MST one edge at a time. For simplicity, we assume that there is a unique minimum spanning tree.

Prim's Algorithm

Rather than build a subgraph one edge at a time, Prim's algorithm builds a tree one vertex at a time.

```
 Prim's algorithm:
let T be a single vertex x
while (T has fewer than n vertices)
{
    find the smallest edge connecting T to G-T
    add it to T
}
```

Since each edge added is the smallest connecting T to G-T, the lemma we proved shows that we only add edges that should be part of the MST. Again, it looks like the loop has a slow step in it. But again, some data structures can be used to speed this up. The idea is to use a heap to remember, for each vertex, the smallest edge connecting T with that vertex.

```
Prim with heaps:
make a heap of values (vertex,edge,weight(edge))
    initially (v,-,infinity) for each vertex
    let tree T be empty
while (T has fewer than n vertices)
{
    let (v,e,weight(e)) have the smallest weight in the heap
    remove (v,e,weight(e)) from the heap
    add v and e to T
    for each edge f=(u,v)
    if u is not already in T
        find value (u,g,weight(g)) in heap
        if weight(f) < weight(g)
        replace (u,g,weight(g)) with (u,f,weight(f))
}
```

Analysis

We perform n steps in which we remove the smallest element in the heap, and at most 2m steps in which we examine an edge f=(u,v). For each of those steps, we might replace a value on the heap, reducing it's weight. (You also have to find the right value on the heap, but that can be done easily enough by keeping a pointer from the vertices to the corresponding values.) I haven't described how to reduce the weight of an element of a binary heap, but it's easy to do in O(log n) time. Alternately by using a more complicated data structure known as a Fibonacci heap, you can reduce the weight of an element in constant time. The result is a total time bound of O(m + n log n).

A Rectilinear Minimum Spanning Tree (RSMT) using Prim's Algorithm connects all terminals together in a way that minimizes the overall distance of the connections in this tree. Prim's Algorithm operates incrementally on a partial tree starting with a single terminal - on each iteration, it adds an edge between the partial tree and the terminal that is closest to any terminal in the partial tree. Given a set of terminal points in a plane, a rectilinear minimum spanning tree is a set of edges which connects all the terminals with minimum rectilinear distance. Rectilinear distance (also called "Manhattan Distance") is the sum of the horizontal and vertical distance between two points. It is used instead of Euclidean distance in VLSI CAD applications because VLSI processes support only horizontal and vertical wires. The RMST can be used as an estimate of the length of a net that connects together multiple terminals. The RMST Problem is a special case of the Minimum Spanning Tree problem studied in Computer Science Algorithms textbooks in that the distances between the points imply a complete graph.

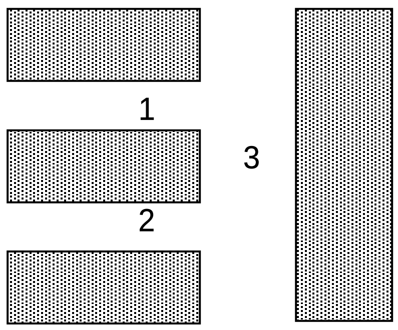
Chapter 6

6.1 Detailed Routing Algorithms

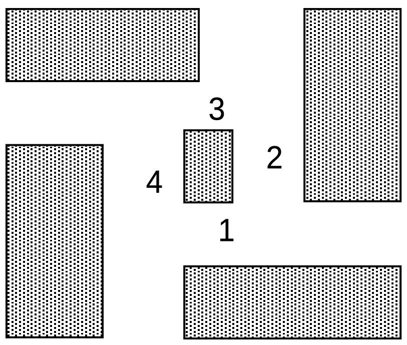
In a two-phase routing approach, detailed routing follows the global routing phase. During the global routing phase, wire paths are constructed through a subset of the routing regions, connecting the terminals of each net. Global routers do not define the wires, instead, they use the original net information and define a set of restricted routing problems. The detailed router places the actual wire segments within the region indicated by the global router, thus completing the required connections between the terminals.

The detailed routing problem is usually solved incrementally, in other words, the detailed routing problem is solved by routing one region at a time in a predefined order. The ordering of the regions is determined by several factors including the criticality of routing certain nets and the total number of nets passing through a region. A routing region may have terminals only on two opposite sides of the region. This type of routing region is called a channel. On the other hand, a rectangular routing region may be closed, and may have terminals on all four sides of the region. This type of routing region is called a switchbox.

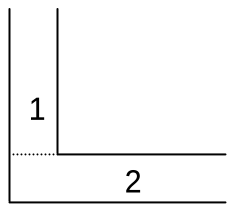
The area of the routing region can be determined exactly only after the routing is completed. If this area is different than the area estimated by the placement algorithm, the placement has to be adjusted to account for this difference in area. If the floorplan is slicing then a left to right sweep of the channels can be done such that no routed channel has to be ripped up to account for the change of areas.



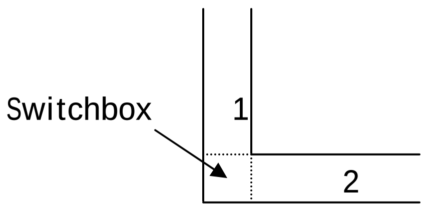
(a)



(b)



(c)



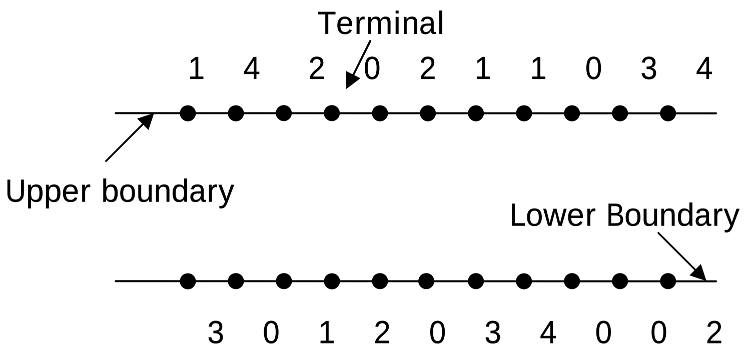
(d)

Figure 6.1: Channels and Switchboxes

Consider the example shown in the Figure 6.1(a). In this floorplan, if channel1 is routed first followed by routing of channel 2 and channel 3, no rerouting would be necessary. In fact, complete routing without rip-up of an already routed channel is possible if the channels are routed in the reverse partitioning order. If the floorplan is non-slicing, it may not be possible to order the channels such that no channel has to be ripped up. Consider the example shown in Figure 6.1(b). In order to route channel 2, channel 1 has to be routed so as to define all the terminals for channel 2. Channel 2 has to be routed before channel 3 and channel 3 before channel 4. Channel 4 requires routing of channel 1 giving rise to a cyclic constraint for ordering the channels. This situation is resolved by the use of L-channels or switchboxes. L-channels are not simple to route and are usually decomposed. Figure 6.1(c) shows decomposition of an L-channel into two 3-sided channels while Figure 6.1(d) shows decomposition of an L-channel into two 3-sided channels and a switchbox.

6.2 Channel Routing Problems

A channel is a routing region bounded by two parallel rows of terminals. Without loss of generality, it is assumed that the two rows are horizontal. The top and the bottom rows are also called top boundary and the bottom boundary, respectively. Each terminal is assigned a number which represents the net to which that terminal belongs as shown in the Figure 6.2.



Net list : 1 4 2 0 2 1 1 0 3 4 0
3 0 1 2 0 3 4 0 0 2 3

Figure 6.2: A channel and its associated net list

Terminal numbered zero are called vacant terminals. A vacant terminal does not belong to any net and therefore requires no electrical connection. The net list of a channel is the primary input to most of the routing algorithms.

The horizontal dimension of the routed channel is called the channel length and the vertical dimension of the routed channel is called the channel height. The horizontal segment of a net is a trunk and the vertical segments that connect the trunk to the terminals are called its branches. The horizontal line along which a truck is placed is called a track. A dogleg is a vertical segment that is used to maintain the connectivity of the two trunks of a net on two different tracks. The pictorial representation of the terms mentioned above is shown in the Figure 6.3.

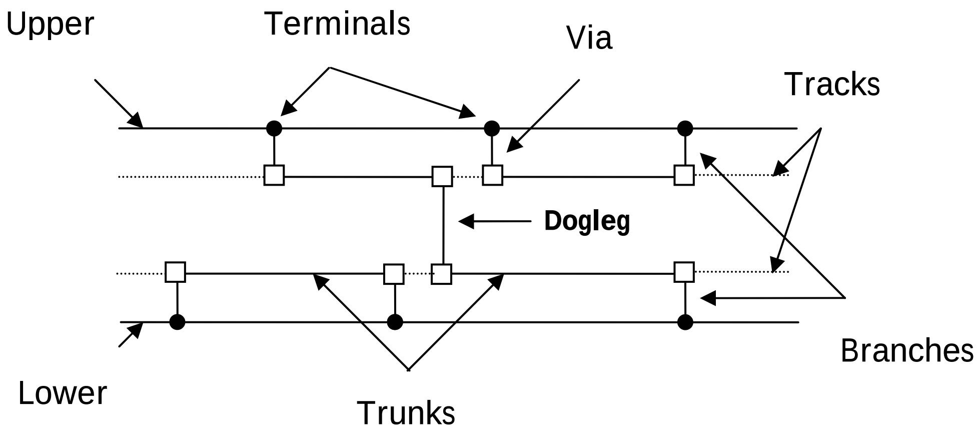


Figure 6.3: Terminology for Channel Routing Problems

A channel routing problem (CRP) is specified by four parameters: Channel length, Top (Botom) terminal list, Left (Right) connection list, and the number of layers. The channel length is specified in terms of number of columns in grid based models, while in gridless models it is specified in terms of λ. The Top and Bottom lists specify the terminals in the channel. The Top list is denoted by T = (T_1,T_2,........,T_m) and the bottom list by B = (B_1,B_2,……,B_m). In the grid based models, T_i (B_i) is the net number for the terminal at the top(bottom) of the ith column, or is 0 if the terminal does not belong to any net. In gridless model, each terminal, T_i (B_i) indicates the net number to which the ith terminal. The Left (Right) connection list, consists of nets that enter the channel from the left (right) end of the channel. It is an ordered list if the channel to the left (right) of the given channel has already been routed.

Given the above specifications, the problem is to find the interconnections of all the nets in the channel including the connection sets so that the channel uses minimum possible area. A solution to a channel routing problem is a set of horizontal and vertical segments for each net. This set of segments must make all terminals of the net electrically equivalent. In the grid based model, the solution specifies the channel height in terms of the total number of tracks required for routing. In gridless models, the channel height is specified in terms of λ.

After Global Routing each channel can be considered as a separate detailed routing problem. A channel will contain some number of connections, each involving logic block pins or vertical feed-throughs. The task of detailed routing algorithm is to allocate wire segments for each connection in a way that allows all connections to be completed. In addition, it may be necessary to minimize the routing delays of the connections. This can be accomplished by limiting the total number and length of segments used by a connection. The appropriate algorithm depends on the segmentation of the tracks and the requirements for minimizing delays.

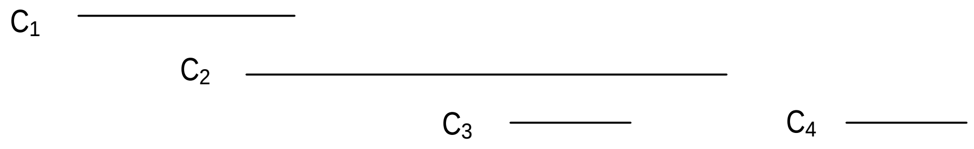


Figure 6.4: A set of Four Connections to be Routed

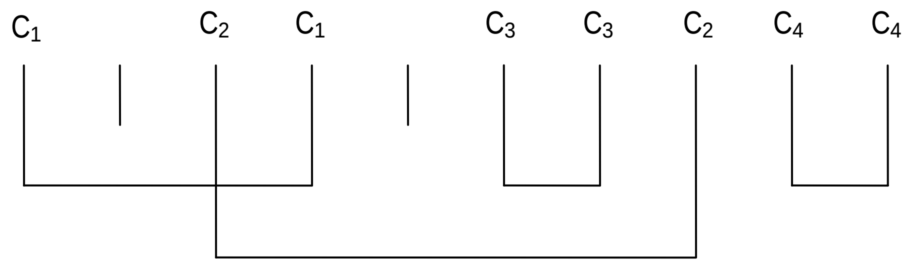


Figure 6.5: Routing of Connections in a Mask-Programmed Channel

Figure 6.4 shows an example of a routing problem in which the connections called C1 to C4 are to be routed. Figure 6.5 indicates how the connections might be routed in a masked programmed channel, where there is a complete freedom for placing the wire segments. The channel is divided into columns, as shown by the vertical segments in the figure. Some columns represent logic block pins and other are vertical feed-throughs as the column labels in the figure indicate, each of C1 to C4 specifies two columns that are to be connected. Figure 6.6-6.9 present several different scenarios for a segmented FPGA channel, and suggests routing solutions for each segmentation. Routing switches are indicated by circles, with an ON switch drawn solid and an OFF switch drawn hollow.

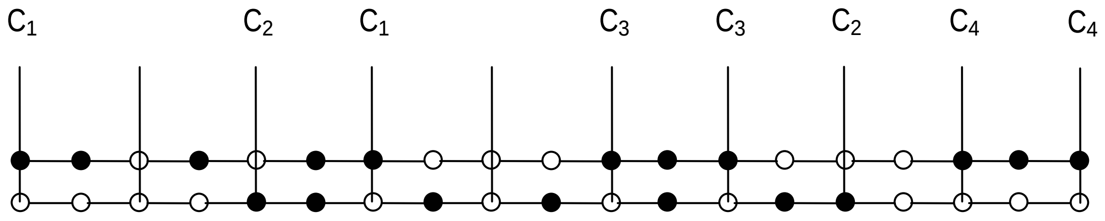


Figure 6.6: Routing in a Fully Segmented Channel

Figure 6.6 depicts one extreme for the track segmentations, in which every track is fully segmented. In this case, each segment spans only one column, meaning that multiple segments are required for every connection

As shown, the four connections can be routed in this architecture with only two tracks.

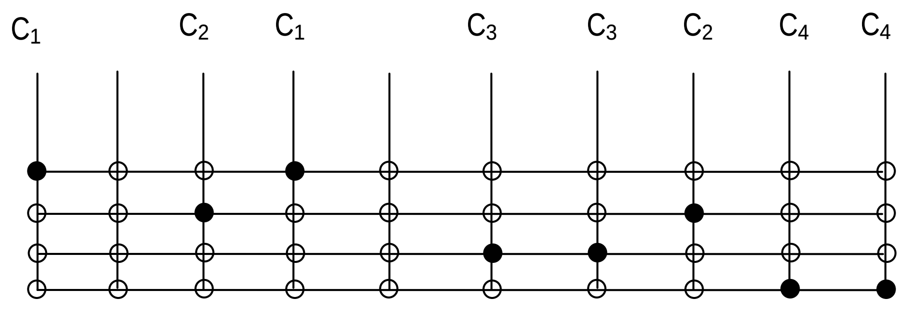


Figure 6.7: Routing in a Non-Segmented Channel

The opposite extreme to full segmentation is depicted in Figure 6.7, which shows a channel in which each track contains only one segment for its entire length. In this case, the number of tracks required for routing is always equal to the number of connections. No routing algorithm is necessary since any choice of track for a connection will do.

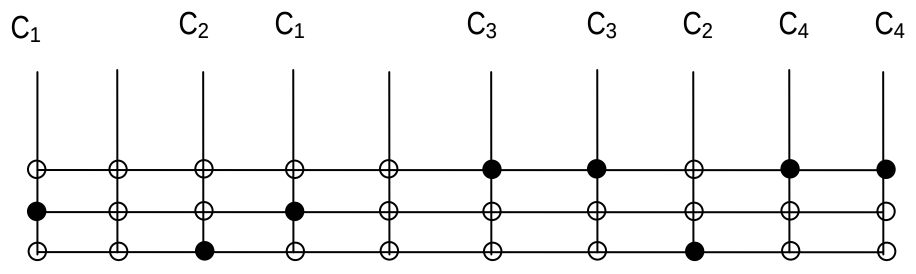


Figure 6.8: Segmented for 1-Segment Routing

An intermediate approach to channel segmentation is illustrated in Figure 6.8, where the tracks have segments of various lengths. Each connection must be routed in a single segment, since no switches are available where segments in the same track meet.
Additional flexibility in the channels can be added by allowing segments that about to be joined by switches, as depicted in Figure 6.9. This implies that connections can occupy more than one segment and greatly increases the complexity of the problem.

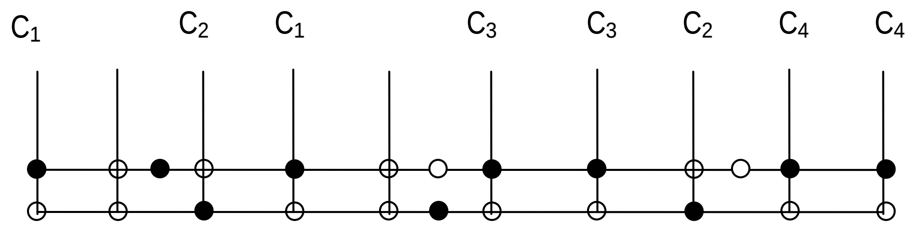


Figure 6.9: Segmented for 2-Segment Routing

The main objective of the channel routing is to minimize the channel height. Additional objectives functions, such as, minimizing the total number of vias used in a multiplayer routing solution, and minimizing the length of any particular net are also used.

Bibliography

[1] M. Sarrafzadeh and C. K. Wong, *An Introduction to VLSI Design*, McGraw-Hill, 1996.

[2] N.Sherwani, Algorithms for VLSI Physical Design Automation,ISBN 0-7923-9294-9.

[3] Karger, Klein, and Tarjan, "A randomized linear-time algorithm to find minimum spanning trees", J. ACM, vol. 42, 1995, pp. 321-328.

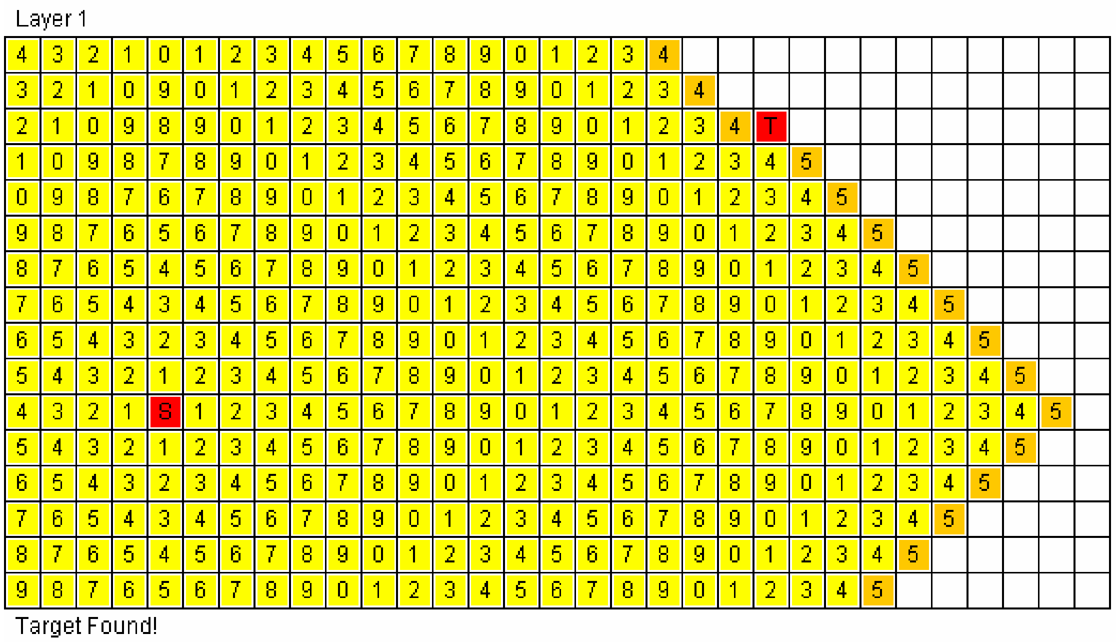
[4] A. Kahng and I. Mandoiu, "GSRC Bookshelf RMST-Pack: Rectilinear Minimum Spanning Tree Algorithms", June 2001.

[5] www.ieee.org

Appendixes

Appendix A

Output Results


Layer 1

4	3	2	1	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4												
3	2	1	0	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4											
2	1	0	9	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	T									
1	0	9	8	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5								
0	9	8	7	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5							
9	8	7	6	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5						
8	7	6	5	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5					
7	6	5	4	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5				
6	5	4	3	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5			
5	4	3	2	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5		
4	3	2	1	S	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	
5	4	3	2	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5		
6	5	4	3	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5			
7	6	5	4	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5				
8	7	6	5	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5					
9	8	7	6	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5						

Target Found!

Figure A.1: Single Layer Maze Routing (a)

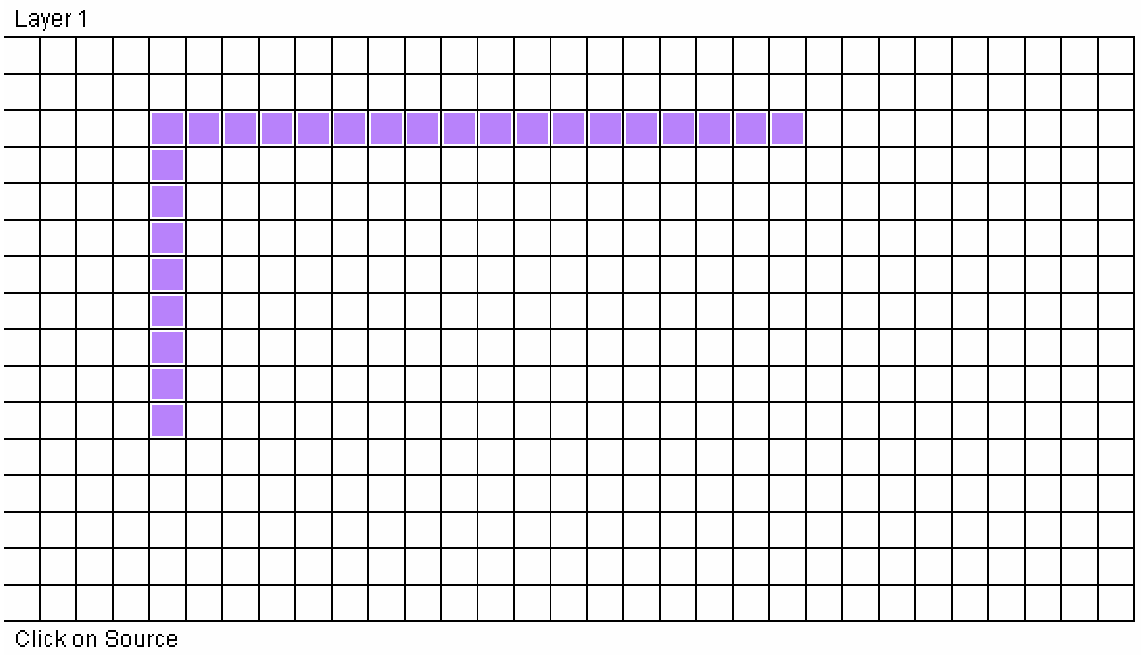


Figure A.2: Single Layer Maze Routing (b)

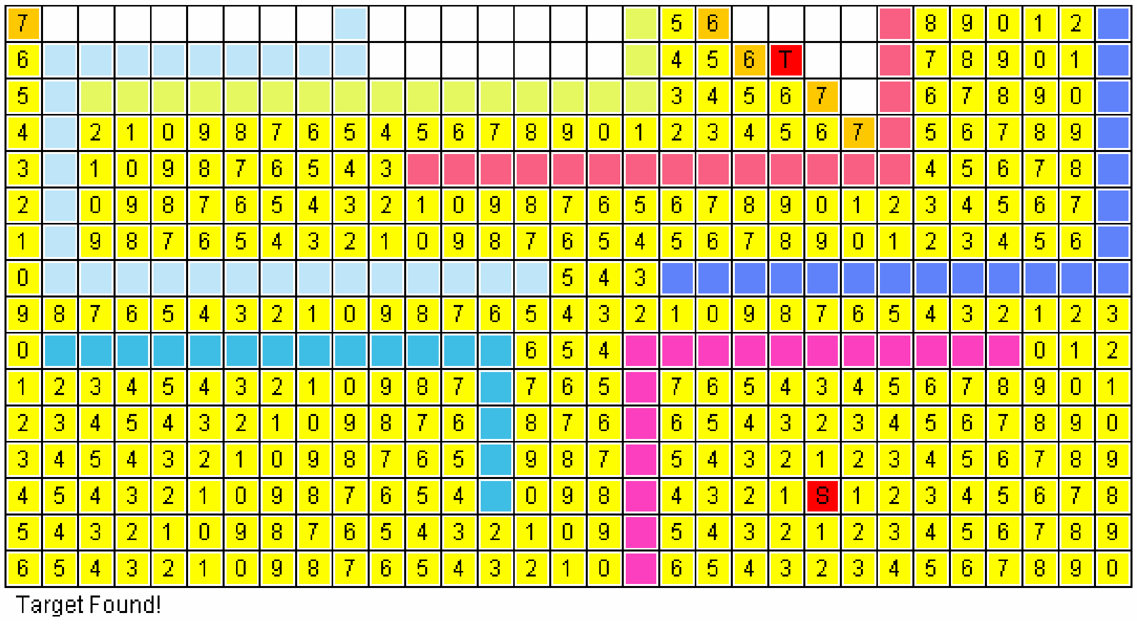


Figure A.3: Multi Layer Maze Routing (a)

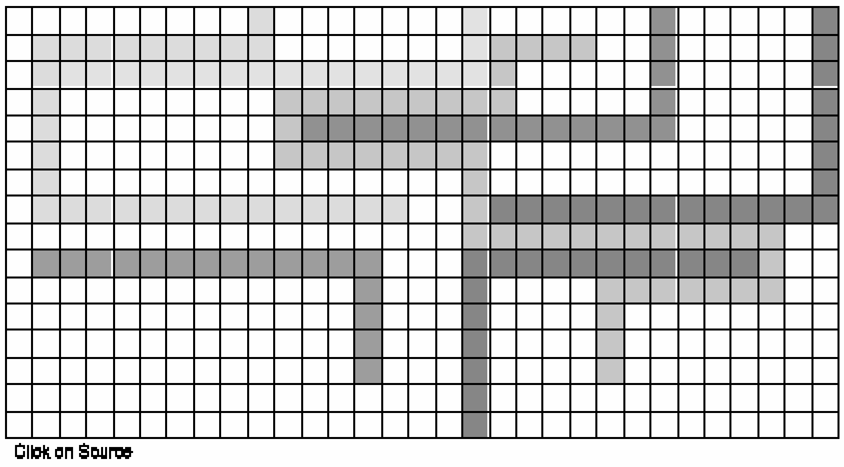


Figure A.4: Multi Layer Maze Routing (b)

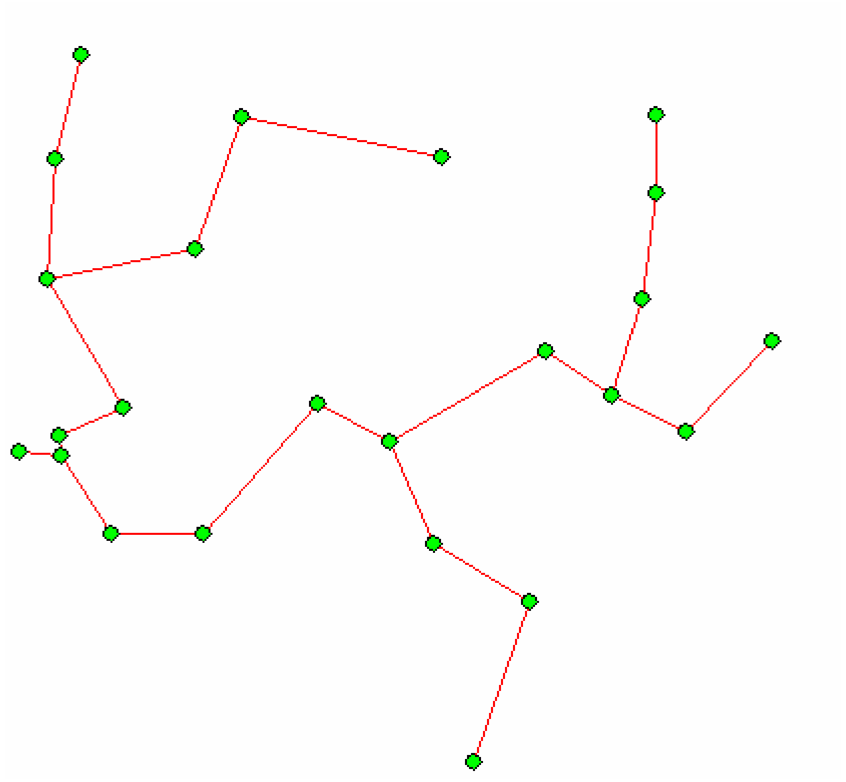


Figure A.5: Minimum Spanning Tree

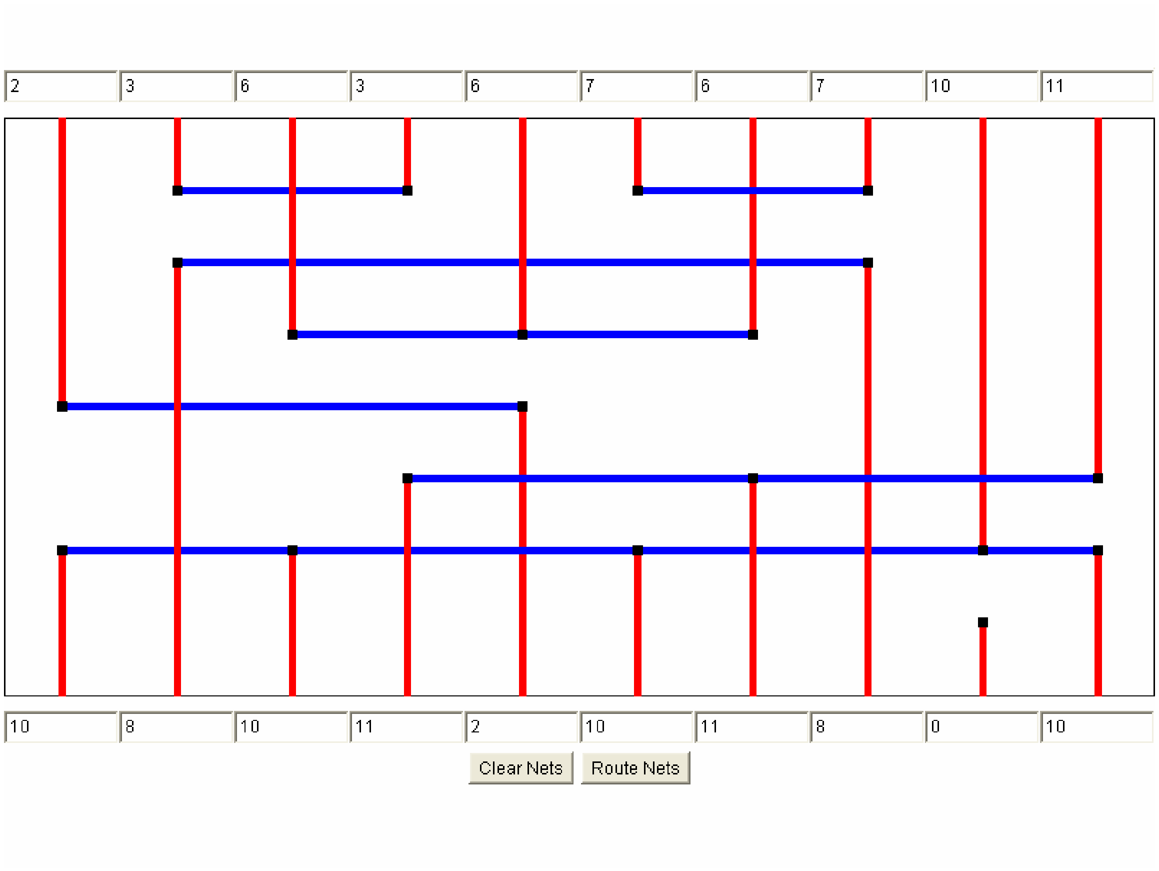


Figure A.6: Channel Routing

Appendix B

Source Code of Various Routing Algorithms

Lee.cpp

```
#include<stdio.h>
#include<conio.h>
#include<dos.h>
#include<graphics.h>
#include<stdlib.h>
#define BLOCKED 1
#define UNBLOCKED 0
#define TRUE 1
#define FALSE 0

int path_exits,path[100],pathlength;
typedef struct
{
int vertexno;
int state;
int distance;
}vertex;

struct list
{
vertex info;
list *next;
};

typedef list* listptr;

void insert(vertex,listptr*);
void retrace(vertex[50][50],int);
void formgrid(void);
int final,m,n,gd,gm,nodes1=0;
FILE *fp;

void main(int argc,char*argv[])
{
int i,j,source,target,blocked,errorcode,sourcex,sourcey;
int temp,path_exists,noblocked;
char *str;
list *plist,*nlist,*ptr;
vertex grid[50][50];
clrscr();
fp=fopen("usercomp.txt","a");
printf("Enter no. of rows:");
scanf("%d",&m);
printf("Enter no. of columns:");
scanf("%d",&n);
for(i=0;i<m;i++)
for(j=0;j<n;j++)
{
grid[i][j].vertexno=i*n+j;
grid[i][j].state=UNBLOCKED;
grid[i][j].distance=-1;
}
printf("Enter source pin:");
```

```
scanf("%d",&source);
printf("Enter target pin:");
scanf("%d",&target);
printf("Enter no. of Blocked pins:");
scanf("%d",&noblocked);
for(i=0;i<noblocked;i++)
{
printf("Enter %d Blocked pin:",i+1);
scanf("%d",&blocked);
grid[blocked/n][blocked%n].state=BLOCKED;
}
fprintf(fp,"\n\n\nLEE ALGORITHM\n");
fprintf(fp,"rows:%d\t",m);
fprintf(fp,"columns:%d\n",n);
fprintf(fp,"source vertex number:%d\t",source);
fprintf(fp,"target vertex number:%d\n",target);
fprintf(fp,"vertices number which are blocked:\n");
for(i=0;i<noblocked;i++)
fprintf(fp,"%s\t",argv[6+i]);
sourcex=source/n;
sourcey=source%n;
grid[sourcex][sourcey].distance=0;
plist=(list*)(malloc(sizeof(list)));
plist->info=grid[sourcex][sourcey];
plist->next=NULL;
nlist=NULL;
temp=1;
path_exits=0;
while(plist!=NULL)
{
for(ptr=plist;ptr!=NULL;ptr=ptr->next)
{
sourcex=(ptr->info).vertexno/n;
sourcey=(ptr->info).vertexno%n;
if(grid[sourcex-1][sourcey].state==UNBLOCKED       &&       (sourcex-1>=0       &&       grid[sourcex-
1][sourcey].distance==-1))
{
nodes1++;
grid[sourcex-1][sourcey].distance=temp;
insert(grid[sourcex-1][sourcey],&nlist);
if (grid[sourcex-1][sourcey].vertexno==target)
{
path_exits=1;
goto jump;
}
}
if((grid[sourcex][sourcey-1].state==UNBLOCKED)&&((sourcey-1)>=0)&&(grid[sourcex][sourcey-
1].distance==-1))
{nodes1++;
grid[sourcex][sourcey-1].distance=temp;
insert(grid[sourcex][sourcey-1],&nlist);
if(grid[sourcex][sourcey-1].vertexno==target)
{
path_exists=1;
goto jump;
}
}
if(grid[sourcex+1][sourcey].state==UNBLOCKED&&((sourcex+1)<m)&&(grid[sourcex+1][sourcey].dist
ance==-1))
{
nodes1++;
grid[sourcex+1][sourcey].distance=temp;
insert(grid[sourcex+1][sourcey],&nlist);
if(grid[sourcex+1][sourcey].vertexno==target)
{
path_exists=1;
goto jump;
}
}
if(grid[sourcex][sourcey+1].state==UNBLOCKED&&((sourcey+1)<n)&&(grid[sourcex][sourcey+1].dista
nce==-1))
{
nodes1++;
grid[sourcex][sourcey+1].distance=temp;
insert(grid[sourcex][sourcey+1],&nlist);
if(grid[sourcex][sourcey+1].vertexno==target)
{
path_exists=1;
goto jump;
}
}
}
temp=temp+1;
plist=nlist;
nlist=NULL;
}
jump:
if(path_exists==1)
{
printf("path exists\n");
final=temp-1;
retrace(grid,target);
}
else
{
printf("path does not exist\n");
getch();
}
fclose(fp);
}

void insert(vertex v,listptr *p)
{
list *temp,*ptr;
if(*p==NULL)
{
ptr=(list*)(malloc(sizeof(list)));
ptr->info=v;
ptr->next=NULL;
*p=ptr;
}
else
```

```
{
ptr=(list*)(malloc(sizeof(list)));
ptr->info=v;
ptr->next=*p;
*p=ptr;
}
}

void retrace(vertex grid[50][50],int target)
{
int targetx,targety,i=0,j,temp,x,y,nodes2=0;
char str[80];
targetx=target/n;
targety=target%n;
temp=final;
path[i++]=target;
while(temp>=0)
{
if((grid[targetx+1][targety].distance==temp)&&((targetx+1)<m))
{nodes2++;
path[i++]=grid[targetx+1][targety].vertexno;
targetx+=1;
temp-=1;
continue;
}
if((grid[targetx][targety+1].distance==temp)&&((targety+1)<n))
{
nodes2++;
path[i++]=grid[targetx][targety+1].vertexno;
targety+=1;
temp-=1;
continue;
}
if(grid[targetx-1][targety].distance==temp && targetx-1>=0)
{
nodes2++;
path[i++]=grid[targetx-1][targety].vertexno;
targetx-=1;
temp-=1;
continue;
}
if((grid[targetx][targety-1].distance==temp)&&((targety-1)>=0))
{
nodes2++;
path[i++]=grid[targetx][targety-1].vertexno;
targety-=1;
temp-=1;
continue;
}
}
pathlength=final+2;
printf("Following is the path from source node to target node\n");
for(i=pathlength-1;i>0;i--)
printf("%d-->",path[i]);
printf("%d\n",path[i]);
getch();
```

```
detectgraph(&gd,&gm);
initgraph(&gd,&gm,"..\\bgi");
setfillstyle(SOLID_FILL,WHITE);
for(i=0;i<m;i++)
for(j=0;j<n;j++)
{
if(grid[i][j].state==BLOCKED)
pieslice(50+j*(540/n),50+i*(380/m),0,360,5);
else
circle(50+j*(540/n),50+i*(380/m),5);
if(grid[i][j].distance!=-1)
outtextxy(40+j*(540/n),30+i*(380/m),itoa(grid[i][j].distance,str,10));
}
formgrid();
setcolor(RED);
setlinestyle(SOLID_LINE,1,3);
x=path[pathlength-1]/n;
y=path[pathlength-1]%n;
moveto(50+y*(540/n),50+x*(380/m));
for(i=pathlength-2;i>=0;i--)
{
x=path[i]/n;
y=path[i]%n;
delay(500);
lineto(50+y*(540/n),50+x*(380/m));
}
setcolor(WHITE);
fprintf(fp,"\n The number of nodes traversed during exploration phase is:%d",nodes1);
fprintf(fp,"\n The number of nodes traversed during retrace phase is:%d",nodes2);
fprintf(fp,"\n The pathlength is:%d",pathlength-1);
outtextxy(0,440,"The number of nodes traversed during exploration phase is:");
outtextxy(600,440,itoa(nodes1,str,10));
outtextxy(0,450,"The number of nodes traversed during retrace phase is:");
outtextxy(600,450,itoa(nodes2,str,10));
outtextxy(0,460,"The pathlength is:");
outtextxy(600,460,itoa(pathlength-1,str,10));
getch();
closegraph();
}
void formgrid(void)
{
int i,j;
setlinestyle(DOTTED_LINE,1,0);
for(i=0;i<m;i++)
{
moveto(50,50+i*(380/m));
for(j=0;j<n;j++)
lineto(50+j*(540/n),50+i*(380/m));
}
for(j=0;j<n;j++)
{
moveto(50+j*(540/n),50);
for(i=0;i<m;i++)
lineto(50+j*(540/n),50+i*(380/m));
}
}
```

souk.cpp

```
#include<dos.h>
#include<conio.h>
#include<stdio.h>
#include<graphics.h>
#include<stdlib.h>
#include<values.h>
#define BLOCKED 1
#define UNBLOCKED 0
#define TRUE 1
#define FALSE 0

int path[100],pathlength;
typedef struct
{
int vertexno;
int state;
int distance;
}vertex;

struct list
{
vertex info;
list *next;
};

typedef list *listptr;
void push(vertex,listptr*);
void pop(listptr*);
void retrace(vertex[50][50],int);
void formgrid(void);
int final,m,n,gd,gm,nodes1=0;
FILE *fp;
main(int argc,char *argv[])
{
int i,j,source,target,blocked,errorcode,sourcex,sourcey,noblocked;
int temp,path_exists,targetx,targety,dirflag=1,samedirflag;
char *str;
listptr plist,nlist,ptr;
vertex grid[50][50];
clrscr();
fp=fopen("usercom.txt","a");
printf("Enter no. of rows:");
scanf("%d",&m);
printf("Enter no. of columns:");
scanf("%d",&n);
for(i=0;i<m;i++)
for(j=0;j<n;j++)
{
grid[i][j].vertexno=i*n+j;
grid[i][j].state=UNBLOCKED;
grid[i][j].distance=-1;
}
printf("Enter source pin:");
scanf("%d",&source);
```

```
printf("Enter target pin:");
scanf("%d",&target);
printf("Enter no. of Blocked pins:");
scanf("%d",&noblocked);
for(i=0;i<noblocked;i++)
{
printf("Enter %d Blocked pin:",i+1);
scanf("%d",&blocked);
grid[blocked/n][blocked%n].state=BLOCKED;
}
fprintf(fp,"\n\n\n SOUKUP'S ALGORITHM\n");
fprintf(fp,"rows:%d\t",m);
fprintf(fp,"columns:%d\n",n);
fprintf(fp,"source vertex number:%d\t",source);
fprintf(fp,"target vertex number:%d\n",target);
fprintf(fp,"vertices number which are blocked:\n");
for(i=0;i<noblocked;i++)
fprintf(fp,"%s\t",argv[6+i]);
sourcex=source/n;
sourcey=source%n;
targetx=target/n;
targety=target%n;
grid[sourcex][sourcey].distance=0;
plist=(listptr)(malloc(sizeof(list)));
plist->info=grid[sourcex][sourcey];
plist->next=NULL;
nlist=NULL;
temp=1;
path_exists=FALSE;
while(plist!=NULL)
{
while(plist!=NULL)
{
samedirflag=0;
sourcex=(plist->info).vertexno/n;
sourcey=(plist->info).vertexno%n;
if
(((sourcex*n+sourcey)!=source)||(((sourcex*n+sourcey)==source)&&(dirflag==1)))
{
dirflag=0;
while((targetx<sourcex)&&(grid[sourcex-1][sourcey].state==UNBLOCKED)&&(grid[sourcex-
1][sourcey].distance==-1))
{
nodes1++;
samedirflag=1;
grid[sourcex-1][sourcey].distance=temp;
if(grid[sourcex-1][sourcey].vertexno==target)
{
path_exists=TRUE;
goto jump;
}
temp++;
push(grid[sourcex-1][sourcey],&plist);
sourcex-=1;
}
if(samedirflag==1)continue;
```

```
while((targety<sourcey)&&(grid[sourcex][sourcey-1].state==UNBLOCKED)&&(grid[sourcex][sourcey-
1].distance==-1))
{
nodes1++;
samedirflag=1;
grid[sourcex][sourcey-1].distance=temp;
if(grid[sourcex][sourcey-1].vertexno==target)
{
path_exists=TRUE;
goto jump;
}
temp++;
push(grid[sourcex][sourcey-1],&plist);
sourcey-=1;
}
if(samedirflag==1)continue;
while
((targetx>sourcex)&&(grid[sourcex+1][sourcey].state==UNBLOCKED)&&(grid[sourcex+1][sourcey].dist
ance==-1))
{
nodes1++;
samedirflag=1;
grid[sourcex+1][sourcey].distance=temp;
if(grid[sourcex+1][sourcey].vertexno==target)
{
path_exists=TRUE;
goto jump;
}
temp++;
push(grid[sourcex+1][sourcey],&plist);
sourcex+=1;
}
if(samedirflag==1)continue;
while
((targety>sourcey)&&(grid[sourcex][sourcey+1].state==UNBLOCKED)&&(grid[sourcex][sourcey+1].dist
ance==-1))
{nodes1++;
samedirflag=1;
grid[sourcex][sourcey+1].distance=temp;
if(grid[sourcex][sourcey+1].vertexno==target)
{
path_exists=TRUE;
goto jump;
}
temp++;
push(grid[sourcex][sourcey+1],&plist);
sourcey+=1;
}
if(samedirflag==1)continue;
}
if((grid[sourcex-1][sourcey].state==UNBLOCKED)&&((sourcex-1)>=0)&&(grid[sourcex-
1][sourcey].distance==-1))
{
nodes1++;
grid[sourcex-1][sourcey].distance=temp;
temp++;
```

```
push(grid[sourcex-1][sourcey],&nlist);
if(grid[sourcex-1][sourcey].vertexno==target)
{
path_exists=TRUE;
goto jump;
}
}
if((grid[sourcex][sourcey-1].state==UNBLOCKED)&&((sourcey-1)>=0)&&(grid[sourcex][sourcey-
1].distance==-1))
{
nodes1++;
grid[sourcex][sourcey-1].distance=temp;
temp++;
push(grid[sourcex][sourcey-1],&nlist);
if(grid[sourcex][sourcey-1].vertexno==target)
{
path_exists=TRUE;
goto jump;
}
}
if
(grid[sourcex+1][sourcey].state==UNBLOCKED&&((sourcex+1)<m)&&(grid[sourcex+1][sourcey].distan
ce==-1))
{
nodes1++;
grid[sourcex+1][sourcey].distance=temp;
temp++;
push(grid[sourcex+1][sourcey],&nlist);
if(grid[sourcex+1][sourcey].vertexno==target)
{
path_exists=TRUE;
goto jump;
}
}
if
(grid[sourcex][sourcey+1].state==UNBLOCKED                                                  &&
((sourcey+1)<n)&&(grid[sourcex][sourcey+1].distance==-1))
{
nodes1++;
grid[sourcex][sourcey+1].distance=temp;
temp++;
push(grid[sourcex][sourcey+1],&nlist);
if(grid[sourcex][sourcey+1].vertexno==target)
{
path_exists=TRUE;
goto jump;
}
}
pop(&plist);
}
plist=nlist;
nlist=NULL;
}
jump:if(path_exists==TRUE)
{
printf("path exists\n");
```

```
final=temp-1;
retrace(grid,target);
}
else
{
printf("path does not exist\n");
getch();
}
return 0;
}
void push(vertex v,listptr *p)
{
listptr temp,ptr;
if(*p==NULL)
{
ptr=(listptr)(malloc(sizeof(list)));
ptr->info=v;
ptr->next=NULL;
*p=ptr;
}
else
{
ptr=(listptr)(malloc(sizeof(list)));
ptr->info=v;
ptr->next=*p;
*p=ptr;
}
}
void pop(listptr *ptr)
{
if((*ptr)->next==NULL)
*ptr=NULL;
else
*ptr=(*ptr)->next;
}
#define TOP 1
#define LEFT 2
#define BOTTOM 3
#define RIGHT 4
void retrace(vertex grid[50][50],int target)
{
int targetx,targety,i=0,j,temp,min=MAXINT,minflag=TOP,x,y,nds2=0;
char str[80];
targetx=target/n;
targety=target%n;
temp=final;
path[i++]=target;
while(temp>0)
{
if((grid[targetx-1][targety].distance!=-1)&&((targetx-1)>=0))
{
nds2++;
min=grid[targetx-1][targety].distance;
minflag=TOP;
}
if((grid[targetx][targety-1].distance!=-1)&&((targety-1)>=0))
```

```
{
nds2++;
if(grid[targetx][targety-1].distance<min)
{
min=grid[targetx][targety-1].distance;
minflag=LEFT;
}
}
if((grid[targetx+1][targety].distance!=-1)&&((targetx+1)<m))
{
nds2++;
if(grid[targetx+1][targety].distance<min)
{
min=grid[targetx+1][targety].distance;
minflag=BOTTOM;
}
}
if((grid[targetx][targety+1].distance!=-1)&&((targety+1)<n))
{
nds2++;
if(grid[targetx][targety+1].distance<min)
{
min=grid[targetx][targety+1].distance;
}
}
temp=min;
switch(minflag)
{
case TOP:
targetx-=1;
target=targetx*n+targety;
path[i++]=target;
break;
case LEFT:
targety-=1;
target=targetx*n+targety;
path[i++]=target;
break;
case BOTTOM:
targetx+=1;
target=targetx*n+targety;
path[i++]=target;
break;
case RIGHT:
targety+=1;
target=targetx*n+targety;
path[i++]=target;
break;
}
}
pathlength=i--;
printf("Following is the path from source node to target node\n");
for(i=pathlength-1;i>0;i--)
printf("%d-->",path[i]);
printf("%d\n",path[i]);
getch();
```

```
detectgraph(&gd,&gm);
initgraph(&gd,&gm,"..\\bgi");
setfillstyle(SOLID_FILL,WHITE);
for(i=0;i<=m;i++)
for(j=0;j<=n;j++)
{
if(grid[i][j].state == BLOCKED)
pieslice(50+j*(540/n),50+i*(380/m),0,360,5);
else
circle(50+j*(540/n),50+i*(380/m),5);
if(grid[i][j].distance != -1)
outtextxy(40+j*(540/n),30+i*(380/m),itoa(grid[i][j].distance,str,10));
}
formgrid();
setcolor(RED);
setlinestyle(SOLID_LINE,1,3);
x = path[pathlength-1]/n;
y = path[pathlength-1]%n;
moveto(50+y*(540/n),50+x*(380/m));
for(i = pathlength-2;i>=0;i--)
{
x = path[i]/n;
y = path[i]%n;
delay(500);
lineto(50+y*(540/n),50+x*(380/m));
}
setcolor(WHITE);
fprintf(fp,"\n The number of nodes traversed during exploration phase is :%d",nodes1);
fprintf(fp,"\n The number of nodes traversed during retrace phase is :%d",nds2);
fprintf(fp,"\n The pathlength is:%d",pathlength-1);
outtextxy(0,440,"The number of nodes traversed during exploration phase is;");
outtextxy(600,440,itoa(nodes1,str,10));
outtextxy(0,450,"The number of nodes traversed during retrace phase is;");
outtextxy(600,450,itoa(nds2,str,10));
outtextxy(0,460,"The pathlength is:");
outtextxy(600,460,itoa(pathlength-1,str,10));
getch();
closegraph();
}
void formgrid(void)
{
int i,j;
setlinestyle(DOTTED_LINE,1,0);
for(i=0;i<m;i++)
{
moveto(50,50+i*(380/m));
for(j=0;j<n;j++)
lineto(50+j*(540/n),50+i*(380/m));
}
for(j=0;j<n;j++)
{
moveto(50+j*(540/n),50);
for(i=0;i<m;i++)
lineto(50+j*(540/n),50+i*(380/m));
}
}
```

```
import java.awt.Color;

/** this class generates a sequence of contrasting colors - useful for
   highlighting different parts of a diagram with different colors
 */
class ColorSequencer {

  /** returns the current color in the sequence */
  public Color current() { return curColor; }

  /** change the color to a contrasting color */

 public Color next() {
    int r = curColor.getRed();
    if (r == 0) r = 1;
    int g = curColor.getGreen();
    if (g == 0) g = 1;
    int b = curColor.getBlue();
    if (b == 0) b = 1;
    // System.out.println("Old color: r=" + r + " g=" + g + " b=" + b);
    int newr = (int)(Math.random() * 43 * b) % 256;
    int newg = r;
    int newb = g;
    if (newr < 100 && newg < 100 && newb < 100) {
      if (Math.random() < 0.5) newg += 100;
      else newb += 100;
    }
    // System.out.println("New color: r=" + newr + " g=" + newg + " b=" + newb);
    return (curColor = new Color(newr,newg,newb));
  }

  private Color curColor = new Color(130,251,23);
}
```

```
import java.awt.*;
import java.awt.image.*;
import java.awt.event.*;


// Grid - panel to display grid for maze routing
//
// Layout:
//                  +--------------------------+
//                  | Layer 1                  |
//                  +--------------------------+
//                  |  |  | |  |  |  |  |  |
//                  +--------------------------+
//                  |  |  | |  |  |  |  |  |
//                  +--------------------------+
//                  | Layer 2                  |
//                  +--------------------------+
//                  |  |  | |  |  |  |  |  |
//                  +--------------------------+
//                  |  |  | |  |  |  |  |  |
//                  +--------------------------+
//                           (additional layers)
//                  +--------------------------+
//                  | Message Area             |
//                  +--------------------------+

class Grid extends Panel {

  private String msg = null;

  private boolean displayParallelMode = false;

  public void setParallelExpand() { displayParallelMode = true; }
  public void setSerialExpand() { displayParallelMode = false; }

  Image myOffScreenImage = null;
  Graphics myOffScreenGraphics = null;

  public Grid(int w, int h, int d) {
    gridArray = new GridPoint[w][h][d];
    for (int i = 0; i < w; i++ )
      for (int j = 0; j < h; j++)
        for (int k = 0; k < d; k++)
          gridArray[i][j][k] = new GridPoint(i,j,k);

    GridPoint.myGrid = this;
    setSize(pixelWidth(), pixelHeight());
    addMouseListener(new MouseAdapter()
    { public void mouseClicked(MouseEvent m) {
        //System.out.println("mouseEvent: " + m);
        handleMouseClick(m.getX(), m.getY());
      }
    } );

  }
```

```
public void setMessage(String s) {
  msg = s;
  redrawGrid();
}

public void flash(GridPoint gp) throws InterruptedException {
  for (int i=0;i<3;i++) {
    gp.highlight(true);
    redrawGrid();
    gridDelay(4);
    gp.highlight(false);
    redrawGrid();
    gridDelay(4);
  }
}

public void redrawGrid() {
  if (myOffScreenImage == null) { // this doesn't work in constructor
    myOffScreenImage = createImage(getSize().width, getSize().height );
    myOffScreenGraphics = myOffScreenImage.getGraphics();
   }
  repaint();
}

public int width() { return gridArray.length; }

public int height() { return gridArray[0].length; }

public int depth() { return gridArray[0][0].length; }

public int pixelWidth() { return width() * Grid.GRIDSIZE; }

public int pixelHeight() { return depth() * (height() + 1) * Grid.GRIDSIZE; }

public static int calculateCols(int pixelWidth) { return (pixelWidth - 2) / GRIDSIZE; }

public static int calculateRows(int pixelHeight, int nLayers) {
  int rawCols = ((pixelHeight - 30) / GRIDSIZE) - 2; // adjust for msg, clr btn
  return rawCols / nLayers - 1; // adjust  -1 for layer labels
}

public GridPoint gridPointAt(int x, int y, int z) {
  if (x < 0 || x >= width() ) return null;
  if (y < 0 || y >= height() ) return null;
  if (z < 0 || z >= depth() ) return null;
  return gridArray[x][y][z];
}

public void reset() {
  for (int i = 0; i < width(); i++ )
    for (int j = 0; j < height(); j++)
      for (int k = 0; k < depth(); k++) {
        GridPoint gp = gridPointAt(i,j,k);
        if (!gp.isObstacle()) gp.reset();
      }
}
```

```
  public void clear() {
    for (int i = 0; i < width(); i++ )
      for (int j = 0; j < height(); j++)
        for (int k = 0; k < depth(); k++) {
          GridPoint gp = gridPointAt(i,j,k);
          gp.reset();
        }
  }
    private class GridLink {
      public GridLink(GridPoint gp) {
        wot = gp;
        next = null;
      }
      private GridLink next;
      private GridPoint wot;
    }

  private GridLink gridPointHead;
  private GridLink gridPointTail;

  void printGridPointQueue() { // for debugging - package visible
    for (GridLink gl = gridPointHead; gl != null; gl = gl.next) {
      System.out.print(gl.wot + " ");
    }
    System.out.println();
  }

  public void enqueueGridPoint(GridPoint gp) throws InterruptedException {
    // if (gp.isEnqueued()) return;  // already there!
    if (gridPointHead == null) gridPointHead = gridPointTail = new GridLink(gp);
    else {
      GridLink gl = new GridLink(gp);
      gridPointTail.next = gl;
      gridPointTail = gl;
    }
    gp.setEnqueued(true);
   if (!displayParallelMode) {
      redrawGrid();
      gridDelay();
    }
  }

  public GridPoint dequeueGridPoint() {
    if (gridPointHead == null) return null;
    else {
      GridPoint gp = gridPointHead.wot;
      gridPointHead = gridPointHead.next;
      gp.setEnqueued(false);
      // debug
//      System.out.println("GridPoint.dequeuePoint - " + gp);
      return gp;
    }
  }

  public void clearQueue() {
```

```
  gridPointHead = null;
}

private static final int DELAY = 100; // 10ms = 0.1s

public static void gridDelay(int d) throws InterruptedException {
    Thread.sleep(d * DELAY);
}

public static void gridDelay() throws InterruptedException {
  gridDelay(1);
}

public int expansion() throws InterruptedException {
  GridPoint gp;
  int actualLength;
  int curVal = 0;
  setMessage("Expansion phase");
  gridDelay(3);
  if (src != null && tgt != null ) {
    src.initExpand();
    if ((actualLength = src.expand()) > 0) {
      clearQueue();
      return actualLength; // found it right away!
    }
    while ((gp = dequeueGridPoint()) != null) {
      if (displayParallelMode && (gp.getVal() > curVal)) {
        curVal = gp.getVal();
        redrawGrid();
        gridDelay(2);
      }
      if ((actualLength = gp.expand()) > 0) {
        clearQueue();
        return actualLength;  // found it!
      }
    }
  }
  return -1;
}

public void traceBack() throws InterruptedException {
  // start at target, then work back
  GridPoint current = tgt;
  while (!current.isSource()) {
    GridPoint next;
    int curval = current.getVal();
    setMessage("Traceback: distance = " + curval);
    current.setRouted();
    redrawGrid();
    gridDelay(3);
    next = current.westNeighbor();
    if (next != null && !next.isObstacle() && next.getVal() < curval) {
      current = next;
      continue;
    }
    next = current.eastNeighbor();
```

```
      if (next != null && !next.isObstacle() && next.getVal() < curval) {
        current = next;
        continue;
      }
      next = current.southNeighbor();
      if (next != null && !next.isObstacle() && next.getVal() < curval) {
        current = next;
        continue;
      }
      next = current.northNeighbor();
      if (next != null && !next.isObstacle() && next.getVal() < curval) {
        current = next;
        continue;
      }
      next = current.upNeighbor();
      if (next != null && !next.isObstacle() && next.getVal() < curval) {
        current = next;
        continue;
      }
      next = current.downNeighbor();
      if (next != null && !next.isObstacle() && next.getVal() < curval) {
        current = next;
        continue;
      }
      System.out.println("AWK! can't trace back! current= " + current);
      break;
    }
    if (current.isSource()) {
      setMessage("Traceback complete");
      flash(current);
      current.setRouted();
    } else System.out.println("Warning: traceBack failed!");
  }

  // the following are used in drawing into the grid panel and are package visible

  static final int GRIDSIZE = 19;
  static final int CHARXOFFSET = 6;
  static final int CHARYOFFSET = 14;

  int gridPanelX(int i, int j, int k) { return i * GRIDSIZE;}

  int gridPanelY(int i, int j, int k) {
    return(k * (GRIDSIZE * (height() + 1) )) + ((j+1) * GRIDSIZE);
  }

  int gridLyrY(int layer) { return (GRIDSIZE * layer * (height()+1) + CHARYOFFSET); }

  int gridMsgY() { return (GRIDSIZE * depth() * (height() + 1)) + CHARYOFFSET; }

  private GridPoint clickedPoint = null;
  private boolean clearPending = false;

  public synchronized void handleMouseClick(int x, int y) {
    clickedPoint = mouseToGridPoint(x, y);
    notifyAll();
    //System.out.println("handleMouseClick: " + clickedPoint);
  }

  public synchronized void requestClear() {
    clearPending = true;
    notifyAll();
  }

  GridPoint mouseToGridPoint(int x, int y) {
    int i, j, k;
    i = x / GRIDSIZE;
    for (k = 0; k < depth(); k++) {
      if (y >= gridPanelY(0,0,k) && y <= gridPanelY(0,0,k+1)-GRIDSIZE) {
        j = (y % ((height() + 1) * GRIDSIZE) / GRIDSIZE) - 1;
        //System.out.println("mouseToGridPoint: found GridPoint at ("
         //    + i + "," + j + "," + k + ") " + gridPointAt(i,j,k));
        if (i >= 0 && i < width() && j >= 0 && j < height() && k >= 0 && k < depth())
          return gridPointAt(i,j,k);
        else return null;
      }
    }
    return null;
  }


  private static final int WAITFORSRC = 0;
  private static final int WAITFORTGT = 1;

  private synchronized void waitForInput() throws InterruptedException {
    while (!clearPending && clickedPoint == null) wait();
  }

  public void run() throws InterruptedException {
    clear();
    int state = WAITFORSRC;
    setMessage("Click on Source");
    while (true) {
      waitForInput();
      if (state == WAITFORSRC) {
        if (clearPending) {
          clear();
          setMessage("Grid cleared!");
          gridDelay(5);
          setMessage("Click on Source");
          clearPending = false;
        } else if (clickedPoint != null) {
          if (!clickedPoint.isRouted()) {
            setSource(clickedPoint);
            System.out.println("Setting source: " + clickedPoint);
            state = WAITFORTGT;
            setMessage("Click on Target");
          } else {
            setMessage("Already routed!");
            gridDelay(5);
            setMessage("Click on Source");
          }
          clickedPoint = null;
        }
      } else if (state == WAITFORTGT) {
        if (clickedPoint != null) {
          setTarget(clickedPoint);
          System.out.println("Setting target: " + clickedPoint);
          clickedPoint = null;
          setMessage("Ready to Route!");
          gridDelay(5);
          redrawGrid();
          route();
          src = null;
          tgt = null;
          state = WAITFORSRC;
          setMessage("Click on Source");
          redrawGrid();
        }
        clearPending = false;
      }
    }
  }

  public void paint(Graphics g) {
    g.setColor(getBackground());
    g.fillRect(0,0,getSize().width,getSize().height);
    for (int k = 0; k < depth(); k++ ) {
      g.setColor(Color.black);
      g.drawString("Layer " + (k+1), CHARXOFFSET, gridLyrY(k));
      for (int j = 0; j < height(); j++)
        for (int i = 0; i < width(); i++)
          gridArray[i][j][k].paintGridPoint(g);
    }
    if (msg != null) {
      g.setColor(Color.black);
      g.drawString(msg, CHARXOFFSET, gridMsgY());
    }
  }

  public void update(Graphics g) {
    paint(myOffScreenGraphics);
    g.drawImage(myOffScreenImage,0,0,this);
  }

  public void setSource(int x, int y, int z) {
    setSource(gridArray[x][y][z]);
  }

  public void setSource(GridPoint s) {
    src = s;
  }

  public void setTarget(int x, int y, int z) {
    tgt = gridArray[x][y][z];
  }

  public void setTarget(GridPoint t) { tgt = t; }
```

```
  public GridPoint getSource() { return src; }

  public GridPoint getTarget() { return tgt; }

  public int route() throws InterruptedException {
    if (src == null || tgt == null) return -1;
    GridPoint.nextRouteColor();
    reset();
    if (src == tgt) {  // trivial case
      src.setRouted();
      return 0;
    } else {
      int actualLength = expansion();
      clearQueue();
      redrawGrid();
      if (actualLength > 0) {
        setMessage("Target Found!");
        flash(tgt);
        traceBack();
      } else setMessage("Target not found!");
      reset();
      redrawGrid();
      return actualLength;
    }
  }

  public int route(GridPoint s, GridPoint t) throws InterruptedException {
    setSource(s);
    setTarget(t);
    return route();
  }

  public int route(int sx, int sy, int sz, int dx, int dy, int dz) throws InterruptedException {
    return route(gridPointAt(sx,sy,sz),gridPointAt(dx,dy,dz));
  }

  GridPoint [][][] gridArray;
  GridPoint src;
  GridPoint tgt;

}
```

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;

class GridPoint {

  public GridPoint(int x, int y, int z) {
    posx = x;
    posy = y;
    posz = z;
  }

  public GridPoint(int x, int y) {
    this(x, y, 0);
  }

  /** return the north neighbor of this point.  returns null if this
      point is on the northern edge of the grid */
  public GridPoint northNeighbor() {
    return myGrid.gridPointAt(posx, posy-1, posz);
  }

  /** return the south neighbor of this point.  returns null if this
      point is on the southern edge of the grid */
  public GridPoint southNeighbor() {
    return myGrid.gridPointAt(posx, posy+1, posz);
  }

  /** return the east neighbor of this point.  returns null if this
      point is on the eastern edge of the grid */
  public GridPoint eastNeighbor() {
    return myGrid.gridPointAt(posx+1, posy, posz);
  }

  /** return the west neighbor of this point.  returns null if this
      point is on the western edge of the grid */
  public GridPoint westNeighbor() {
    return myGrid.gridPointAt(posx-1, posy, posz);
  }

  /** return the neighbor on the next level up.  Returns null if there
      is no layer above current layer */
  public GridPoint upNeighbor() {
    return myGrid.gridPointAt(posx, posy, posz-1);
  }

  /** return the neighbor on the next level down.  Returns null if there
      is no layer below current layer */
  public GridPoint downNeighbor() {
    return myGrid.gridPointAt(posx, posy, posz+1);
  }

  /* expand a routing search */
  public int expand() throws InterruptedException {
    GridPoint xp;
    if ( (xp = westNeighbor()) != null && xp.val == UNROUTED ) {
```

```
    xp.val = val+1;
    if (xp.isTarget()) return xp.val;
    else myGrid.enqueueGridPoint(xp);
  }
  if ( (xp = eastNeighbor()) != null && xp.val == UNROUTED ) {
    xp.val = val+1;
    if (xp.isTarget()) return xp.val;
    else myGrid.enqueueGridPoint(xp);
  }
  if ( (xp = southNeighbor()) != null && xp.val == UNROUTED ) {
    xp.val = val+1;
    if (xp.isTarget()) return xp.val;
    else myGrid.enqueueGridPoint(xp);
  }
  if ( (xp = northNeighbor()) != null && xp.val == UNROUTED ) {
    xp.val = val+1;
    if (xp.isTarget()) return xp.val;
    else myGrid.enqueueGridPoint(xp);
  }
  if ( (xp = upNeighbor()) != null && xp.val == UNROUTED ) {
    xp.val = val+1;
    if (xp.isTarget()) return xp.val;
    else myGrid.enqueueGridPoint(xp);
  }
  if ( (xp = downNeighbor()) != null && xp.val == UNROUTED ) {
    xp.val = val+1;
    if (xp.isTarget()) return xp.val;
    else myGrid.enqueueGridPoint(xp);
  }
  return -1;
}


public boolean isTarget() {
  return ( this == myGrid.getTarget() );
}

public boolean isSource() {
  return ( this == myGrid.getSource() );
}

public boolean isEmpty() {
  return val == UNROUTED;
}

public void paintGridPoint(Graphics g) {
  g.setColor(Color.black);
  g.drawRect(myGrid.gridPanelX(posx,posy,posz),myGrid.gridPanelY(posx,posy,posz),
      myGrid.GRIDSIZE,myGrid.GRIDSIZE);
  if (isRouted()) {
    g.setColor(routedColor);
    fillGridPoint(g);
  } else if (isObstacle()) {
    g.setColor(Color.red);
    fillGridPoint(g);
  }
  else if (isSource()) {
    if (highlighted) g.setColor(Color.yellow);
    else g.setColor(Color.red);
    fillGridPoint(g);
    g.setColor(Color.black);
    labelGridPoint(g, "S");
  } else if (isTarget()) {
    if (highlighted) {
      g.setColor(Color.yellow);
      fillGridPoint(g);
      g.setColor(Color.black);
      labelGridPoint(g, Integer.toString(val % 10));
    } else {
      g.setColor(Color.red);
      fillGridPoint(g);
      g.setColor(Color.black);
      labelGridPoint(g, "T");
    }
  } else if (val < UNROUTED) {
    if (isEnqueued()) g.setColor(Color.orange);
    else g.setColor(Color.yellow);
    fillGridPoint(g);
    g.setColor(Color.black);
    labelGridPoint(g, Integer.toString(val % 10));
  }
}

private void fillGridPoint(Graphics g) {
  g.fillRect(myGrid.gridPanelX(posx,posy,posz)+2,myGrid.gridPanelY(posx,posy,posz)+2,
      myGrid.GRIDSIZE-3,myGrid.GRIDSIZE-3);
}
private void labelGridPoint(Graphics g, String s) {
  g.drawString(s, myGrid.gridPanelX(posx,posy,posz)+myGrid.CHARXOFFSET,
                myGrid.gridPanelY(posx,posy,posz)+myGrid.CHARYOFFSET);
}

public String toString() {
  return "GridPoint(" + posx + "," + posy  + "," + posz + ")[" + val + "]";
}

public void reset() { val = UNROUTED; }

public void initExpand() { val = 0; }

public void setRouted() {
  val = ROUTED;
  routedColor = cseq.current();
}

public boolean isRouted() { return (val == ROUTED); }

public void setObstacle() { val = OBSTACLE; }

public boolean isObstacle() { return (val == ROUTED || val == OBSTACLE); }

public boolean lessThan(GridPoint p2) { return val != 0 && val < p2.val; }
```

```
  private static final int ROUTED = -2;
  private static final int OBSTACLE = -1;
  private static final int UNROUTED = 9999;

  public int manhattanDistance(GridPoint p2) {
    return ( Math.abs(p2.posx - posx) + Math.abs(p2.posy - posy) +
      Math.abs(p2.posz - posz) );
  }

  private int posx;
  private int posy;
  private int posz;
  private boolean highlighted = false;
  private boolean enqueued = false;
  boolean isEnqueued()  { return enqueued; }
  void setEnqueued(boolean e) { enqueued = e; }

  private int val = UNROUTED;

  private Color routedColor = null;

  public int getVal() { return val; }

  public void highlight(boolean h) { highlighted = h; }

  static Grid myGrid;

  private static ColorSequencer cseq = new ColorSequencer();

  public static void nextRouteColor() { cseq.next(); }
}
```

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
import javax.swing.*;

/**
 * Title:       MazeApplet
 * Description:  Animation of Lee Algorithm for maze routing.
 * Copyright:    Copyright (c) 2001
 * Company:      Lafayette College
 * @author John A. Nestor
 * @version 1.0
 */

public class MazeApplet extends Applet implements Runnable, ActionListener
{
  private Grid myGrid = null;

  private Thread myThread = null;

  public void init() {

      String lstr = getParameter("layers");
      int nlayers;
      if (lstr != null) nlayers = Integer.parseInt(lstr);
      else nlayers = 1;
      int ncols = Grid.calculateCols(getSize().width);
      int nrows = Grid.calculateRows(getSize().height, nlayers);
      myGrid = new Grid(ncols,nrows,nlayers);
      String modestr = getParameter("mode");
      if (modestr != null && modestr.equals("parallel")) myGrid.setParallelExpand();
      else myGrid.setSerialExpand();
      setLayout(new BorderLayout());
      add(myGrid, "Center");
      Button clearBtn = new Button("CLEAR");
      Panel btnPanel = new Panel();
      btnPanel.setLayout(new FlowLayout(FlowLayout.CENTER));
      btnPanel.add(clearBtn);
      add(btnPanel, "South");
      clearBtn.addActionListener(this);
      show();
      repaint();
      if (myThread == null) {
        myThread = new Thread(this);
        myThread.start();
      }
   }

   public void start() {
      if (myThread == null) {
        myThread = new Thread(this);
        myThread.start();
      }
   }

   public void stop() {
    myThread.interrupt();
    myThread = null;
  }

  public void actionPerformed(ActionEvent e) {
    myGrid.requestClear();
  }

  private static final int WAITFORSRC = 0;
  private static final int WAITFORTGT = 1;

  private GridPoint clicked = null;

  public void run() {
    try {
      myGrid.run();
    } catch (InterruptedException e) {}
  }
}
```

```
//////////////////////////////////////////////////////////////////////////
// "Tree.java" - Demonstrates various types of minimum-length
// interconnection trees (minimum spanning tree, Steiner tree, etc.)
//
// Author: Joe Ganley, mst.10.ganley@spamgourmet.com
//////////////////////////////////////////////////////////////////////////



import java.lang.*;
import java.awt.*;


public class Tree extends java.applet.Applet {

private final int          maxN = 30;              // the max number of terminals
private int                n = 10;                 // the current number of terminals
private final int          r = 4;                  // the radius of a terminal
private Point                          p[];                     // the terminals
private Point                          current;    // the current one and its old location
private boolean                        m[][];                   // the minimum spanning tree edges
private Rectangle          border, inner;          // applet borders
private Scrollbar          sb;                     // scrollbar for changing n
private Image                          buffer;                  // buffer for double-buffering
private Graphics           bufg;                   // buffer's Graphics


public void init() {
  p = new Point[maxN];
  current = null;
  m = new boolean[maxN][maxN];

  border = new Rectangle(0, 0, size().width - 1, size().height - 1);
  inner = new Rectangle(r + 1, r + 1,
                                        size().width - 2 * r - 3, size().height - 2 * r - 23);

  // initialize the terminals to random locations
  for (int i = 0; i < maxN; i++) {
    p[i] = new Point((int) Math.round(Math.random()
                                                   * (size().width - 2 * r - 2) + r + 1),
                               (int) Math.round(Math.random()
                                                   * (size().height - 20 - 2 * r - 2) + r + 1));
    for (int j = 0; j < maxN; j++) {
      m[i][j] = false;
    }
  }
  mst();

  setBackground(Color.white);

  setLayout(new BorderLayout());
  sb = new Scrollbar(Scrollbar.HORIZONTAL, n, 5, 2, maxN);
  add("South", sb);

  buffer = createImage(size().width, size().height);
  bufg = buffer.getGraphics();
  bufg.setFont(getFont());
} // init()


public void update(Graphics g) {
  bufg.setColor(getBackground());
  bufg.fillRect(border.x, border.y, border.width, border.height);
  bufg.setColor(Color.black);
  bufg.drawRect(border.x, border.y, border.width, border.height);

  // first do the MST edges
  for (int i = 0; i < n; i++) {
    for (int j = (i + 1); j < n; j++) {
      if (m[i][j]) {
            bufg.setColor(Color.red);
            bufg.drawLine(p[i].x, p[i].y, p[j].x, p[j].y);
      }
    }
  }

  // redraw the terminals
  for (int i = 0; i < n; i++) {
    bufg.setColor(Color.green);
    bufg.fillOval(p[i].x - r, p[i].y - r, 2 * r, 2 * r);
    bufg.setColor(Color.black);
    bufg.drawOval(p[i].x - r, p[i].y - r, 2 * r, 2 * r);
  }

  // draw the current one in cyan
  if (current != null) {
    bufg.setColor(Color.cyan);
    bufg.fillOval(current.x - r, current.y - r, 2 * r, 2 * r);
    bufg.setColor(Color.black);
    bufg.drawOval(current.x - r, current.y - r, 2 * r, 2 * r);
  }

  g.drawImage(buffer, 0, 0, null);
} // update(Graphics)


public void paint(Graphics g) {
  update(g);
} // paint(Graphics)


public boolean handleEvent(Event evt) {
  switch (evt.id) {

  case Event.MOUSE_DOWN: {
    Rectangle rect = new Rectangle();

    current = null;
    for (int i = 0; (i < n) && (current == null); i++) {
      rect.reshape(p[i].x - r, p[i].y - r, 2 * r, 2 * r);
      if (rect.inside(evt.x, evt.y)) {
            current = p[i];
      }
    }
    break;
  }

  case Event.MOUSE_UP: {
    current = null;
    repaint();
    break;
  }

  case Event.MOUSE_DRAG: {
    if (current != null) {
      if (inner.inside(evt.x, evt.y)) {
              current.move(evt.x, evt.y);
      }
      else {
              current.move(Math.max(Math.min(evt.x, inner.x + inner.width), inner.x),
                           Math.max(Math.min(evt.y, inner.y + inner.height), inner.y));
      }
      mst();
      repaint();
    }
    break;
  }

  case Event.SCROLL_LINE_UP: case Event.SCROLL_LINE_DOWN:
  case Event.SCROLL_PAGE_UP: case Event.SCROLL_PAGE_DOWN:
  case Event.SCROLL_ABSOLUTE: {
    n = sb.getValue();
    mst();
    repaint();
    break;
  }

  default: {
    break;
  }

  } // switch

  return(true);
} // handleEvent(Event)


// Euclidean distance between two points (x1,y1) and (x2,y2)
private int distance(int x1, int y1, int x2, int y2) {
  return((int) Math.round(Math.sqrt(
                             (double) (x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 - y2))));
} // distance(int,int,int,int)


// mst - compute a minimum spanning tree using Prim's algorithm with "dumb"
// heaps.  This is the fastest graph algorithm for complete graphs, though
// we could do better geometrically - but with 10 terminals, why bother?
private void mst() {
  int dist[], neigh[], closest, minDist, d;

  dist = new int[n];
  neigh = new int[n];

  // initialize data structures
  for (int i = 0; i < n; i++) {
    dist[i] = distance(p[0].x, p[0].y, p[i].x, p[i].y);
    neigh[i] = 0;
    for (int j = 0; j < n; j++) {
      m[i][j] = false;
    }
  }

  // find terminal closest to current partial tree
  for (int i = 1; i < n; i++) {
    closest = -1;
    minDist = Integer.MAX_VALUE;
    for (int j = 1; j < n; j++) {
      if ((dist[j] != 0) && (dist[j] < minDist)) {
              closest = j;
              minDist = dist[j];
      }
    }

    // set an edge from it to its nearest neighbor
    m[neigh[closest]][closest] = true;
    m[closest][neigh[closest]] = true;

    // update nearest distances to current partial tree
    for (int j = 1; j < n; j++) {
      d = distance(p[j].x, p[j].y, p[closest].x, p[closest].y);
      if (d < dist[j]) {
              dist[j] = d;
              neigh[j] = closest;
      }
    }
  }
} // mst()

} // class Tree extends java.applet.Applet
```

```
import java.io.*;
import java.awt.*;
import java.applet.*;
import java.util.StringTokenizer;
import java.util.Vector;
import java.util.Enumeration;

/** This program implement's the "Left Edge Algorithm" for channel
   routing.  It includes vertical constraints but cannot deal with cyclic
   vertical constraints.

*/

public class ChannelRouterApplet extends Applet {

  private TextField[] upperFields, lowerFields;
  private Netlist n;
  ChannelRouterCanvas cv;

  public void init() {
    System.out.println("starting applet!!!");
    n = new Netlist();
    setLayout(new BorderLayout());
    Panel up = new Panel();
    Panel lp = new Panel();
    Panel gp = new Panel();
    Panel bp = new Panel();
    cv = new ChannelRouterCanvas(n);
    setLayout(new BorderLayout());
    up.setLayout(new GridLayout(1, Netlist.NUM_COLS));
    lp.setLayout(new GridLayout(1,Netlist.NUM_COLS));
    gp.setLayout(new BorderLayout());
    bp.setLayout(new FlowLayout());
    Button cb = new Button("Clear Nets");
    Button rb = new Button("Route Nets");

    upperFields = new TextField[Netlist.NUM_COLS];
    lowerFields = new TextField[Netlist.NUM_COLS];
    for (int i = 0; i < Netlist.NUM_COLS; i++) {
      TextField tf;
      upperFields[i] = tf = new TextField("", 2);
      up.add(tf);
      lowerFields[i] = tf = new TextField("", 2);
      lp.add("Center", tf);
    }
    gp.add("North", up);
    gp.add("South", lp);
    gp.add("Center", cv);
    bp.add(cb);
    bp.add(rb);
    add("South", bp);
    add("Center", gp);

  }

  /** Clear all net fields */
```

```
private void clearNetFields() {
  for (int i = 0; i < Netlist.NUM_COLS; i++) {
    upperFields[i].setText("");
    lowerFields[i].setText("");
  }
}

/** Scan the terminal fields for each column and build up the netlist data structure */
private void scanFieldsAndRoute() {
  n.clear();
  for (int i = 0; i < Netlist.NUM_COLS; i++) {
    String uname = upperFields[i].getText().trim();
    String lname = lowerFields[i].getText().trim();
    Terminal term;
    Net nt;
    // process upper terminal of column i
    if (!uname.equals("")) {
      nt = n.findNet(uname);
      if (nt == null) {
        nt = new Net(uname);
        n.addNet(nt);
        }
      term = n.getTerminal(i, Terminal.TOP);
      nt.addTerminal(term);
    }
    // now process lower terminal of column i
    if (!lname.equals("")) {
      nt = n.findNet(lname);
      if (nt == null) {
        nt = new Net(lname);
        n.addNet(nt);
      }
      term = n.getTerminal(i, Terminal.BOTTOM);
      nt.addTerminal(term);
    }
  }
  // n.write(System.out);
  n.leftEdgeAlgorithm();
  // n.write(System.out);
}

public boolean handleEvent(Event evt) {
  if (evt.id == Event.WINDOW_DESTROY) System.exit(0);
  return super.handleEvent(evt);
}

public boolean action(Event evt, Object arg) {
  if (arg.equals("Route Nets")) {
    scanFieldsAndRoute();
    cv.repaint();
  } else if (arg.equals("Clear Nets")) {
    clearNetFields();
    scanFieldsAndRoute();  // to clear the old nets away
    cv.repaint();
  } else return super.action(evt, arg);
  return true;
```

```
 }

}

class ChannelRouterCanvas extends Canvas {

  private Netlist nl;

  public ChannelRouterCanvas(Netlist n) { nl = n; }

  private static final int BORDER_Y_OFFSET = 10;
  private static final int HEIGHT_ADJUST = 10;
  private static final int WIDTH_ADJUST = 1;

  private int colWidth;
  private int trackHeight;

  private void setScale() {
    Dimension d = size();
    colWidth = (d.width) / Netlist.NUM_COLS;
    trackHeight = (d.height - HEIGHT_ADJUST - BORDER_Y_OFFSET) / (nl.getMaxTrack() + 1);
  }

  /** @return the Y coordinate for track number. */
  public int trackY(int track) {
    return BORDER_Y_OFFSET + trackHeight * track;
  }
  /** X coordinate of column */
  public int colX(int col) {
    return colWidth/2 + colWidth * col;
  }

  public int channelWidth() {
    return (colWidth * Netlist.NUM_COLS - 1);
  }

  public int channelHeight() {
    return (trackY(nl.getMaxTrack() +1) - trackY(0) + 1);
  }


  private Font f;
  private FontMetrics fm;
  private boolean fontsSet = false;

  private void setFonts(Graphics g) {
    if (fontsSet) return;
    f = new Font("Helvetica", Font.PLAIN, 12);
    fm = g.getFontMetrics(f);
    g.setFont(f);
    fontsSet = true;
  }

  public void paint(Graphics g) {
    setFonts(g);
    setScale();
    g.drawRect(0, BORDER_Y_OFFSET, channelWidth(), channelHeight());
    if (nl.getCyclesPresent()) {
      g.drawString("This circuit contains unresolved vertical constraint cycles",
        20, trackY(1) - 5);
    }
    for (Enumeration e = nl.getNets() ; e.hasMoreElements() ;) {
      Net n = (Net)e.nextElement();
      if (n.getTrack() != Net.TRACK_UNASSIGNED) {
        g.setColor(Color.blue);
        g.fillRect(colX(n.getLeftEdge()), trackY(n.getTrack())-2,
          colX(n.getRightEdge()) - colX(n.getLeftEdge())+2, 5);
        // int xnoff = fm.stringWidth(n.getName()) / 2;
        for (Enumeration eterms = n.getTerminals(); eterms.hasMoreElements() ; ) {
          Terminal t = (Terminal)eterms.nextElement();
          g.setColor(Color.red);
          if (t.getTopOrBottom()) {
            g.fillRect(colX(t.getColumn())-2, trackY(0),
              5, trackY(n.getTrack()) - trackY(0) + 3);
            //g.drawString(n.getName(), colX(t.getColumn()) - xnoff, trackY(0) - 2);
          } else {
            g.fillRect(colX(t.getColumn()) - 2, trackY(n.getTrack())-2,
              5, trackY(nl.getMaxTrack()+1) - trackY(n.getTrack())+4);
            //g.drawString(n.getName(), colX(t.getColumn()) - xnoff, trackY(nl.getMaxTrack()+1) +
fm.getAscent());
          }
          g.setColor(Color.black);
          g.fillRect(colX(t.getColumn())-3, trackY(n.getTrack())-3, 7, 7);
        }
      }
    }
  }

}

class Netlist
{
  private Terminal[] upperTerms;
  private Terminal[] lowerTerms;
  private Vector nlist;
  private int maxTrack = Net.TRACK_UNASSIGNED;
  private boolean cyclesPresent = false;

  public static final int NUM_COLS = 10;

  public Netlist() {
    nlist = new Vector();
    upperTerms = new Terminal[NUM_COLS];
    lowerTerms = new Terminal[NUM_COLS];
    for (int i = 0; i < NUM_COLS; i++) {
      upperTerms[i] = new Terminal(i, Terminal.TOP);
      lowerTerms[i] = new Terminal(i, Terminal.BOTTOM);
    }
  }

  public boolean getCyclesPresent() { return cyclesPresent; }
```

```
/** return terminal object for the given position */
public Terminal getTerminal(int pos, boolean topOrBottom) {
  if (topOrBottom == Terminal.TOP) return upperTerms[pos];
  else return lowerTerms[pos];
}

/** insert net into netList sorted by left edge */
public void addNet(Net n) {
  int i;
  for (i = 0; i < nlist.size(); i++) {
    if (((Net)nlist.elementAt(i)).getLeftEdge() > n.getLeftEdge()) {
      nlist.insertElementAt(n, i);
      return;
    }
  }
  nlist.addElement(n);
}

void unmarkNets() {
  for (Enumeration e = getNets() ; e.hasMoreElements() ; )
    ((Net)e.nextElement()).setMark(false);
}

/** make vertical constraints and check for cycles - if they occur, just remove one constraint
     even though it's wrong. */
public void makeConstraints() {
  for (int i = 0; i < NUM_COLS; i++) {
    Net unet = upperTerms[i].getNet();
    Net lnet = lowerTerms[i].getNet();
    if (unet != null && lnet != null && unet != lnet) {
      //System.out.println("Adding constraint " + unet.getName()
      //  + " above " + lnet.getName());
      lnet.addConstraint(unet);
    }
  }
 // Search for cycles in constraints - just delete a constraint when one is found (for now)
 cyclesPresent = false;
 for (int i = 0; i < NUM_COLS; i++) {
   unmarkNets();
   Net lnet = lowerTerms[i].getNet();
   if (lnet != null && !lnet.checkConstraint()) cyclesPresent = true;
  }
}



public void clear() {
  nlist.removeAllElements();
  for (int i = 0; i < NUM_COLS; i++) {
    upperTerms[i].setNet(null);
    lowerTerms[i].setNet(null);
  }
}

public Enumeration getNets() { return nlist.elements(); }
```

```
  public Net findNet(String findName) {
    for (int i = 0; i < nlist.size(); i++) {
      Net nt = (Net)nlist.elementAt(i);
      if (nt.getName().equals(findName)) return nt;
    }
    return null;
  }

  public void write(PrintStream os) {
    os.println("Writing netlist");
    for (Enumeration e = nlist.elements() ; e.hasMoreElements() ;) {
      ((Net)e.nextElement()).writeNet(os);
   }
  }

  /** determine whether we can assign net testNet to track trk by scanning nets */
  private boolean canRoute(Net testNet, int trk) {
    for (Enumeration ne = nlist.elements(); ne.hasMoreElements() ; ) {
      Net n = (Net)ne.nextElement();
      if (n.getTrack() == trk && testNet.overlap(n)) return false;
    }
    return true;
  }

  /** assign nets to tracks in channel using the Left Edge Algorithm */
  public void leftEdgeAlgorithm() {
    makeConstraints();
    int track = 0;
    boolean done = false;
    while (!done) {
      track++;
      done = true;
      // place unrouted nets in current track if they fit
      for (Enumeration e = nlist.elements(); e.hasMoreElements() ; ) {
        Net n = (Net)e.nextElement();
        if (n.isRouted()) continue;  // skip nets already done
        if (n.testConstraints() && canRoute(n, track)) {
          //System.out.println("Assigning net " + n.getName() +
          //    "(" + n.getLeftEdge() + ":" + n.getRightEdge() +
          //    ") to track" + track);
          n.setTrack(track);
        }
        else done = false;  // at least one net (this one) remains unrouted!
      }
    }
    maxTrack = track;
  }

  public int getMaxTrack() { return maxTrack; }

}

class Net {

  /** Symbolic constant for unrouted net */
  public static final int TRACK_UNASSIGNED = 0;
```

```
 private String name;
 private Vector terms = new Vector(5);
 private Vector constraints = new Vector(2);  // contains vertical constraints (if any)
 private int leftEdge = Integer.MAX_VALUE;
 private int rightEdge = 0;
 private int track = TRACK_UNASSIGNED;
 private boolean mark = false;

 /** Create a new net from a string containing the net name
    */
 public Net(String n) { ;
    name = n;
 }

 /** @return true if net assigned to a track */
 public boolean isRouted() {
   return (getTrack() != Net.TRACK_UNASSIGNED);
 }

 /** Get the assigned track of this net
    * @return              the assigned track of this net
    */
 public int getTrack() { return track; }

 /** Assign this net to a track
     * @param t            The track this net will be assigned to.
     */
 public void setTrack(int t) { track = t; }

 /** get the left edge of this net
    * @return the leftmost column of this net
    */
 public int getLeftEdge() { return leftEdge; }

/** get the right edge of this net
    * @return the rightmost column of this net
    */
 public int getRightEdge() { return rightEdge; }

 public boolean overlap(Net n2) {
   int l1 = getLeftEdge();
   int r1 = getRightEdge();
   int l2 = n2.getLeftEdge();
   int r2 = n2.getRightEdge();
   return (l2 <= r1 && r2 >= l1) || (l1 <= r2 && r1 >= l2);
 }

 /** write out the net */
 public void writeNet(PrintStream os) {
   os.print(name + " ");
   if (getTrack() != TRACK_UNASSIGNED) os.print(track + " ");
   for (Enumeration e = terms.elements(); e.hasMoreElements() ; )
     ((Terminal)e.nextElement()).writeTerminal(os);
   os.println();
 }
```

```
/** @return the name of this net
  */
public String getName() { return name; }

/** Add a terminal to this net
  * @param trm       The terminal to be added to this net.
  */
public void addTerminal(Terminal trm){
  terms.addElement(trm);
  trm.setNet(this);
  if (trm.getColumn() < leftEdge) leftEdge = trm.getColumn();
  if (trm.getColumn() > rightEdge) rightEdge = trm.getColumn();
}

/** @return an enumeration containing the terminals of this net */
public Enumeration getTerminals() {
  return terms.elements();
}

public void addConstraint(Net nt) {
  if (!constraints.contains(nt))
    constraints.addElement(nt);
}

public boolean checkConstraint() {
  if (constraints == null) return true;
  boolean result = true;
  setMark(true);
  for ( int i = 0; i < constraints.size(); i++ ) {
    Net ctgt = (Net)constraints.elementAt(i);
    if (ctgt.getMark()) {
      //System.out.println("Constraint cycle found from Net " + getName() + //debug
      //    " to " + ctgt.getName() + " ... removing");
      constraints.removeElementAt(i);
      result = false;
    } else {
      if (!ctgt.checkConstraint()) result = false;
    }
  }
  setMark(false);
  return result;
}

public Enumeration getConstraints() {
  return constraints.elements();
}

/** Test vertical constraints on this net;  @return true if all nets in constraint are already routed */
public boolean testConstraints() {
  for (Enumeration e = getConstraints(); e.hasMoreElements(); ) {
    Net testnet = (Net)e.nextElement();
    if (!testnet.isRouted()) return false;
  }
  return true;
}
```

```
  public void setMark(boolean m) { mark = m; }

  public boolean getMark() { return mark; }

}


class Terminal {

  private int column;
  private boolean topOrBottom;
  private Net owner;

  /** Symbolic constant for terminals at top of channel */
  public static final boolean TOP = true;
  /** Symbolic constants for terminals at bottom of channel */
  public static final boolean BOTTOM = false;

  public Terminal(int col, boolean tOrB) {
    column = col;
    topOrBottom = tOrB;
  }

  public Terminal(String spec) throws IOException {
    switch (spec.charAt(spec.length() - 1)) {
      case 'u':
      case 'U': topOrBottom = Terminal.TOP;
                break;
      case 'l':
      case 'L': topOrBottom = Terminal.BOTTOM;
                break;
      default:  throw new IOException();
    }
    String s = spec.substring(0, spec.length()-1);
    column = Integer.parseInt(spec.substring(0, spec.length() - 1));
  }

  public int getColumn() { return column; }

  public boolean getTopOrBottom() { return topOrBottom; }

  public Net getNet() { return owner; }

  public void setNet(Net n) { owner = n; }

  public void writeTerminal(PrintStream os) {
    os.print(column);
    if (topOrBottom == Terminal.TOP) os.print("U ");
    else os.print("L ");
  }

};
```

Maskprog.cpp

```
#include<conio.h>
#include<stdio.h>
#include<graphics.h>
#include<stdlib.h>
#define UNUSED 0
#define USED 1
void main()
{
int gd,gm,distance[100],left,right,ind,x,y,max=0;
int a[100],flag[100],i,j,m,n,col,row,inc;
char *str;
clrscr();
printf("enter the number of connections\n");
scanf("%d",&m);
printf("enter their left pin and right pin nos. starting from 0\n");
for(i=0;i<100;i++)
{a[i]=0;distance[i]=-1;}
for(i=1;i<=m;i++)
{scanf("%d%d",&left,&right);
if(right>max)
max=right;
a[left]=a[right]=i;
}
n=10;
for(i=0;i<n;i++)
flag[i]=UNUSED;
col=max+1;
row=n;
for(i=0;i<col;i++)
{
ind=0;
if(a[i]==0)continue;
for(j=0;j<i;j++)
{
if(a[j]==a[i])
{ind=1;
flag[distance[j]]=UNUSED;
distance[i]=distance[j];
break;
}
}
if(ind==1)continue;
for(j=0;j<row;j++)
{
if(flag[j]==UNUSED)
{
distance[i]=j;
flag[j]=USED;
break;
}
}
}
detectgraph(&gd,&gm);
initgraph(&gd,&gm,"..\\bgi");
inc=540/col;
setcolor(WHITE);
for(i=0;i<col;i++)
{x=50+i*inc;
line(x,60,x,70);
}
for(i=0;i<col;i++)
{x=50+i*inc;
if(distance[i]==-1)continue;
setcolor(a[i]);
outtextxy(x,50,itoa(a[i],str,10));
y=60+20*(distance[i]+1);
line(x,60,x,y);
for(j=j+1;a[j]!=a[i];j++);
x+=(j-i)*inc;
line(50+inc*i,y,x,y);
line(x,y,x,60);
outtextxy(x,50,itoa(a[i],str,10));
distance[j]=-1;
}
setcolor(WHITE);
settextstyle(DEFAULT_FONT,0,2);
outtextxy(40,400,"ROUTING IN A MASK-PROGRAMMED CHANNEL");
getch();
closegraph();
}
```

Fullsegment.cpp

```
#include<conio.h>
#include<stdio.h>
#include<graphics.h>
#include<stdlib.h>
#define UNUSED 0
#define USED 1
void main()
{
int gd,gm,distance[100],left,right,ind,x,y,possible;
int a[100],flag[100],i,j,m,n,col,row,inc,max=0;
char *str;
float r;
clrscr();
printf("enter the number of connections\n");
scanf("%d",&m);
printf("enter their left pin and right pin nos. starting from 0\n");
for(i=0;i<100;i++)
a[i]=0;distance[i]=-1;
for(i=1;i<=m;i++)
{
scanf("%d%d",&left,&right);
if(right>max)
max=right;
a[left]=a[right]=i;
}
printf("enter the number of tracks\n");
scanf("%d",&n);
for(i=0;i<n;i++)
flag[i]=UNUSED;
col=max+1;
row=n;
for(i=0;i<col;i++)
{
ind=0;
if(a[i]==0)continue;
for(j=0;j<i;j++)
{
if(a[j]==a[i])
{
ind=1;
flag[distance[j]]=UNUSED;
distance[i]=distance[j];
break;
}
}
if(ind==1)continue;
possible=0;
for(j=0;j<row;j++)
{
if(flag[j]==UNUSED)
{
distance[i]=j;
flag[j]=USED;
possible=1;
break;
}
}
if(possible==0)
{
printf("No arrangement possible with the desired layout\n");
getch();
exit(-1);
}
}
detectgraph(&gd,&gm);
initgraph(&gd,&gm,"..\\bgi");
inc=620/(col+1);
for(i=0;i<n;i++)
line(inc,y=60+i*30,inc*col,60+i*30);
for(j=0;j<col;j++)
line(inc*(j+1),50,inc*(j+1),y);
for(i=0;i<col;i++)
{
if(distance[i]==-1)continue;
setcolor(a[i]);
setfillstyle(SOLID_FILL,a[i]);
pieslice(x=inc*(i+1),60+distance[i]*30,0,360,4);
for(j=i+1;a[j]!=a[i];j++);
x+=inc/2;
while(x<=inc*(j+1))
{
pieslice(x,60+distance[i]*30,0,360,4);
x+=inc;
}
pieslice(inc*(j+1),60+distance[i]*30,0,360,4);
outtextxy(inc*(j+1),40,itoa(a[i],str,10));
outtextxy(inc*(i+1),40,itoa(a[i],str,10));
distance[j]=-1;
}
setcolor(WHITE);
for(i=0;i<2*col-1;i++)
for(j=0;j<n;j++)
circle(inc*(i*0.5+1.0),60+j*30,4);
setcolor(WHITE);
settextstyle(DEFAULT_FONT,0,2);
outtextxy(40,400,"ROUTING IN A FULLY SEGMENTED CHANNEL");
getch();
closegraph();
}
```

1Segment.Cpp

```
#include<stdio.h>
#include<conio.h>
#include<graphics.h>
#include<stdlib.h>
#define UNUSED 0
#define USED 1
#define OPEN 1
#define CLOSED 1

void main()
{
int gd,gm,distance[100],left,right,ind,x,y,k,temp,possible;
int status[100][100],a[100],flag[100],i,j,m,n,col,row,inc;
char *str;
float r;
clrscr();
printf("enter the number of connections\n");
scanf("%d",&m);
printf("enter their left and right pin nos. starting from 0");
printf("and including the switches:\n");
for(i=0;i<100;i++)
a[i]=0;
for(i=0;i<100;i++)
{
for(j=0;j<100;j++)
status[i][j]=UNUSED;
distance[i]=-1;
}
for(i=1;i<=m;i++)
{
scanf("%d%d",&left,&right);
a[left]=a[right]=i;
}
printf("enter the no. of tracks\n");
scanf("%d",&n);
for(i=0;i<n;i++)
flag[i]=UNUSED;
col=right+1;
row=n;
printf("enter the vertax no. of switches that open\n");
scanf("%d",&i);
while(i!=-1)
{
status[i/col][i%col]=OPEN;
scanf("%d",&i);
}
for (i=0;i<col;i++)
{
if(a[i]==0)continue;
if(distance[i]!=-1)
{
flag[distance[i]]=UNUSED;
continue;
}
```

```
possible=0;
for(j=0;j<row;j++)
{
temp=0;
if(flag[j]==USED)continue;
for(k=i+1;a[k]!=a[i];k++)
{
if (status[j][k]==OPEN)
{
temp=1;
break;
}
}
if(temp==0)
{
possible=1;
status[j][i]=status[j][k]=CLOSED;
distance[i]=distance[k]=j;
flag[j]=USED;
break;
};
}
if(possible==0)
{printf("No arrangement possible with the desired layout\n");
getch();
exit(-1);
}
}
detectgraph(&gd,&gm);
initgraph(&gd,&gm,"..\\bgi");
setcolor(WHITE);
setfillstyle(SOLID_FILL,WHITE);
inc=620/(col+1);
for(i=0;i<row;i++)
line(inc,y=60+i*20,inc*col,60+i*20);
for(j=0;j<col;j+=2)
{
if(a[j]!=0)
outtextxy(inc*(j+1),40,itoa(a[j],str,10));
line(inc*(j+1),50,inc*(j+1),y);
}
for(i=0;i<row;i++)
for(j=0;j<col;j+=2)
{
if(status[i][j]==UNUSED)
{setcolor(WHITE);
circle(inc*(j+1),60+i*20,5);
}
else
{
setcolor(a[j]);
setfillstyle(SOLID_FILL,a[j]);
pieslice(inc*(j+1),60+i*20,0,360,5);
setcolor(WHITE);
circle(inc*(j+1),60+i*20,5);
}
```

```
}
setcolor(BLACK);
setfillstyle(SOLID_FILL,BLACK);
for(i=0;i<row;i++)
for(j=1;j<col;j+=2)
{
if(status[i][j]==-1)
pieslice(inc*(j+1),60+i*20,0,360,5);
}
setcolor(WHITE);
settextstyle(DEFAULT_FONT,0,2);
outtextxy(60,400,"ROUTING IN A 1-SEGMENT CHANNEL");
getch();
closegraph();
}
```

2Segment.cpp

```
#include<conio.h>
#include<stdio.h>
#include<graphics.h>
#include<stdlib.h>
#define UNUSED 0
#define USED 1
#define OPEN 1
#define CLOSED 1
#define SWITCH 2
void main()
{
int gd,gm,connect,distance[100],left,right,ind,x,y,k,temp,possible;
int status[100][100],a[100],flag[100],i,j,m,n,col,row,inc;
char *str;
float r;
clrscr();
printf("enter the number of connections\n");
scanf("%d",&m);
printf("enter their left and right pin nos. starting from 0");
printf("and including the switches:\n");
for(i=0;i<100;i++)
a[i]=0;
for(i=0;i<100;i++)
{
for(j=0;j<100;j++)
status[i][j]=OPEN;
distance[i]=-1;
}
for(i=1;i<=m;i++)
{scanf("%d%d",&left,&right);
a[left]=a[right]=i;
}
printf("enter the no. of tracks\n");
scanf("%d",&n);
for(i=0;i<n;i++)
flag[i]=UNUSED;
col=right+1;
row=n;
printf("enter the vertax no. of switches that present \n");
scanf("%d",&i);
while(i!=-1)
{
status[i/col][i%col]=SWITCH;
scanf("%d",&i);
}
for (i=0;i<col;i++)
{
if(a[i]==0)continue;
if(distance[i]!=-1)
{
flag[distance[i]]=UNUSED;
continue;
}
possible=0;
```

```
for(j=0;j<row;j++)
{
temp=0;
if(flag[j]==USED)continue;
for(k=i+1;a[k]!=a[i];k++)
{
if (status[j][k]==SWITCH)
{
connect=k;
temp++;
if(temp==2)
break;
}
}
if(temp<=1)
{
possible=1;
status[j][i]=status[j][k]=CLOSED;
if(temp==1)status[j][connect]=CLOSED;
distance[i]=distance[k]=j;
flag[j]=USED;
break;
};
}
if(possible==0)
{printf("No arrangement possible with the desired layout\n");
getch();
exit(-1);
}
}
detectgraph(&gd,&gm);
initgraph(&gd,&gm,"..\\bgi");
setcolor(WHITE);
setfillstyle(SOLID_FILL,WHITE);
inc=620/(col+1);
for(i=0;i<row;i++)
line(inc,y=60+i*20,inc*col,60+i*20);
for(j=0;j<col;j+=2)
{
if(a[j]!=0)
outtextxy(inc*(j+1),40,itoa(a[j],str,10));
line(inc*(j+1),50,inc*(j+1),y);
}
for(i=0;i<row;i++)
for(j=0;j<col;j+=2)
{
if (status[i][j]==OPEN)
{setcolor(WHITE);
circle(inc*(j+1),60+i*20,5);
}
else if(status[i][j]==CLOSED)
{
setcolor(a[j]);
setfillstyle(SOLID_FILL,a[j]);
pieslice(inc*(j+1),60+i*20,0,360,5);
setcolor(WHITE);
circle(inc*(j+1),60+i*20,5);
}
}
setcolor(BLACK);
setfillstyle(SOLID_FILL,BLACK);
for(i=0;i<row;i++)
for(j=1;j<col;j+=2)
{
if(status[i][j]==SWITCH)
circle(inc*(j+1),60+i*20,5);
else if (status[i][j]==CLOSED)
{setcolor(WHITE);
setfillstyle(SOLID_FILL,WHITE);
pieslice(inc*(j+1),60+i*20,0,360,5);
}
}
setcolor(WHITE);
settextstyle(DEFAULT_FONT,0,2);
outtextxy(60,400,"ROUTING IN A 2-SEGMENT CHANNEL");
getch();
closegraph();
}
```