A GRAPH BASED HEURISTIC CHANNEL ROUTER

by

Cheung-Lai Tse

A thesis
presented to the University of Manitoba
in partial fulfillment of the
requirements of the degree of
Master of Science
in
Electrical Engineering

Winnipeg, Manitoba, Canada

© Cheung-Lai Tse, 1987

Permission has been granted to the National Library of Canada to microfilm this thesis and to lend or sell copies of the film.

The author (copyright owner) has reserved other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without his/her written permission.

L'autorisation a été accordée à la Bibliothèque nationale du Canada de microfilmer cette thèse et de prêter ou de vendre des exemplaires du film.

L'auteur (titulaire du droit d'auteur) se réserve les autres droits de publication; ni la thèse ni de longs extraits de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation écrite.

ISBN 0-315-37294-X

A GRAPH BASED HEURISTIC CHANNEL ROUTER

BY

CHEUNG-LAI TSE

A thesis submitted to the Faculty of Graduate Studies of
the University of Manitoba in partial fulfillment of the requirements
of the degree of

MASTER OF SCIENCE

© 1987

Permission has been granted to the LIBRARY OF THE UNIVERSITY OF MANITOBA to lend or sell copies of this thesis, to the NATIONAL LIBRARY OF CANADA to microfilm this thesis and to lend or sell copies of the film, and UNIVERSITY MICROFILMS to publish an abstract of this thesis.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

ABSTRACT

The VLSI channel routing problem is addressed in this thesis. First, major classes of routing algorithms, including maze-running, line-search, and channel routing algorithms, as well as other routing approaches, including hardware routers, expert routers, and the simulated annealing technique are described. Then, the development and implementation of a graph-based heuristic (non-exhaustive search) non-dogleg channel routing algorithm is described. The algorithm is capable of generating optimal or near optimal solutions for an important class of channels that arises frequently in gate array, standard cell, and building block layout designs. The efficiency of the algorithm has been demonstrated through twelve examples obtained from published literature. The algorithm produces optimal non-dogleg solutions for nine of the twelve examples using a single set of parameters. In particular, Deutsch's Difficult Example was routed in 28 tracks, which only a related router of Yoshimuro and Kuh was able to obtain.

In order to produce routing solutions at or near channel densities (the least lower bound on channel heights), and to cope with channels with vertical constraint loops, the non-dogleg algorithm was extended to allow doglegging at terminal positions. The extended algorithm routed nine of the twelve examples in density, and Deutsch's Difficult Example in 20 tracks, which is the same as the best published result from channel routers of the same nature (dogleg at terminal positions only). In addition, the extended algorithm is able to handle a large class of channels with vertical constraint loops.

ACKNOWLEDGEMENTS

I wish to express my sincere thanks to my advisor, Dr. W. Kinsner, for his excellent guidance, endurable motivation and consistent support throughout the course of this research, and for his suggestion of this research topic.

I would also like to thank all the students and staff in the Industrial Applications of Microelectronics Centre, Inc., Winnipeg, Manitoba, Canada, for their support throughout my stay in the Centre. In particular, I would like to thank Joe Silva and Scott Handford for their help in using the Optimate PCB layout design package on the Apollo workstation. In addition, I would like to thank Dr. W. L. Kocay for his comment on NP-completeness.

Finally, the partial financial support from the University of Manitoba Graduate Fellowship, the National Sciences and Engineering Research Council (NSERC) of Canada and Manitoba Strategic Research Contract through Dr. W. Kinsner's grants, and the Industrial Applications of Microelectronics Centre, Inc., is gratefully acknowledged.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

DEFINITION OF TERMS

Algorithm An algorithm is a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time [AHO83].

branch-and-bound An implicit exhaustive search that eliminates sets of suboptimal solutions by estimated optimality upper bounds.

greedy A greedy algorithm attempts to obtain the optimal solution by selecting at each step the option that is locally optimal. The solution is not necessarily optimal.

heuristic A heuristic algorithm utilizes rules or experience from similar types of problems to quickly produce good but not necessarily optimal solutions.

Building Block With the building block design method, circuit modules are grouped into blocks. Placement and routing are performed in the block level only.

Channel A routing area formed between circuit modules.

density The maximum number of wire traces that crosses a vertical track in a channel. This is the least lower bound on the channel height.

ordering The maximum ordering number in vertical constraint graph of the channel. This is a lower bound on the channel height.

rectilinear A general rectilinear channel is a channel that has a rectilinear boundary (not necessarily rectangular) and terminals located on any or all sides.

regular A regular channel is a channel that has a rectangular boundary and terminals located on two opposite sides only.

switchbox A switchbox is a special rectilinear channel that has a rectangular boundary and terminals located on any or all four sides.

Circuit Layout A circuit layout is the physical representation of the corresponding structural representation of a circuit.

Doglegging Doglegging is the bending of an otherwise straight wire.

Expert System An expert system is a computer program that embodies the expertise of one or more experts in some domain and applies this knowledge to make useful inferences for the user of the system [HAY83].

Gate Array Gate arrays consist of a matrix of identical components or functional elements (cells) that has passed through all the steps in the fabrication process except the final interconnection stage (metalization).

Graph A graph consists of a set of points called vertices, and lines connecting the points, called edges.

directed A directed graph, G=(V,E), consists of a set of vertices V and a set of edges E, where the edges are ordered pairs of vertices (v,w) or v→w.

undirected An undirected graph, G=(V,E), consists of a set of vertices V and a set of edges E, where the edges are unordered pair of vertices.

Horizontal Constraint Graph An undirected graph representing the horizontal constraint relationships between the nets in a channel.

Layout Compactor A layout compactor spaces the circuit elements and interconnections to pack the circuit element as tightly as possible without violating constraints defined by the user and the design rules.

MIMD Multiple Instruction Multiple Data. A mode of parallel processing where each node processor follows its own instruction stream.

Net A net consists of a set of terminals and connections that makes the terminals electrically common and isolated from other nets or circuit modules. A net is realized as a collection of wire segments connecting the set of specified terminals in the layout design process through

the routing process. Net = (terminals, connections) [KIN86a].

Net List A net list is the list of all the nets and their associated terminals in a circuit.

NP-Completeness A decision problem is a problem with a yes or no answer. A polynomial (P) problem is a decision problem for which there is an algorithm which will solve (i.e., answer yes or no) any instance of the problem in a polynomial number of steps. A certificate for a decision problem whose answer is yes is a character string which demonstrates the answer (e.g., for the problem: Is there a routing with channel height ≤ 20, a certificate could be the actual routing). A non-deterministic polynomial (NP) problem is a decision problem such that for every instance of the problem whose answer is yes, there is a certificate which can be verified in a polynomial number of steps. Every P problem is also an NP problem. A problem is NP-complete if it is an NP problem and all NP problem is polynomial-time reducible to it. The algorithm for an NP-complete problem is universal, in that all other NP problems can be solved using such an algorithm. An NP-complete problem is very hard and probably has no polynomial time algorithm [SHI86, PAP82].

Overflow An overflow is a connection that cannot be routed under the given routing constraints (specification) [KIN86a].

Placement Placement is the process of arranging all the components within a two-dimensional area such that the placement configuration will facilitate the routing process [KIN86a].

Pseudo Language A pseudo language is a combination of the constructs of a programming language together with informal English statements [AHO83].

Routing Routing is the process of converting the set of intended connections into physical wires within the routing region using one or more routing layers, provided that physical and electrical constraints are satisfied [KIN86a].

detailed Detailed routing is the last step of the complete routing process. It defines the exact geometry of the wires in terms of layer, via, and track assignments.

loose Loose routing (or global routing) is the preliminary step of the complete routing process. It calls for a routing plan in which each net is assigned to particular routing regions without specifying the exact geometry of the interconnecting wires.

Routing Algorithm A routing algorithm is a method of tracing wires on a routing medium according to the specified constraints [KIN86a].

maze-running A class of sequential methods of tracing wires along an expanding area, from the starting point to the end point. A path between the two points can always be found if such a path exists. The method was first introduced by Lee.

line-search A class of sequential methods of tracing wires along straight lines until a blockage is encountered. The method was originally developed by Hightower.

channel A class of sequential methods of tracing wires along channels formed by modules or components. The first phase of the channel routing algorithm is a global assignment of nets to channels, followed by the second phase of local assignment of nets to tracks within the channels. The method was first introduced by Hashimoto and Stevens.

Router A router is the software or hardware implementation of a routing algorithm.

expert An expert router utilizes an expert system to perform routing. See also Expert System.

grid A grid router performs routing on a grid. Wire segments are constrained to lay only on the grid.

gridless A gridless router performs routing without the constraint of a grid.

hardware A hardware router implements a routing algorithm in hardware.

Routing Void A routing void is the area in the routing region where traces cannot be placed.

SIMD Single Instruction Multiple Data. A mode of parallel processing where instructions are broadcasted to all node processors. Thus each node processor executes the same instruction but operates on different local data.

Simulated Annealing A multivariate optimization technique analogous to the cooling of a fluid into a low energy state.

Standard Cell The standard cell design method is based on a library of predesigned functional cells, each of which has been fully characterized in both electrical and performance terms.

Terminal A terminal is the endpoint of a connection [KIN86a].

Trace A trace is the physical representation of a connection that makes different points in a circuit electrically common. A trace can only be defined on a single layer. To connect traces on different layers vias are required. Traces and vias together constitute wires [KIN86a].

Vertical Constraint Graph A directed graph representing the vertical constraint relationships between the nets in a channel.

Vertical Constraint Loop A directed cycle in the vertical constraint graph. A channel with vertical constraint loops is unroutable without doglegs.

Via A via is a feed-through or a contact where wire segments on different layers are connected together. It contributes to the creation of wires.

Wire A wire is the physical realization of a net which makes different points in a circuit electrically common. A wire includes at least one trace and two terminals. If the traces are located on different layers, the wire also includes at least one via. Wire = {terminals, traces, vias} [KIN86a].

CHAPTER I
INTRODUCTION

The design of electronic circuits can, in general, be considered as the transformation of a behavioural description of the circuit concepts into a physical description for implementation. For complex systems such as very large scale integrated circuits (VLSI), the transformation process is achieved by a hierarchical decomposition from behavioural descriptions to structural descriptions, then to physical descriptions.

A behavioural description is the textual or mathematical description of a system. It gives a precise definition of the system behaviour with no concern in its actual implementation. For example, the addition of two multi-digit binary numbers A and B could be described behaviourally as A+B. However, how the operation is actually implemented, for example, whether sequentially with a one bit adder or simultaneously with a multi-bit parallel adder, is not described. It is the structural description that defines the translation of the behavioural description into interconnecting functional blocks in the form of, for example, data-flow diagrams and structural charts. A more detailed structural description could include circuit schematics for the hardware, and software description language specifications for the software and firmware. With the higher level descriptions specified, the physical representation of a system is the final stage of the design by which structures are translated into physical layouts. Many different form of layout styles are possible. Common methods include the use of standard components mounted on printed circuit boards (PCBs) or surface mount boards (SMBs), and

semi-custom or full-custom integrated circuits fabricated on silicon dies.

Continuing advances in integrated circuit technology are driving circuit densities to higher and higher levels. This ever increasing circuit complexity has rendered the already tedious, error-prone, and time-consuming layout process almost impossible to be handled manually. Compounding the problem is the requirement of very high quality layouts that must consider complex physical constraints such as ringings, crosstalks [WEX85, POL86], current surges and heat dissipations [KIN86c]. One typical example of manual layout design was the Z8000 microprocessor. In its design, very little computer aids were used. As a result, 50% of the whole design effort, or 6600 man-hours, was devoted to the layout design phase alone [RIC80].

As the design gets more complex, the design effort and turnaround time increase at a higher rate, particularly in the layout design phase. Consequently, a large amount of effort has been devoted to providing computer aids to the human designer with the layout design problem. Such layout design aids fall into the general category of computer-aided design (CAD) and computer-aided engineering (CAE) tools.

1.1 VLSI Layout Design

The design of VLSI circuits, being a branch of electronic circuit design, involves the process of transforming a given circuit behaviour into a circuit laid out on a silicon die. The first phase of this transformation, as with any electronic designs, is the behavioural design phase. It converts circuit concepts into formally defined behavioural descriptions. The second phase of the transformation is the structural design phase in which a network of

interconnecting components or modules is designed realizing the specified behaviour. The modules may be large functional blocks such as ALUs or PLAs, logic gates such as NAND or NOR gates, or even isolated transistors or resistors. The interconnections, on the other hand, are usually specified as nets connecting terminals of the modules. Note that at this stage of the design, modules and interconnections are still conceptual units. It is in the final physical or layout design phase that the network of functional blocks is mapped onto the surface of a silicon die giving the precise geometry and position of its constituent modules and interconnecting wires.

The VLSI layout problem can be described as follows: A number of circuit modules are to be arranged in a given area such that there are no overlaps between the modules, and all interconnecting terminals are to be connected by mutually noninterfering wires laid out in designated routing regions. More precisely, the modules of a given circuit are to be placed within a two dimensional region in such a way that each module takes on a unique area in the region and the arrangement of the modules is such that it facilitates the routing of interconnections. Furthermore, for a given interconnection list (net list) and module placement configuration, all electrical connections must be converted into physical connections within designated routing regions satisfying design constraints including electrical constrains such as maximum signal delays and design rule constraints such as minimum feature clearances.

The layout design problem, like many design automation problems such as logic synthesis, testing, and partitioning, is widely known to be NP-complete [UED86, KIN87]. The optimal solution to this problem requires running times that grows exponentially with the size of the problem. It is

highly unlikely that an efficient polynomial time algorithm exists [BRE76]. With circuit complexities pushing into upwards of a million transistors per die, the layout design problem has grown beyond the capability of today's computers. For example, just arranging the modules of a 20,000 gate circuit using the commercial placement program COSMIC [SCH83] requires over 300 CPU hours [UED86], a time approaching the mean time to failure of many complex computing systems. Even the recent increases in computing power coupled with efficient and reliable software still cannot guarantee that the densest VLSI circuits can be laid out completely in a reasonable time and storage [LUD83].

1.2 VLSI Layout Strategies

Various simplification methods have been used to reduce the VLSI layout problem into more manageable sub-problems. Traditionally, the method of partitioning is used to simplify the problem by dividing the layout design process into two separate steps, namely placement and routing. In the placement step circuit modules are assigned to physical locations on the die. Then, in the routing step nets are realized as wire traces connecting the terminals. Both of these subproblems, though simpler, are still NP-complete [SHI86]. But the overall reduction in complexity is significant enough to allowed many previously unmanageable designs to be tackled.

Partitioning of the layout design problem into two separate and disjoint steps has, however, necessitated the use of iterative processes involving repeated application of the placement and routing steps. The reason for this cyclic phenomenon is the intrinsic mutual dependency of the

placement and routing processes. If a placement configuration does not allow a reasonable level of routing completion, it would be necessary to repeat the placement process and then re-do the routing based on the new placement configuration. This cycle must be repeated until an acceptable layout is generated.

So far, most computer-aided layout systems have taken this separate placement and routing approach. However, as both steps are so critically dependent on each other, there are problems with such an iterative method, particularly as the scale of the design extends more into the VLSI area:

1. Placing the modules with no knowledge of how the routing process will route the interconnecting wires makes the placement process particularly difficult to generate the optimum placement configuration.
2. Dividing the layout process into two disjoint steps with no communication between the placement and the routing processes unnecessarily increases the number of iterations required. The inability for the routing process to provide feedbacks to the placement process in exactly why routing was failed in case of an incompletion essentially makes the placement process a blind process. The placement process does not know what is required by the routing process. When routing fails, it simple generates another placement configuration and hope that it will find an acceptable solution in a reasonable number of trials.

In light of these problems, attempts have been made to combine the placement and routing process [LOO79, SOU79, BUR85, SZE86]. However the research is still in its infancy and no significant results have been reported. The attempts reported so far have been primarily concerned with highly

regular structures such as gate arrays, and the improvements in the layout were too small to justify the enormous running time [SOU79].

In addition to the above method of partitioning the VLSI layout design process into placement and routing, another approach is to begin with a rough placement of the circuit modules and routing of the interconnections, and then perform a layout compaction to optimize the layout. The initial rough layout could be the result of a symbolic (stick) layout design [MEA80] to be translated into final mask layout, or the result of a placement/routing process requiring a further optimization. In any case, once the initial design is completed, a compaction process is applied to optimize the design. In this approach, the initial rough placement configuration represents only the topological arrangement of the modules, that is, the relative positions of the modules only. The actual physical placement of the modules are determined by the compactor according to the process design rules and user defined constraints such predefined module locations. In the compaction process, the interconnecting wires are considered as stretchable. That is, without changing the topology of the routing, the compactor is free to shrink or stretch any wire [LIA83].

Most compactors use minimum area or maximum density as their main goal. In other words, their goal is to achieve the minimum die size for a circuit given the design rules, the components, the interconnections, and the user defined constraints. Although minimum area may not be the only requirement, it well reflects many others, for example minimum signal delays and maximum yield.

The general layout compaction problem has also been proven to be

NP-complete [SAT83, TAY84]. A mathematical representation of the layout compaction process is: minimize the product, Area = Max(x)Max(y), where x and y are horizontal and vertical dimensions of the die. This formulation results in a quadratic problem that is very difficult to solve [GAR79]. In fact, for all but very small cases, its solution is unreasonable on today's computers. To make the problem manageable, the following assumption is made: minimization of the layout is achieved when the layout in each dimension is minimized. The implication here is that the original two-dimensional compaction problem can be decomposed into two one-dimensional problems, each of which can be solved independently using simpler algorithms. The new formulation results in a linear problem that minimizes the vertical or horizontal dimension subject to the given constraints. But, because the layout design problem is two-dimensional, and because linear compactors minimize each dimension separately, the order in which compaction is performed, that is, a horizontal compaction first or a vertical compaction first, becomes important [TAY84].

Encouraging results have been reported with the compaction approach to layout design [OHT86]. However, this layout style is still relatively new and no significant results have been reported. Therefore, not until better compaction algorithms, particularly two-dimensional compaction algorithms, are developed, this method remains primarily as a research.

In this thesis, the common approach of partitioning the layout design problem into placement and routing will be used. In the next two sections, a brief overview of the placement and routing process will be presented.

1.3 Placement

In general, VLSI placement is the process of arranging all circuit modules within the die area such that the resulting placement configuration will facilitate the routing process [KIN86a]. Although the modules are placed on the die according to a number of constraints such as heat dissipations and signal crosstalks, the main objective of the placement process is to generate a placement configuration that would allow 100% routing within the given area [SOU81]. Other constraints represent merely secondary objectives that impose additional requirements on how some modules must be placed. For example, due to heat dissipation considerations some modules may have to be placed in certain fixed positions, or in order to reduce signal crosstalks some modules may have to be placed next to some specific modules. But the ultimate objective is still to generate a 100% routable placement configuration that satisfies those constraints. Such an objective is, however, not mathematically well defined. It is very difficult to predict whether 100% routing can be achieved without performing the actual routing.

To access the quality of a placement configuration, pre-routing (post-placement) analysis schemes and routability indicators have been reported in literature [SO73, FOS75, HEL77, KON86, SAS86]. Although they usually do not require as much running time as the actual routing process, a comprehensive analysis still requires an appreciable amount of time. Therefore, such analyses are normally used to access the quality of the final placement configuration only and not the intermediate configurations. In particular, it is commonly used to choose between alternative placement configurations, or to identify congested areas so that they can be avoided early in the routing process.

In order to guide the placement process towards a final solution, a simplified objective function is used. The assumption is that if the simplified objective function is improved, the routability is also improved. Various simplified objective functions have been proposed. The three most representative ones are the total routing length, the maximum cut line, and the maximum density [GOT86].

Regardless of the specific objective function selected, most of the placement algorithms can be classified as either constructive or iterative [HAN72, SOU81, GOT86]. Constructive algorithms produce a solution using heuristic rules, often in a sequential and deterministic manner [GOT86]. Iterative algorithms, on the other hand, produce a solution by successive modification of the initial solution. In most CAD/CAE layout design systems, algorithms from both of these classes are employed. Usually an initial solution is obtained using a constructive algorithm and the solution is improved gradually with an iterative algorithm.

For constructive placement algorithms the placement configuration is formed by adjoining unplaced modules to the set of placed modules. One by one, the unplaced modules are selected and positioned in the partially formed placement configuration. Once a module is positioned, it will not be moved again. An example of constructive placement is the clustering algorithm: for each unplaced module a measure of the expected number of interconnections to the placed modules is computed and the one with the largest value is selected for placement.

Iterative placement algorithms, on the other hand, improve upon a placement configuration by applying small local changes. Typically a subset

of modules is selected and deterministically repositioned until the best configuration is found. A widely known algorithm is the force directed placement method, where the connections between modules are interpreted as springs that stretched out due to displacements between the modules.. The placement configuration is then pulled together by successively reducing the forces of connections between modules using techniques such as a pair-wise exchange of modules.

Using whichever technique, once the modules are placed, the next step in the layout design process is to route the interconnections. In the next section, a brief overview of the routing process will be presented.

1.4 Routing

Routing is the process of converting interconnections into wires within the designated routing regions according to constraints such as maximum wire lengths [KIN85]. The interconnections of module terminals, usually specified in the form of a net list, are made through one or more routing layers in the designated routing regions. Generally, two routing layers are used with vertical and horizontal traces on alternate layers. A physical wire changes direction by means of a via at the intersection of a horizontal trace and a vertical trace.

Many routing algorithms have been developed in the past three decades. The first recognized algorithm was developed by Lee in 1961 [LEE61]. The Lee algorithm is actually the shortest path algorithm by Moore [MOO59] applied to a grid structure representing the wiring space. Since Lee's original paper, a large number of extensions and variations have been published. These

algorithms are often referred to as grid expansion or maze-running algorithms due to their similarity to finding an entrance-to-exit path in a grid-structured maze.

Even though maze-running algorithms were created before the integrated circuits era and has been applied primarily to printed circuit board design, it is still used in many current VLSI layout design systems. This is due to the generality of the algorithm and the guarantee of finding a path if one exists. The main disadvantages of the Lee type maze-running algorithms lie in their large demand on memory and running time, and their inherent sequential nature of routing one net at a time. Such inherent negative features of the maze-running algorithms have given rise to other classes of routing algorithms. In the next chapter, a more detailed description of maze-running, line-search, and channel routing algorithms will be presented.

1.5 Motivation

The size of the resulting layout has always been a major concern in VLSI design. Smaller layouts are less expensive and may exhibit better performance. An over-sized die has several adverse effects on both the cost and the performance of an integrated circuit:

1. A larger die with modules placed farther apart requires longer interconnecting wires. As gate delays are becoming shorter, especially in very high speed integrated (VHSI) circuits using such technologies as emitter coupled logic (ECL) and Gallium Arsenide (GaAs), signal delays due to excessively long interconnecting wires is becoming an increasingly significant factor.

2. With a larger die size the number of circuits that can be fabricated on a wafer is reduced. Hence, the cost per die is increased.
3. As the number of functioning circuits per wafer decreases exponentially as the die size increases [GOT86], a larger die size implies a lower probability of a circuit functioning, a lower yield, and hence a higher cost.

It is, therefore, obvious that the larger the die size the poorer the performance and the higher the manufacturing cost.

In order to make efficient use of the available die area, modules of active devices and interconnecting wires compete with each other for space on the die surface. It has been observed that wiring can occupy more than half of the die area. Moreover, dead space containing no usable devices or wirings due to improper module placement and interconnection routing represents a major waste of precious die area. Therefore, in order to minimize the die size both the routing space and the dead space must be minimized.

Currently available CAD/CAE tools for VLSI layout design tend to use more die area than actually needed. In other words, they necessarily waste die space. Automatic layouts still cannot compete with manual layouts as far as the die size is concerned. This is because an experienced designer can understand the given design and find a more compact layout using his knowledge and intuitions. However, due to the extreme complexity of VLSI designs and a world shortage of skilled designers, in order to complete a design in a reasonable time and cost, CAD/CAE tools are indispensable despite their present inadequacies. In fact, it is widely recognized that the greatest impediment to the successful use of VLSI technology is the lack of adequate

CAD/CAE tools to support VLSI circuit design [LOS80].

In this thesis the VLSI routing problem, in particular, the channel routing problem, will be addressed. The main reason why routing remains such a difficult problem is its global nature: the way one interconnection is routed influences the potential solution for other interconnections. Any fragmentation of the routing task into smaller domains may affect this global aspect and hence the quality of the routing results [LUD83]. However, the problem of routing is simply too complex to be considered globally without subdividing it into smaller, more manageable, domains. As will be discussed in the next chapter on routing algorithms, the method of channel routing represents a simple yet effective approach to the VLSI routing problem.

1.6 Thesis Objectives

The objectives of this research work are:

1. To identify areas that can be improved in existing detailed routing algorithms through a survey of published results.
2. To improve the existing channel routing algorithms through the development of a detailed routing algorithm for regular channels in an attempt to obtain optimum or near optimum results for at least one class of channel routing problems.
3. To improve the detailed routing algorithm for regular channels by extending it to allow doglegs and hence relaxing the restriction on one horizontal track per net.

1.7 Thesis Structure

This thesis addresses the VLSI channel routing problem. The concept of channel routing, although introduced over a decade ago, is still the most popular routing strategy for LSI and VLSI layout designs. The next chapter describes on the major classes of routing algorithms, including maze-running, line-search, and channel routing algorithms, as well as other routing approaches, including hardware routers, expert routers, and the simulated annealing technique. It is then followed by Chapter III on the introduction of a non-dogleg detailed routing algorithm for regular channels. In Chapter IV, the extension of the non-dogleg algorithm to allow doglegs is described along with the implementation and the experimental results. Finally, conclusions for both the non-dogleg and the dogleg channel routing problems are drawn and recommendations for further research are given in the last chapter of this thesis.

CHAPTER II
VLSI ROUTING ALGORITHMS

Since the first routing algorithm was introduced by Lee in 1961 [LEE61], numerous routing algorithms have been developed. Many of those algorithms were very innovative and provided much understanding into the complex problem of routing. It is essential that those developments be understood before any fruitful research can be attempted. This chapter describes the three most important classes of routing algorithms that formed the basis of most computer-aided layout design systems today, namely, maze-running, line-search, and channel routing algorithms. In addition, other routing approaches, including hardware routers, expert routers and the simulated annealing technique are described.

2.1 Maze-Running and Line-Search Routing Algorithms

The maze-running and line-search algorithms are the most well known routing algorithms. The maze-running algorithms find a path between two points on a grid similar to finding an entrance-to-exit route in a grid-structured maze. The line-search algorithms, on the other hand, search for a path as a sequence of line segments. In the following two sections these two classes of algorithms will be described in detail.

2.1.1 Maze-Running Routing Algorithms

Maze-running or grid-expansion algorithms are grid based algorithms that find a connecting path between two terminals of a net on the grid similar to finding an entrance-to-exit path in a grid-structured maze. Algorithms in this class assume the use of a uniform rectangular grid. The spacing between the grid points is such that for wires less than a certain width routing on the grid points will automatically provide the proper clearance between the wires. Although the use of such a grid eliminates the possibility of curved or diagonal traces typically seen in manual layouts, this loss of generality substantially simplifies the routing problem, thus allowing more complex problems to be tackled. This concept of grid based routing has in fact given rise to the entire class of maze-running algorithms, and the use of CAD/CAE systems in layout designs.

Lee Algorithm

The first maze-running algorithm that aimed at automated wire routing was developed by Lee in 1961 [LEE61] and is commonly called the Lee algorithm. It was originally developed for single layer routing on a planar rectangular grid. The objective is to find the shortest path, if such a path exists, on the grid connecting the two terminals of a net. Before a more precise definition of the algorithm, it would be best to begin with a simple example illustrating the algorithm.

Assume a small routing region as shown in Fig. 1a. For a better illustration, the grid points are represented by rectangular cells. The numbers inside the cells are labels used in the algorithm, and the blackened

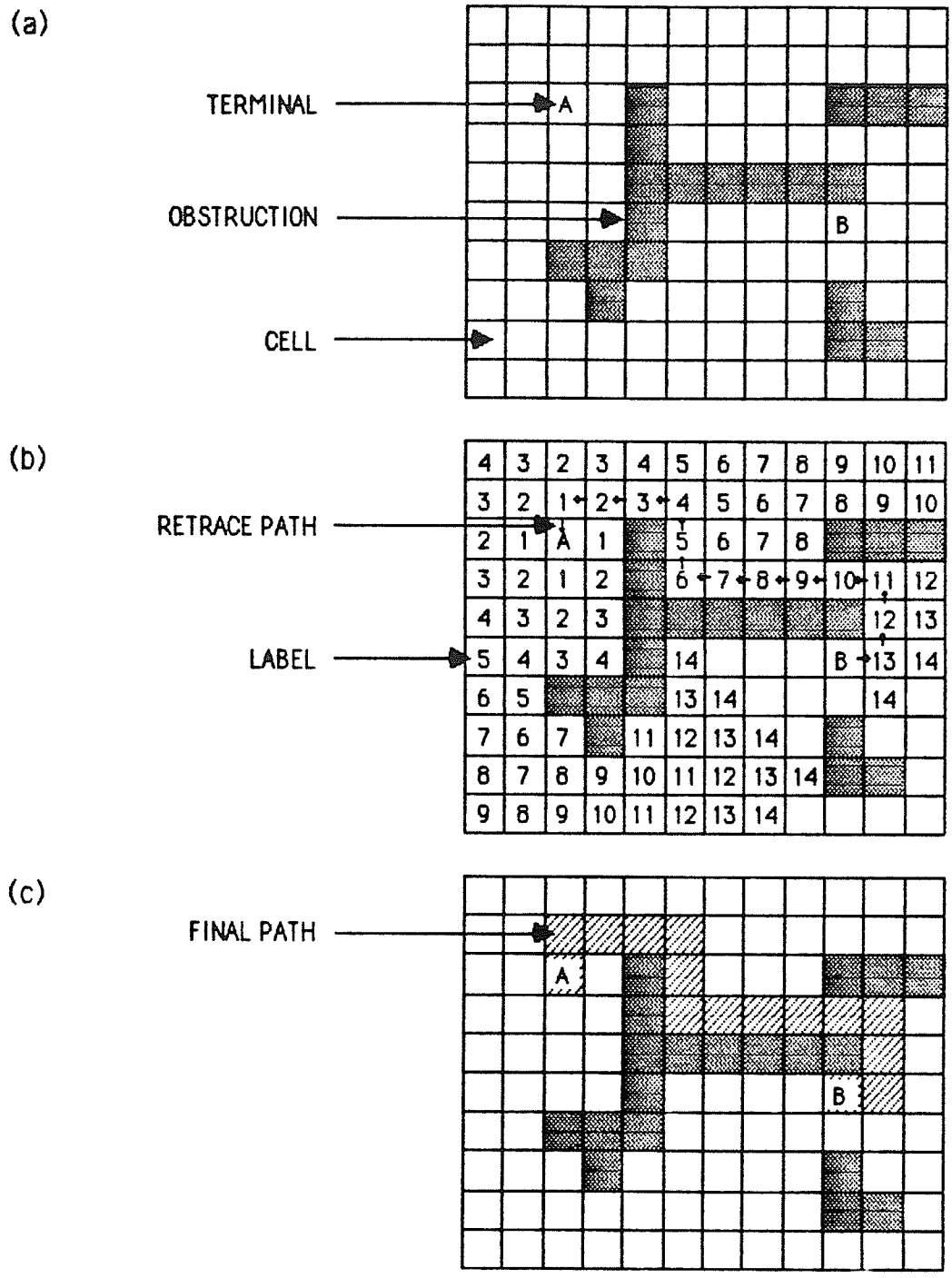


Fig. 1. Lee algorithm routing example. (a) Routing grid; (b) Cell expansion and retracing (note that B is labelled 14); (c) Final path between terminals A and B.

cells represent obstructions, such as routing voids or traces of routed nets. The cells labelled A and B are the two terminals of the net to be connected using the Lee algorithm. The goal is to find the shortest path between the two cells while avoiding the blackened obstruction cells.

The algorithm can be decomposed into two separate phases: the cell expansion phase and the retracing phase.

1. Cell Expansion (or wave propagation): The algorithm begins by selecting one of the two terminals as the source cell and the other terminal as the target cell. In theory, either terminal can be chosen as the source terminal. In practice, however, one terminal may be more desirable than the other, as will be seen later. Here, terminal A is selected as the source and terminal B as the target. With the source and target cells chosen, a label of 1 is entered into every empty cell immediately adjacent to the source cell. Since no diagonal traces are allowed, the adjacent cells are the north, west, south, and east cells. Next, a label of 2 is entered into every empty cell immediately adjacent to the cells labelled 1, and so on; increasing the label by one in every expansion step. This process continues until either (i) there are no empty cells adjacent to the labelled cells meaning that no path exists connecting the cells A and B on the grid and the algorithm terminates, or (ii) the target cell B is reached meaning that a path has been found and the algorithm proceeds to the next retracing phase. The key purpose of this cell expansion phase is in the labelling of the cells, where the numbers represent the Manhattan distance of the cells from the source cell. This process can also be viewed as a breadth-first-search in graph theory as the cells closer to the source cell are searched first.

2. Retracing: After the cell expansion phase, there exists at least one path connecting the source cell and the target cell among the expanded cells. However, the exact optimal path has not been determined. The main purpose of this retracing phase is to trace out the exact optimal path from the target cell back to the source cell. As shown in Fig. 1b the target cell B has been reached in the 14th expansion step. It follows that there must be a cell with a label of 13 adjacent to B. Similarly, for a cell with a label of 13 there must be a cell with a label of 12 adjacent to it, and so on. Therefore, by tracing the labelled cells in descending order from the target cell back to the source cell, the desired shortest path can be identified. In the retracing process, there are often more than one adjacent cell with the correct label. In theory, any one of those cells can be chosen and they will all yield paths having the shortest length. In practice, however, the cells not leading to a change in the path direction will normally be chosen to minimize the number of corners in the wire. For example, in Fig. 1b, the cell labelled 9 on the retracing path has two adjacent cells labelled 8. In such a case, the one to the left is chosen instead of the one above to avoid a change in the path direction. Once a path is selected, all the cells along the path are labelled as occupied and become obstructions for subsequent interconnections while the rest of the expanded cells are relabelled as empty and remain available for subsequent routing.

The result of these two steps are shown in Fig. 1c. The final optimal path is indicated by shaded cells in the diagram.

From the above example, it can be seen that the Lee algorithm is fairly simple. But the most important properties of the Lee algorithm are that: (i) it

guarantees to find a path if one exists, and (ii) it guarantees to find the optimal path if more than one path exist. As in the above example, the final path of length 14 is the shortest possible path connecting the cells A and B.

With the basic principles of the Lee algorithm illustrated in the above example, a more precise definition of the algorithm in pseudo-codes would help to clarify the details. First, a few definitions are required. Assume that the routing area is divided into a finite number, N, of subareas called cells labelled C_i. The cells representing obstructions such as routing voids or routed wires are labelled as occupied and are not available for routing. The remaining cells are labelled as free and can be used to form a path. Moreover, a cell C_i has associated with it a cost, $f(C_i)$, and a path p consisting of the starting cell S and a set of cells C_1, C_2, C_3, ..., C_n each adjacent to the previous cell forming an (n+1)cell path of

$$p = \{S, C_1, C_2, C_3, ..., C_n\}$$

has a path cost of, F(p), where

$$F(p) = \sum_{i=1}^{n} f(C_i).$$

In the original Lee path finding algorithm, the cost of a cell is the Manhattan distance between that cell and the source cell. However, path cost function can be generalized to any monotonic function or set of monotonic functions represented by a vector $F = [F_1, F_2, ..., F_n]$. A path cost function is monotonic if for every path p and any of its subpath p_i, the following inequality holds.

$$F(p_i) \leq F(p)$$

With the above definitions in order, the Lee algorithm can be described more precisely in pseudo-code. Instead of showing the exact algorithm as presented by Lee, the algorithm is restated here to show just the essence of the algorithm; in particular, the cell expansion and retracing phases. The pseudo-codes used are similar to the Pascal language, and the syntax should be self-explanatory.

[Lee (maze-running) Routing Algorithm]:

```
Lee( cell_map );
   cell_map:   an ordered list of cells.
               The list includes:
               1) the location of obstruction cells;
               2) the location of the source and target cells;
               3) the definition of cell neighbourhood;
               4) the path cost function.

               Each individual cell is a structure which includes:
               1) a cell identification field: id;
               2) a cell label field: label;

               The individual fields are referenced as cell.id and cell.label,
               respectively.

   (Variable Definitions)
   source:     the source cell;
   target:     the target cell;
   cell:       variable holding a cell;
   path_cell:  variable holding a cell in the retrace phase;
   neighbour:  variable holding a neighbour cell;
   label:      current label in the cell expansion phase;
   found:      boolean variable used to terminate cell expansion;
```

```
begin
(initialization)
label := source.label := 0;
found := false;

(cell expansion)
while not found and number of empty cells in cell_map > 1 do
   begin
   for all empty cell in cell_map do
      if cell.label = label then
         for neighbour in the neighbourhood set of cell do
            if neighbour.label = empty then
               neighbour.label := label + 1;
            else if neighbour.id = target.id then
               (target cell is reached; terminate cell expansion)
               found := true;
   label := label + 1;
   end;

(retrace)
if found then
   begin
   path_cell := target;
   repeat
      output path_cell.id;
      for neighbour in the neighbourhood set of path_cell do
         if neighbour.label = path_cell.label - 1 then
            path_cell = neighbour;
   until path_cell.id = source.id
   end
else
   (all cells have been examined and no path is found)
   output "no path is found."
end.                                                          □
```

From the above description of the algorithm it can be seen that the Lee algorithm requires at least N^2 memory for an N×N routing region, and $O(N^2)$ running time in the worst case, or $O(L^2)$ for a path of length L in the cell expansion phase and O(L) in the retracing phase. Obviously, for a large routing region, the memory required to store the cell labels and the running time required to find a path through the grid are the two major drawbacks of the algorithm. In light of these storage and speed problems, a large number of variations and extensions to the original Lee algorithm have been developed [AKE67, GEY71, RUB74, HOE76, KOR82].

Extensions of the Lee Algorithm

There are a large number of variations on the original Lee algorithm. Among the various extensions, several of the important ones are described in this section and they are organized into storage reduction techniques, speed up techniques, multi-terminal net extensions, and multi-layer extensions.

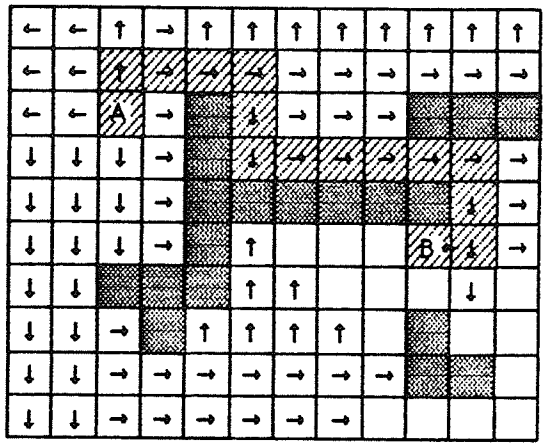
Storage Reduction Techniques

The huge storage requirement of the Lee algorithm represents a very serious drawback, especially for large scale or dense layouts. For example, a double layer 1000×1000 grid would have two million cells. Wire lengths as long as 1000 could be expected in such a grid. Thus, during the cell expansions process, labels as large as 1000 would be used requiring at least 10 bits of storage per cell. If double-byte words are used, a total of over 3.8 mega-bytes must be allocated to store the grid. However, as can be seen from the example in Fig. 1, the minimum amount of information that must be

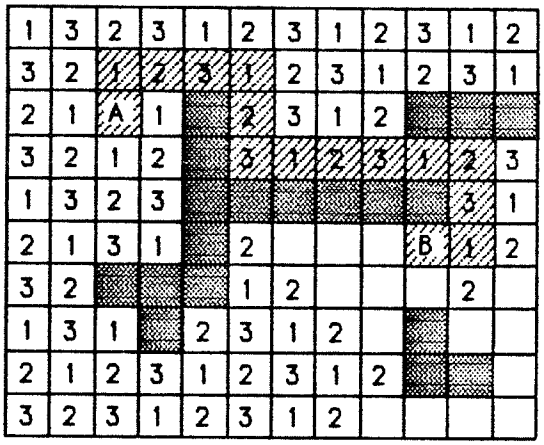
present includes only (i) a means of distinguishing between occupied cells and empty cells, and (ii) a means of distinguish between the predecessor cells from the successor cells in the retracing process. Based on this observation, a number of labelling scheme have been devised to encode the necessary information using fewer number of bits per label.

1. Direction Labelling Scheme: At each cell expansion step the adjacent cells are given direction labels of north, west, south, or east indicating the directions of expansion as shown in Fig. 2a. If a cell can be reached from more than one direction, one direction label is entered. Using the four direction labels, together with the empty cell and the occupied cell labels, there are a total of six distinct labels. Hence, three bits of storage is required per cell. This labelling scheme, however, has one drawback. Since only one of the cell expansion direction is entered, only one path can be found even if more than one exist. Thus the choice of selecting the path with fewer number of corners is sacrificed for the reduced storage.
2. 1-2-3 Labelling Scheme: Instead of storing the directions of expansion, a sequence of 1,2,3,1,2,3,... is used for labelling as shown in Fig. 2b. In the retracing phase, the reverse sequence is traced. This labelling scheme uses a total of five distinct labels, and thus still requires three bits of storage per cell. However, this technique allows multiple paths to be identified and the one with fewest corners to be selected.
3. 1-1-2-2 Labelling Scheme: This labelling scheme is similar to the 1-2-3 labelling scheme, except that a 1,1,2,2,1,1,2,2,... sequence is used as shown in Fig. 2c. Note that in the example the target cell B is reached with a 1 preceded by another 1. Therefore the retrace sequence

(a)



(b)



(c)

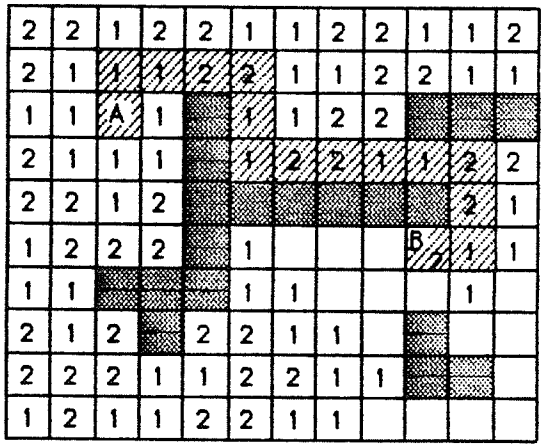


Fig. 2. Storage reduction techniques. (a) Direction labelling; (b) 1-2-3 labelling; (c) 1-1-2-2 labelling.

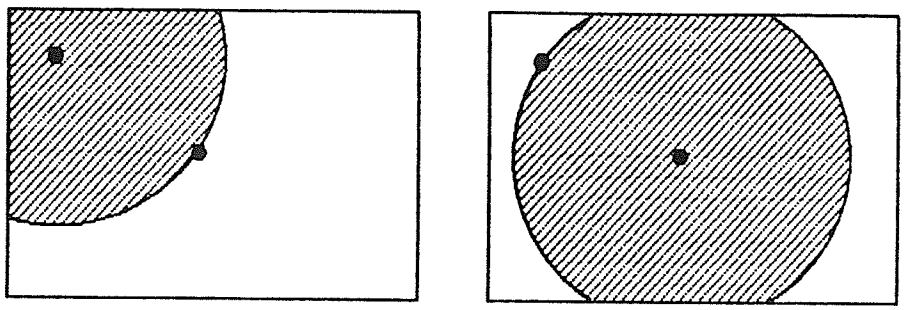
should be 1,1,2,2,1,1,... The biggest advantage of this labelling scheme is that only two labels are used in the sequence. Together with the occupied cell and the empty cell labels, a total of four distinct labels are required. Hence, only two bits of storage are required per cell instead of three as in the previous two schemes [AKE67].

All of the above labelling schemes are designed to reduced the amount of storage per cell. However, in practical routers, the selection of labelling schemes must be made according to the tradeoff between storage and efficiency [OHT86]. For example, although the 1-1-2-2 scheme requires only two bits per cell, the final label clearance process may be as involved as the cell expansion phase. Thus, additional bits are often used to simplify the processing.

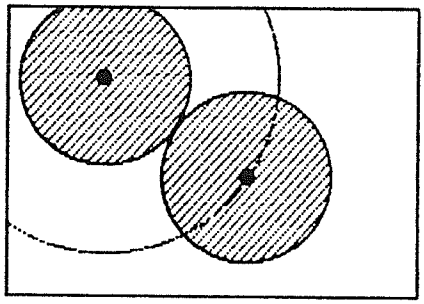
Speed-Up Techniques

Several speed up techniques are also possible. Most of such techniques are focused on the cell expansion phase. The objective is to reduce the number of cells that must be examined to find a connection path. However, in some speed up techniques such as double framing, the resulting path may not always be the optimum. The use of such techniques would thus involve a tradeoff between speed and optimality. The common approach is to find a solution quickly with a speed up technique, and if the resulting solution is unacceptable, the parameters are modified or a complete search is performed again. One point should be noted here is that, while the speed up techniques tend to reduce the average processing time, the worst case computational complexity remains unchanged.

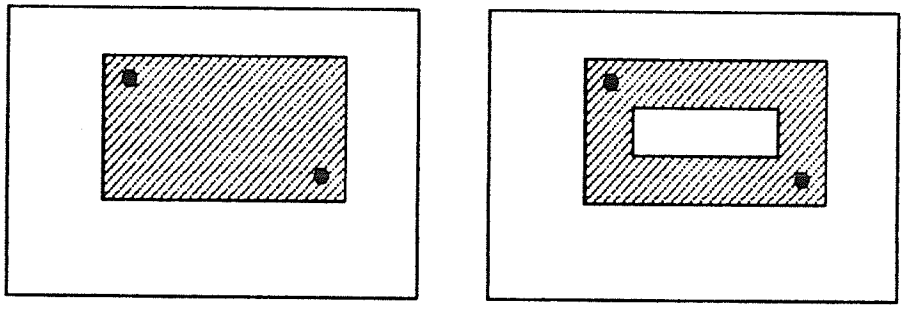
1. Starting Point Selection: For a given pair of terminals, it is more desirable to start the cell expansion process from the terminal closer to the boundary of the routing region since the number of cells that need to be examined are fewer (Fig. 3a).
2. Double Fanout: Cell expansion is performed from both terminals simultaneously until a point of contact is reached (Fig. 3b). This technique reduces the number of cells that need to be examined but requires a more complex scheme to keep track of the two simultaneous cell expansion processes.
3. Framing: An artificial rectangular frame is imposed around the terminal pair and no cell expansions are allowed outside this boundary (Fig. 3c). Typically, the frame is about 10-20% larger than the rectangle defined by the terminal pair and may be expanded or removed if a path cannot be found within this frame. The reason behind this technique is that, it has been observed that in most cases the shortest path between two cells is within the rectangle defined by the terminals. Therefore, by restricting the search area the probability of finding a path is still high while a considerable speed up is possible. A further extension to this framing technique is double framing, where a second, smaller frame is imposed inside the rectangle defined by the terminals to further reduce the search area. The use of this interior frame may prevent the algorithm from finding the shortest path. However, this technique is commonly used to quickly identify simple paths.



(a)



(b)



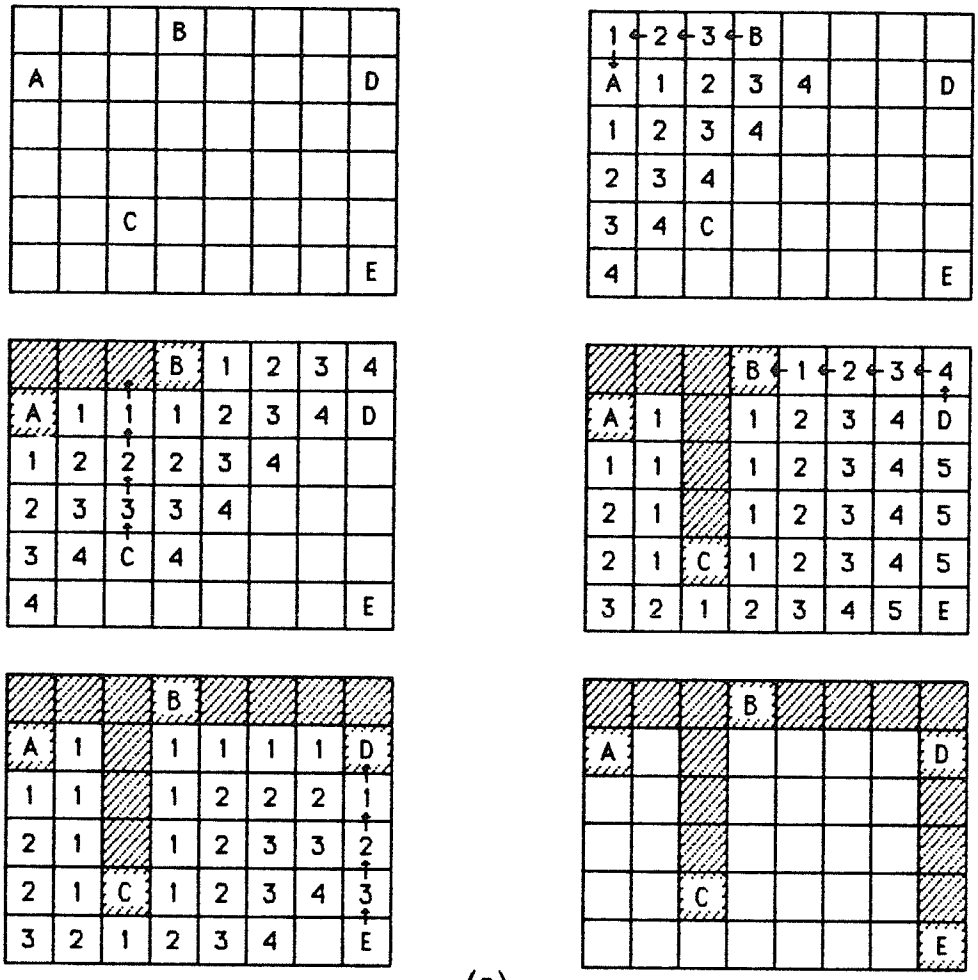
(c)

Fig. 3. Speed up techniques. (a) Starting point selection; (b) Double fanout; (c) Single and double framing.

Multi-Terminal Net Extension

In the original Lee algorithm, only the connection of two terminal nets are considered. When a net with three or more terminals (multi-terminal net) is to be routed, a direct application of the algorithm is not possible. One solution to this problem is to begin by using one of the terminals as the source and the rest of the terminals as the targets, and find a path between the source and the first target reached. Once a path is found, all the cells in the two terminal path become source cells and all the other terminals remain as target cells, and the path finding process is repeated. Then all the cells in the resulting three terminal path become source cells and the other terminals remain as target cells, and so on, until all the terminals are connected. Figure 4a shows the routing of a 5-terminal net.

The interconnecting path obtained by this process is not always the optimum. In fact, this multi-terminal net routing problem is equivalent to the Minimum Steiner Tree problem in graph theory, which is known to be an NP-problem [GAR79]. The possibility of finding the optimal solution in polynomial time is unlikely. There is, however, one simple technique that is often able to improve the resulting path. The idea is to break the resulting multi-terminal path into two sub-paths. Then, one of the sub-paths can be used as the source and the other as the target. If the resulting path is better than the original path, the new path is kept and a better overall path is found. Applying this technique to the example in Fig. 4a, a shorter path is found as shown in Fig. 4b. This technique clearly requires longer running time, but it does often allow better paths to be found.



(a)

(b)

Fig. 4. Multi-terminal net routing example.

(a) Multi-terminal routing using terminal A as the first source cell;

(b) A shorter path found by rerouting sub-paths.

Multi-Layer Extension

One of the most important variations to the original Lee algorithm is the extension to multi-layer regions with interconnecting vias. One way to model the three dimensional routing problem on a grid is to consider a three dimensional array of regular cubes as shown in Fig. 5a, where a double layer case is illustrated. Note that the grid points are now represented by cubic cells. As before, a pair of terminal cells are given, and the goal is to find the shortest path connecting the terminal pair.

The three dimensional Lee algorithm is very similar to the two dimensional case, except that the cells adjacent to a given cell now consist of not only the four planar cells in the north, west, south and east positions, but also the cells in the top and bottom positions. Using similar cell expansion and retracing processes as in the two dimensional case would lead to a path connecting the terminal pair with the minimum number of cells. It is assumed that an inter-layer connection through a via has the same cost as a unit length wire. The result of the example in Fig. 5a is shown in Fig. 5b.

Summary

Maze-running algorithms are capable of finding the optimal path between two terminals with respect to any monotonic cost functions, provided that such a path exists. Numerous variations and extensions have been published to enhance the original maze-running algorithm developed by Lee. Some of the most important ones are the storage reduction techniques, the speed up techniques, the multi-terminal net extension, and the multi-layer extension. But, even with these enhancements maze-running algorithms still require

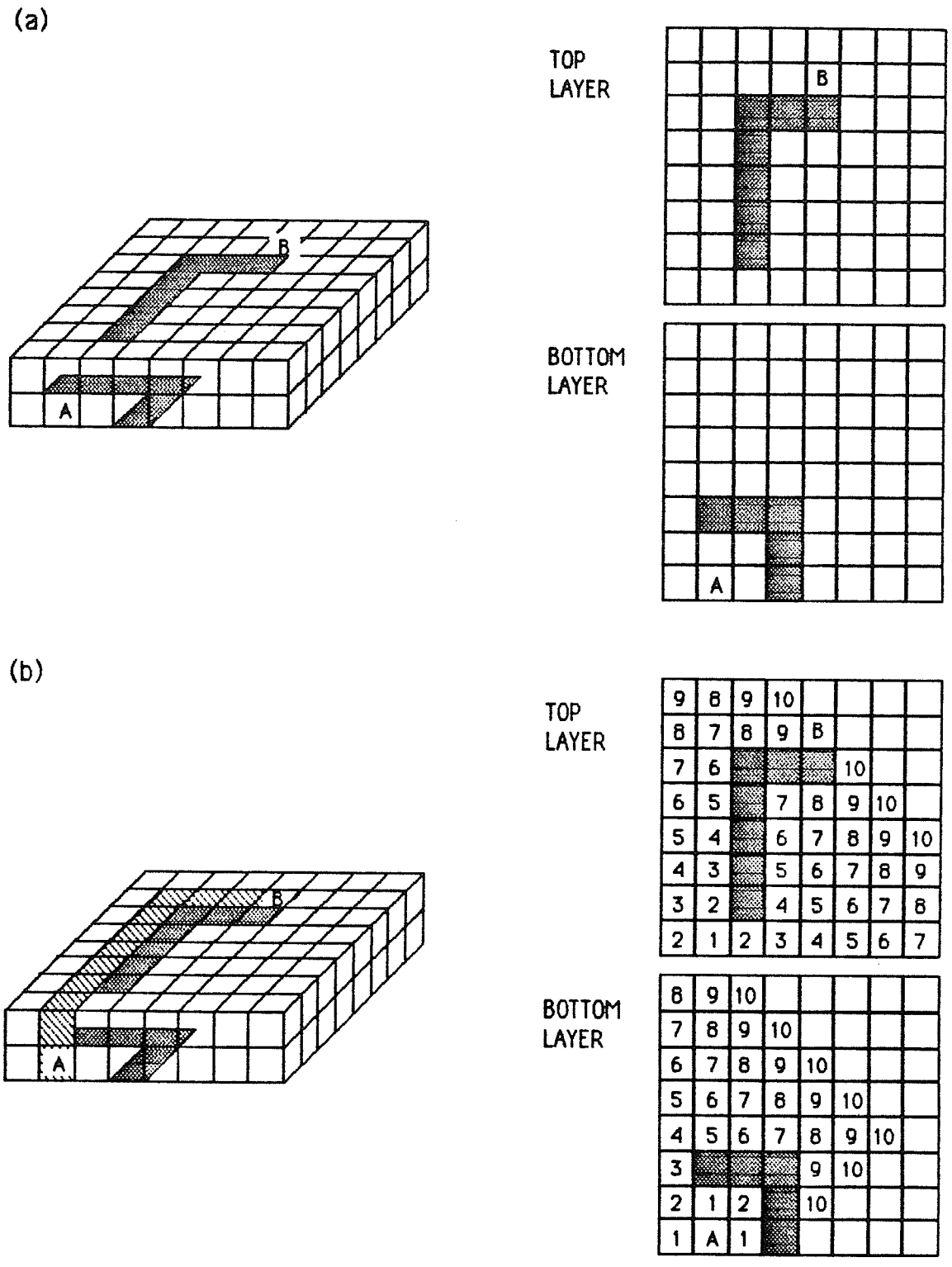


Fig. 5. Multi-layer extension. (a) Double layer routing problem;
(b) Routing result.

substantial amount of storage and running time. To address these drawbacks, the class of line-search algorithm have thus been developed. In the next section, the line-search routing algorithms will be described.

2.1.2 Line-Search Routing Algorithms

The class of line-search routing algorithms was first proposed by Hightower [HIG69] to reduce the storage requirement and to speed up the running time of the maze-running algorithms. Basically, paths are found by constructing sequences of connected line segments starting from the terminal points until they intersect. There are three major differences between the line-search algorithms and the maze-running algorithms.

1. The line-search algorithms also proceed to find a path by running on a grid. But, unlike the maze-running algorithms in which a unit of memory is allocated for each grid point, the routing space is considered as a continuous plane and paths are represented by a set of line segments. In this sense, the line-search algorithms can be viewed as proceeding on an virtual grid.
2. The line-search algorithms process and store line segments rather than cell maps. Thus, in most cases the amount of storage required, and the running time is substantially reduced.
3. The line-search algorithms, although usually able to find a path, cannot guarantee that a path be found even if one exists. Furthermore, the optimality of the resulting path cannot be guaranteed. However, the small storage requirement and the fast running time still make this class of algorithms very attractive.

Hightower Algorithm

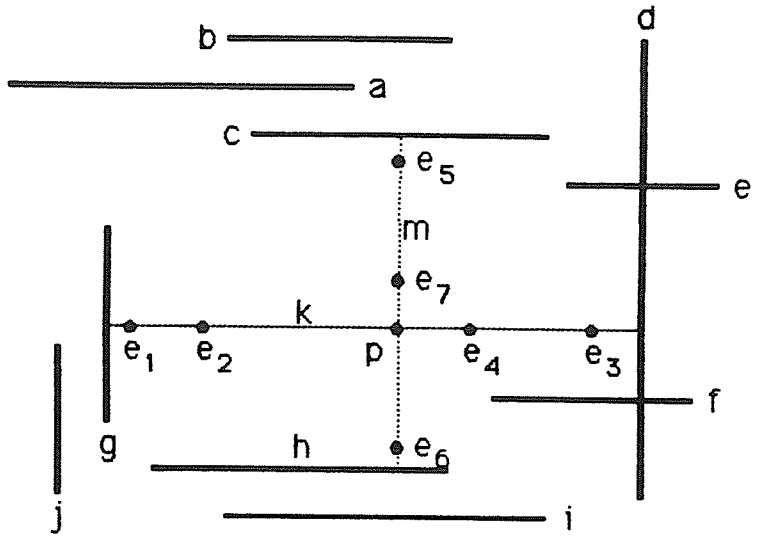
The most representative and best known line-search routing algorithm is due to Hightower [HIG69]. In this section, the Hightower algorithm will be described.

First a few definitions are required before the algorithm cannot be explained.

1. A cover of a point p is a line segment α such that a perpendicular to a passes through p.
2. A horizontal (vertical) cover of p is a cover in the horizontal (vertical) direction such that no other covers of p exist between it and p.
3. A horizontal (vertical) escape line is a horizontal (vertical) line segment through p bounded by the vertical (horizontal) covers of p.
4. An escape point e is a point on the horizontal (vertical) escape line of p which is not covered by both horizontal (vertical) cover of p, or any other horizontal (vertical) line segments between p and the cover.
5. The object point is the escape point currently being processed.
6. The target point is the point to be reached from the object point.
7. A unit is the minimum spacing between wires. It defines the grid spacing.

An illustration of the above definitions is given in Fig. 6.

With the terms defined, consider now the example in Fig. 7. It is required to connect the terminal points A and B in the given routing region with blockage wires a, b, c, d and e. First the escape lines α_0 and α_1 are constructed through the object point A (Fig. 7a). Notice that there are no escape points along the horizontal escape line α_0 since the horizontal covers of A are the top and bottom boundaries. Along α_1, on the other hand, the point



1. Covers of point p: b, c, d, h, i, g.
2. Horizontal covers: c, h.
3. Verical covers: g, d.
4. Horizontal escape line of p: k.
5. Vertical escape line of p: m
6. Horizontal escape points: e_5 and e_6, but not e_7.
7. Vertical escape points: e_1, e_2 and e_4, but not e_3.

Fig. 6. An illustration of definitions used in the Hightower algorithm.

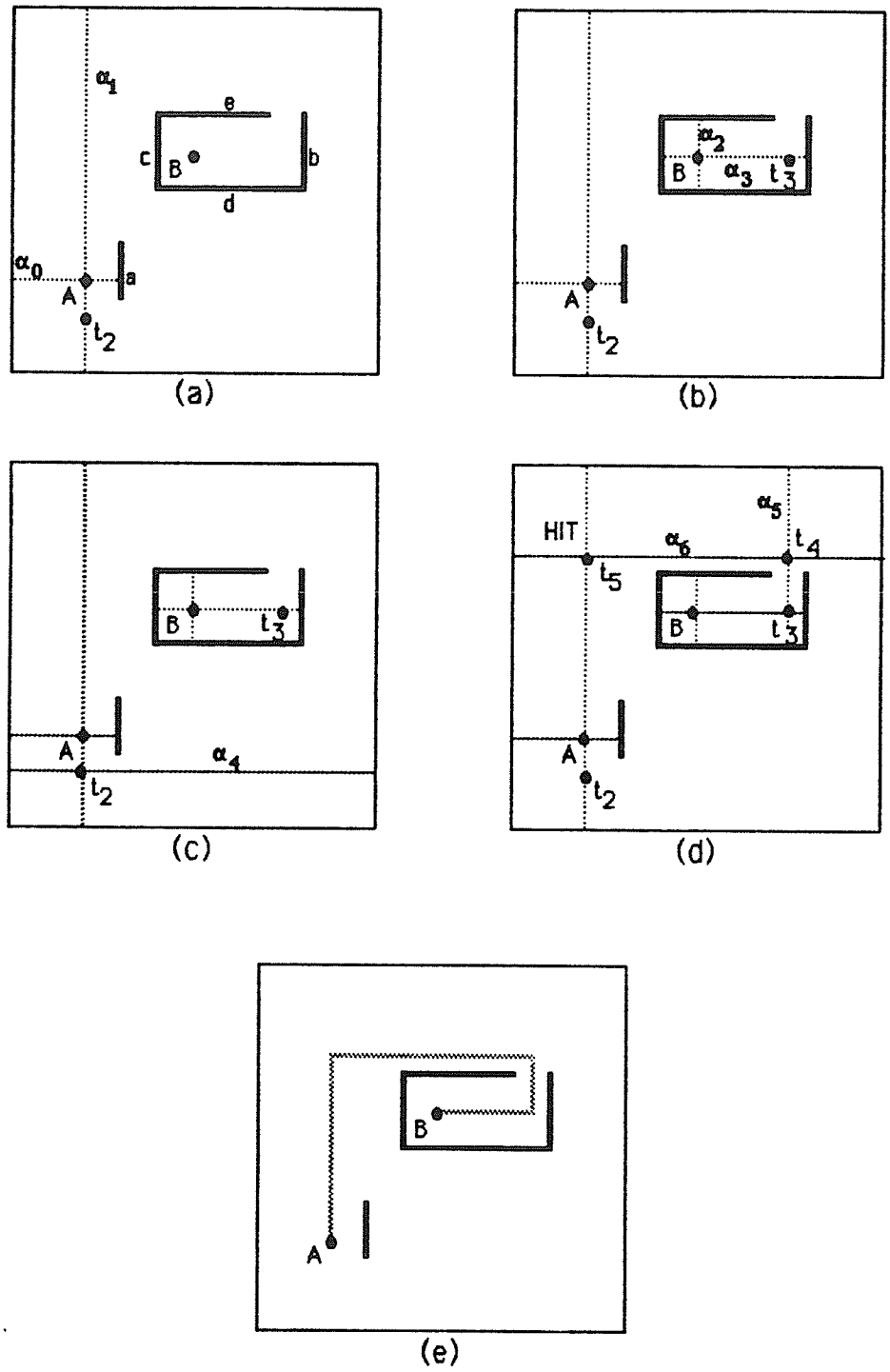


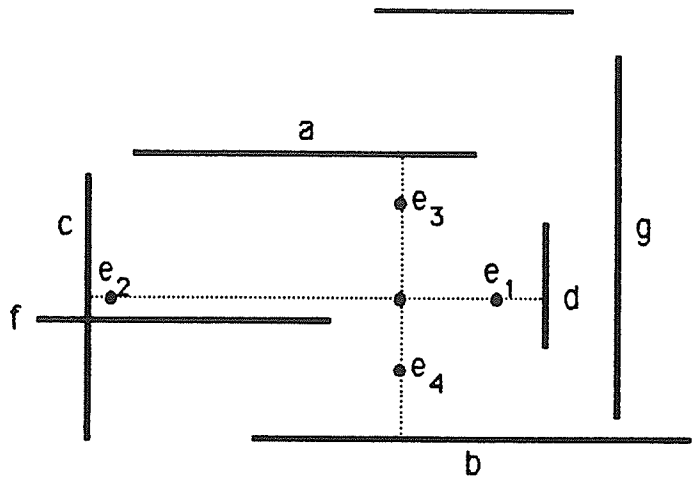
Fig. 7. Hightower algorithm routing example.

(a-d) Escape processes; (e) Final route.

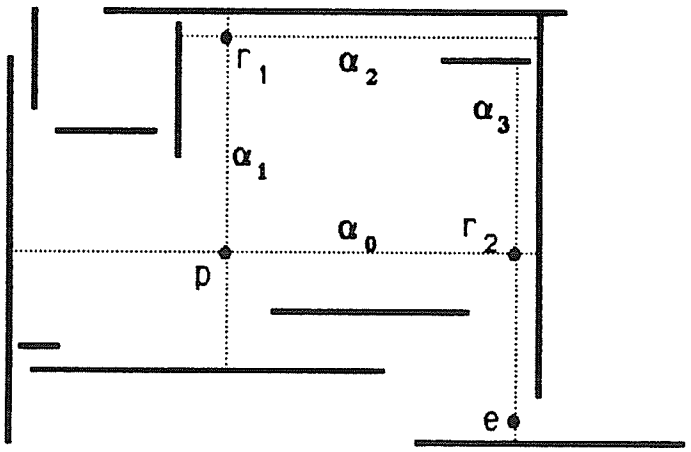
t_2, which is one unit below the bottom end of blockage a is an escape point. Turn now to the other terminal point B, and in a similar fashion the escape point t_3 is found (Fig. 7b). Next, returning to escape point t_2, escape line α_4 is constructed (Fig. 7c). There are, however, no escape points along α_4 since the horizontal covers of t_2 are the top and bottom boundaries. Thus, returning to t_3, the escape line α_5 is constructed, and the escape point t_4 for t_3 with respect to blockage d is identified. Continuing, the escape line α_6 is constructed which intersects escape line α_1 at t_5, and a path is found (Fig. 7d).

From the above example, it can be seen that the novel part of the Hightower algorithm is in the path finding process; which involves two escape processes called the escape process I and the escape process II. An illustration of the escape process I is shown in Fig. 8a. This escape process searches for escape points one unit away from the endpoints of the covers. When more than one escape point is available, the process returns the first one found. In Fig. 8a, points e_1 and e_2 are valid escape points around blockage a, and e_3 and e_4 are valid escape points around blockage d. But, there are no valid escape points around blockage b or c.

Figure 8b illustrates the escape process II, which is used if escape process I fails to find a valid escape point. In this case the procedure searches points, each one unit apart along an escape line through p, starting from a horizontal cover and moving towards p. Through each point an escape line is constructed. Then an escape point is sought along the new escape line using escape process I. If a valid escape point is found the process is completed, otherwise another point is tried until p is reached, in which case no escape from p is possible. In Fig. 8b, first the escape lines α_0 and α_1 are



(a)



(b)

Fig. 8. An illustration of the escape processes.

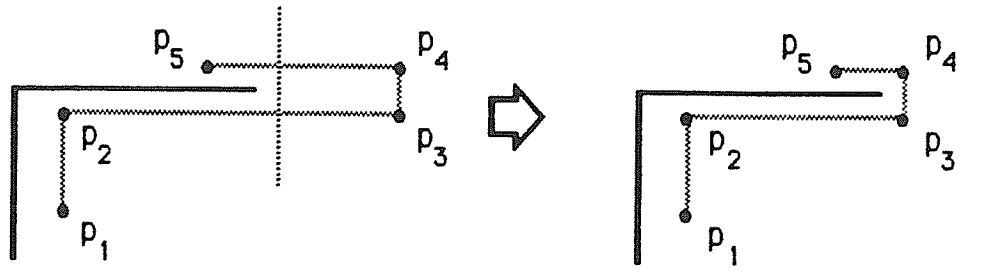
(a) Escape process I; (b) Escape process II.

constructed, and no escape points can be found using escape process I. Then, the escape line α_2 is constructed through r_1 and again no escape points can be found. Finally, the escape line α_3 is constructed through r_2 and the escape point e is found.

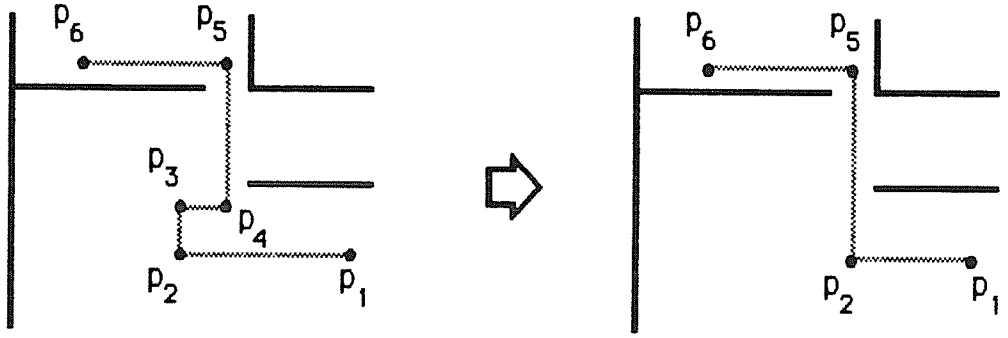
The escape processes alternates between escaping from a point on a path from A and escaping from a point on a path from B. If any escape lines associated with a path from A intersects one associated with B, by tracing backward from this point to the preceding points a path will be established.

Once a path is found refinement procedures can be invoked to improve the shape of the path. One refinement is to delete all escape points not on the corners of the path between A and B. This is necessary because the algorithm, lacking knowledge of the overall problem, tends to generate more escape points than are necessary for a path. Thus, the purpose of this refinement process is to discard those points that are superfluous. After the first refinement process, every point represents a corner in the path. However, there may still be part of the path that is redundant. As an example, consider Fig. 9a. The path shown is much longer than it need to be. A perpendicular to segment (p_2,p_3) and segment (p_4,p_5), which are part of the same path, would shorten the path. Figure 9b illustrates another example, where by extending segment (p_5,p_4) to meet (p_1,p_2), p_4, p_3, and p_2 can be replaced with the intersection of (p_5,p_4) and (p_1,p_2), and a shorter path would result.

The following is the main procedures of the Hightower line-search algorithm defined in pseudo-code.



(a)



(b)

Fig. 9. Two path refinement techniques.

[Hightower (line-search) Routing Algorithm]:

```
Hightower;
    (Variable Definitions)
    stacka:     stack of escape points associated with A;
    stackb:     stack of escape points associated with B;
    object:     object point;
    target:     target point;
    A_path:     flag controlling whether to search from A or from B;
    intersect:  intersection flag;

begin
(initialization)
push A into stacka;
push B into stackb;
A_path := true;
intersect := false;

(main procedure)
repeat
   if no escape from A_path then
      if no escape from not A_path then
         return no path is found;
      else
         begin
         A_path := not A_path;
         continue;
         end;
   else
      begin
      if A_path then
         object := pop stacka;
      else
         object := pop stackb;
      target := not A_path;
```

```
        call escape processes;
        end
until intersect;
call path determining process;
call refinement processes;
return path;
end.
```

□

The memory requirement and time complexity of the Hightower algorithm are unfortunately not described in the original paper [HIG69], and the data structures are only ambiguously described. However, using a linked list data structure, Ohtsuki [OHT86] has analysed the memory requirement and the time complexity of the Hightower algorithm. In the linked list data structure, escape lines and blockages are stored as series of horizontal and vertical line segments. Each line segment is defined by a 3-tuple specifying the x and y coordinates of its two endpoints. For example, a horizontal line segment is represented by (x_1,y_1,x_2) where (x_1,y_1) is the coordinate of the leftmost point on the segment, and x_2 is the x-coordinate of the rightmost point on the segment. With this data structure it is clear that the required memory space is proportional to the number of the escape lines, n. For an N×N unit square routing region, n could be as large as $O(n^2)$, but usually $n \ll N^2$. Therefore, the line-search algorithms still compares favorably to maze-running algorithms in terms of memory requirement.

In terms of time complexity, although the Hightower algorithm generates only one escape line in each step of the line-searching process, many other lines could be investigated in order to choose an escape line. Therefore, it could require $O(n^2)$ time in the worst case. However, the algorithm is usually able to find a path with L line segments in O(nL) time. Thus, the Hightower algorithm runs in time proportional to the number of

corners in a path. If the routing region is not so congested, simple path with small number of corners can usually be found making the line-search algorithms faster compared to maze-running algorithms. But, for complicated mazes, the line-search algorithms do not improve speed so much in contrast to its memory saving.

Line-Expansion Algorithm

Although the Hightower line-search algorithm has been used extensively, especially for the routing of PCBs, no major improvements in the algorithm had been made until Heyns et al. [HEY80] combined the line-search algorithm and the maze-running algorithm into a new algorithm [SOU81, OHT86]. The Heyns routing algorithm expands from a line like the Hightower algorithm, but it fills an area like the Lee algorithm. However, the expanded area is not kept in memory; only its boundary segments are remembered as in the Hightower algorithm. The characteristics of the resulting path are similar to the Hightower algorithm and, in most cases, have the minimum number of corners. This algorithm has several advantages over the original line-search and maze-running algorithms: (i) it finds a path, whenever a path exists; (ii) it is based on a virtual grid that does not restrict the wire width and the path location; (iii) it is fast and requires relatively little memory; (iv) it can use penalty functions similar to those used in maze-running algorithms. It is thus very suitable for routing irregularly structured layouts such as PCBs and VLSI building block designs.

The Heyns algorithm is based on expanding a line in its perpendicular direction. For every grid point (spaced one unit apart) of the line, it is

investigated for how far it can be expanded, that is, how far a perpendicular line through this point can be extended before it is blocked by an obstruction. The expansion zone is defined as the zone consisting of all the grid points that can be reached by a line beginning on the expanded line and perpendicular to it. The idea is that, instead of generating one escape line at a time as the Hightower algorithm, the borders of the zone that can be reached by all possible escape lines are generated. Figure 10 illustrates the expansion zone of a line segment ℓ. The algorithm searches for grid points in the expansion zone using a modified maze-running algorithm. However, the grid points are not kept in memory; only its boundary segments, called active lines, are pushed onto a stack. The generated active lines are then expanded outside the zone for further searches.

The above procedure is initiated from both of the terminals by entering them as starting active lines into the stack. From both of the terminals, wave propagation processes similar to that of the Lee algorithm but advance zone by zone are performed. Figure 11a shows the active lines generated during the search. A connection is found when an active line reaches the wavefront advanced from the other terminal. In Fig. 11a, this occurs in the shaded area, called the solution zone, in which the two wavefronts intersected. The rest of the algorithm is to backtrace from the solution zone towards the terminals, generating a final connecting path as shown in Fig. 11b. An important consideration in the line-expansion algorithm is to generate a stop line when an active line meets the wavefront advanced from the same terminal. The stop line is imposed to prevent duplicated search of a zone. It thus helps to speed up the algorithm.

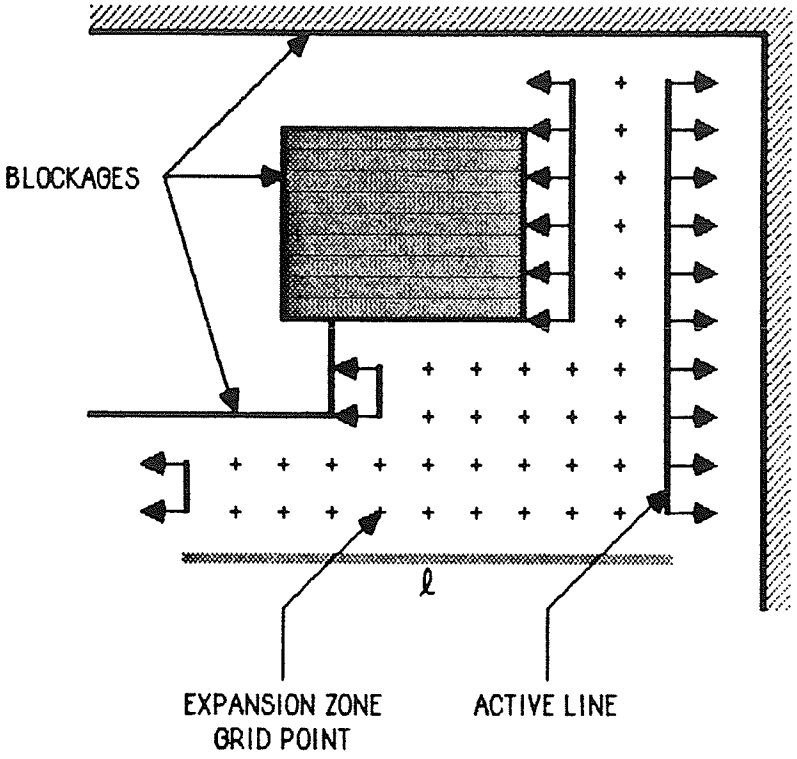
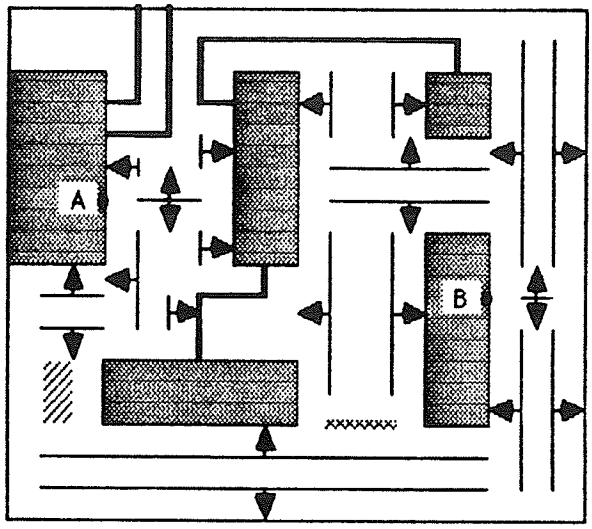
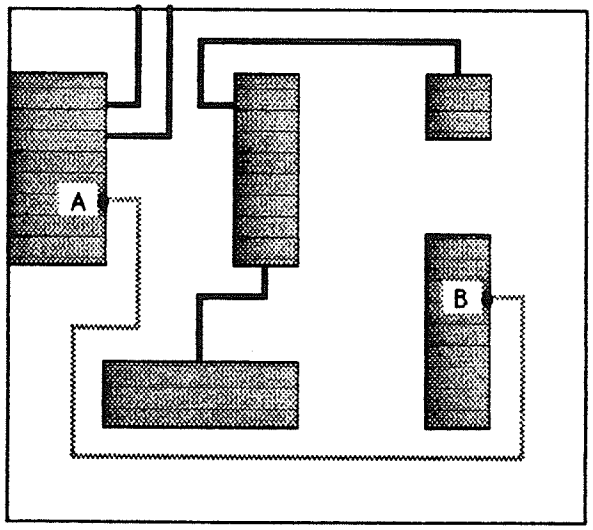


Fig. 10. Expansion of a line ℓ in the upward direction. The arrows indicate the direction in which the active lines are being expanded.



solution zone stop line

(a)



(b)

Fig. 11. Line expansion algorithm. (a) Active lines generated during the search; (b) Final path between A and B.

The line-expansion algorithm compares favorably with the Hightower line-search algorithm in that it guarantees a solution if one exists, and that it can be readily extended to more general cases with multiple terminals, source/target points, layers, etc. A minor drawback, however, is that it does not always find the minimum bend path.

Summary

Line-search algorithms are very efficient techniques for finding a path between two points on a plane with blockages. The main advantages are their low storage and running time requirements. But unlike maze-running algorithms, they cannot guarantee that a path be found, even if one exists, and they cannot guarantee that the path found is the optimal. Although they usually yield the path with the minimum number of corners, it is not always the case. However, line-search algorithms still, in general, run faster and require less storage than maze-running algorithms.

The line-expansion algorithm of Heyns et al. extended the Hightower line-search algorithm by replacing escape lines with expansion zones and by using a maze-running process to expand from zone to zone. It is thus able to find a path whenever a path exists and still has the speed and storage advantages of line-search algorithms.

Aside from speed and storage considerations, both maze-running and line-search algorithms are sequential algorithms that route one net at a time. In such algorithms, when a net is routed it becomes a blockage that may prevent subsequent nets to be routed. Such fragmentation of the routing task could result in poor routing patterns and excessive overflows. To address

this problem, channel routing algorithms have been developed. In the next section, channel routing algorithms will be described in detailed.

2.2 Channel Routing Algorithms

Channel routing is a special case of the general routing problem where interconnections are to be routed within channels. Channels are rectangular routing regions with no interior obstructions and with fixed terminals located on two opposite sides. The interconnections may exit from the channel through floating terminals on the remaining two sides, but the exact location is determined by the router.

Channel routing was first proposed by Hashimoto and Stevens [HAS71]. It was originally used in the design of the ILLIAC IV control unit boards. Since then, channel routing has gained tremendous popularity, particularly in gate array and standard cells layout designs.

The most salient difference between channel routing algorithms and other routing algorithms is their division of the routing process into loose routing and detailed routing. Although simultaneous routing of all the nets is still impossible, such a division of the routing task allows the routing of each net to be influenced by all or part of the other nets. In the following discussion, the loose routing process will be briefly described, while the emphasis will be on the detailed routing process.

2.2.1 Loose Routing

Loose routing (also called global routing) is the preliminary step of the complete routing process. It calls for a routing plan in which each net is assigned to particular regions on the die reserved for routing. The goal is not only to make 100% assignment of nets to regions for the detailed routing process, but also to, for example, minimize wire lengths and die size, and to control routing through narrow or critical channels by routing some nets around to avoid bottlenecks.

After the placement process, the positions of the modules or blocks are defined. Surrounding each module is some extra space for routing. In channel routing, such empty spaces are organized as routing channels. The task of the loose routing process is to determine, for each net, the channels through which the wire segments of a net will be traversing. The operation is called loose routing because it only determines the channels to be traversed without actually fixing the exact position of the wire segments in each channel. In general, the loose routing process consists of the following steps: (i) channel definition, (ii) channel assignment, (iii) routing order determination, and (vi) optimization.

Channel Definition

This step involves the definition of the routing channels. For gate array layouts, since the dies are pre-fabricated up to the metalization stage, the number, the size, and the shape of the channels are all defined in the array architecture. For standard cell layouts, although the number and the size of the routing channels are variable, they are arranged in parallel rows or

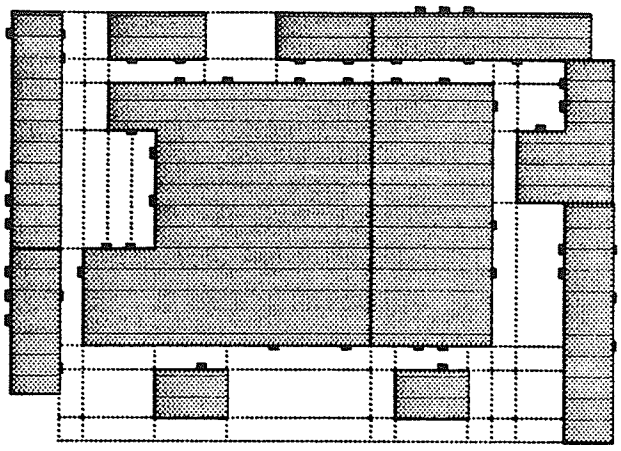
columns. Thus, after placement, the channels for gate array and standard cell layouts are already defined. It is for building block layouts that the channel definition process plays the most important role.

Due to the irregular block geometry of building block layouts, the routing regions are of irregular shapes. In this case, channel routing can be generalized to include, in addition to regular channels, other rectilinear routing regions, such as L-shaped regions, with fixed and/or floating terminals located on all sides. In general, regular channels are considered the most desirable because the regular channel routing problem is relatively well understood and very efficient algorithms have been reported. However, it is often impossible to use only regular channels, and other rectilinear regions are required.

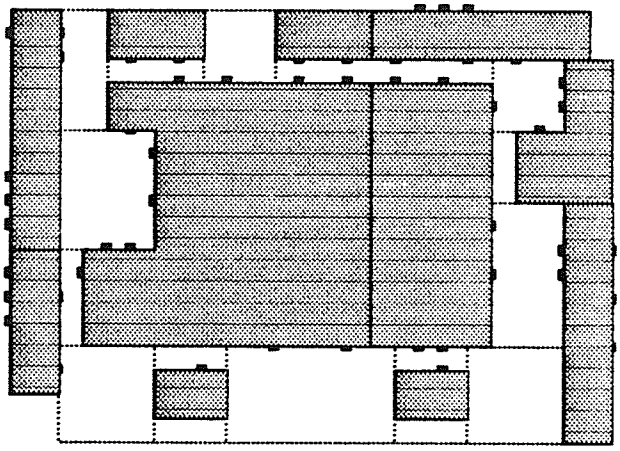
In general, the channel configuration must fit the algorithm for loose routing and provide a fair representation of the routing region for the detailed routing process. The definition of the routing region has a direct impact on all parts of the layout: the adjustment of the placement configuration when routing fails, the data organization, and the algorithms of both loose and detailed routing. Examples showing the results of the channel definition process are shown in Fig. 12.

Channel Assignment

After the channel definition step, the next step is to decide through which channels the wires for each net will be traversing. This process is called channel assignment. The main objective is to assign all the nets to the channels without exceeding the channel capacities. Furthermore, the



(a)



(b)

Fig. 12. Examples of channel definitions. (a) Channels defined by dividing the routing region into small rectangles using the shorter of the two possible edges for each corner (MIT PI System); (b) Channels defined by combining the small rectangles into larger regions (Bell Laboratories).

assignment should keep the wires as short as possible and evenly distributed. The requirement for even wire distribution is important because finding merely the shortest path connecting the terminals of a net tends to overcrowd certain channels, especially those in the center of the die. The overcrowded channels may become very difficult or even impossible to be routed by the detailed router. This would result in excessive overflows and poor layouts.

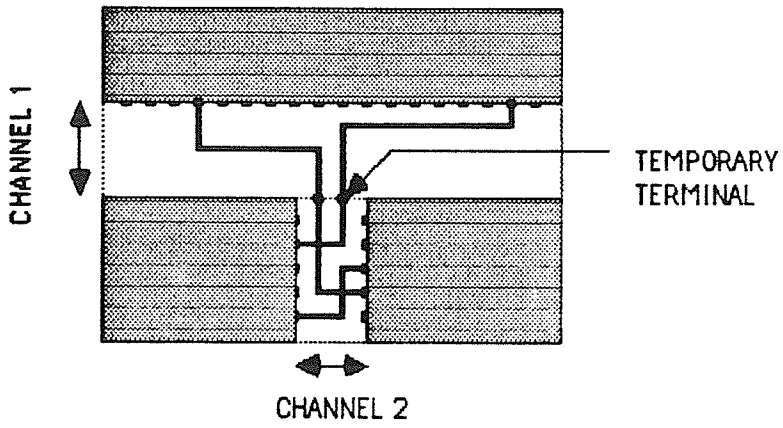
In gate array layouts, since the channel capacities are fixed in the array architecture, if 100% assignment is not achievable the only solution is to use a larger array. In standard cell layouts, the channel width is adjustable. If higher capacity is required of a channel, the channel width can be increased. Thus, 100% assignment is always achievable as long as the die size is allowed to increase. The considerations in such cases would be in the die size, the wire lengths, and the wire distributions. In building block layouts, the blocks and the channels are floating in the sense that their locations are not fixed. Each channel has a certain initial capacity defined by the placement configuration. When nets are assigned to the channels, it may turn out that more space is required. In this case the blocks are pushed apart to make more room. If, on the other hand, less space than originally reserved is required, the blocks are brought closer so that there is no waste space.

Channel assignment can be accomplished by various methods. The sequential method of assigning one net at a time based on the minimum rectilinear Steiner tree is a popular approach. Usually a channel graph is used to represent the loose routing region. For example, the routing area can be divided into a set of routing regions by extending each horizontal and vertical line bounding a block until it intersects another block or the external

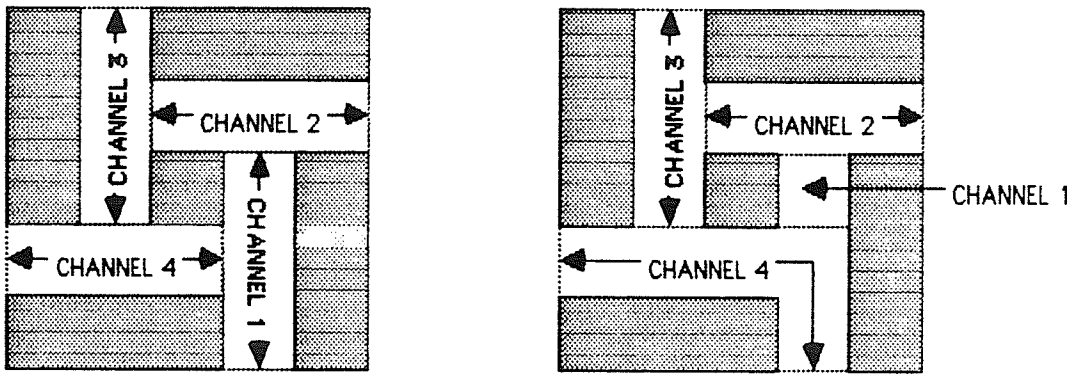
boundary. The routing region interfaces can be represented by vertices, and the routing regions can be represented by edges. Routing conditions such as congestion factors and channel lengths can be represented as edge weights. With the channel graph defined, net assignments can be performed using a variety of algorithms. Since the minimum rectilinear Steiner tree problem is NP-complete, heuristic algorithms are usually used.

Routing Order Determination

This step determines the order of the channels in which detailed routing should be performed [KAJ83, KUH86]. As an example, consider the two channels shown in Fig. 13a. Two nets are to be routed across the channels through the temporary terminals. However, the exact locations of the two temporary terminals are not known until channel 2 is routed. Hence, channel 2 must be routed before channel 1. Such ordering constraints must be determined and resolved before detailed routing can be performed. However, it is possible for a cyclic ordering constraint to occur. For example, in Fig. 13b the channels 1, 2, 3 and 4 are so defined that every channel requires another channel to be routed first. In order to resolve such cyclic constraints, it may be necessary to redefine some of the channels or create new channels. It is also important that the number of complex routing regions be kept to the minimum. In Fig. 13b, a new configuration is formed where a new L-shaped region is defined to break the cycle.



(a)



(b)

Fig. 13. Channel ordering. (a) An example of channel ordering. Channel 2 must be routed before channel 1. (b) An example of redefining a channel (channel 4) to break a cyclic channel ordering constraint. The order should now be channel 1, 2, 3, 4.

Optimization

Due to the sequential nature of most loose routing algorithms, after the initial assignment an optimization process may be used to improve the routing result. Usually an iterative technique is used with progressive penalties in critical channels and dynamic net ordering priorities.

The result of loose routing is a decomposition of the routing problem into smaller detailed routing problems, one for each regular or rectilinear channel. Loose routing is closely tied to both its predecessor, the placement process, and its successor, the detailed routing process. The main objective is to develop a good routing plan based on the given placement configuration so that detailed routing can be completed efficiently. In the next section, a detailed discussion of the detailed routing process will be presented.

2.2.2 Detailed Routing

As described previously, depending on the shape of the channel routing region and the locations of the fixed terminals, the channel routing regions can be regular channels or general rectilinear channels. In this section, the two layer routing problems for these two type of channels will be defined, and for each problem several representative algorithms will be described.

Regular Channel Routing Algorithms

The regular channel routing problem can be defined as follows:

1. A channel is an open-ended rectangular routing region. If a horizontal orientation is assumed, the fixed terminals are located on the top and

bottom boundaries of the channel while the left and right boundaries are open and interconnections may exit. Furthermore, The channel height (distance between the top and bottom boundaries) is indefinitely extendable.

2. The channel has no initial interior obstructions, such as, pre-routed wires or interior routing voids.
3. Routing is performed on a virtual rectilinear grid consisting of vertical and horizontal grid lines called vertical and horizontal tracks, respectively. All wire traces must be routed inside the channel and on the tracks, that is, no routing outside the channel and no diagonal traces.
4. Two routing layers are available with vertical traces exclusively on one layer and horizontal traces exclusively on the other layer. Traces located on different layers are connected by vias.
5. Net terminals are located on the intersections of the top and bottom channel boundaries and the vertical tracks, and are accessible on at least the vertical routing layer.
6. The objective is to route all the nets in minimum channel height, or in other words, in minimum number of horizontal tracks.

An illustration of the regular channel routing model is shown in Fig. 14. It is worth mentioning that although the channel height is allowed to extend indefinitely, certain channels are impossible to route. For example, the channel in Fig. 15a is unroutable even with an arbitrary channel height if no free vertical tracks are available. By adding one extra vertical track, the channel can be routed as shown in Fig. 15b.

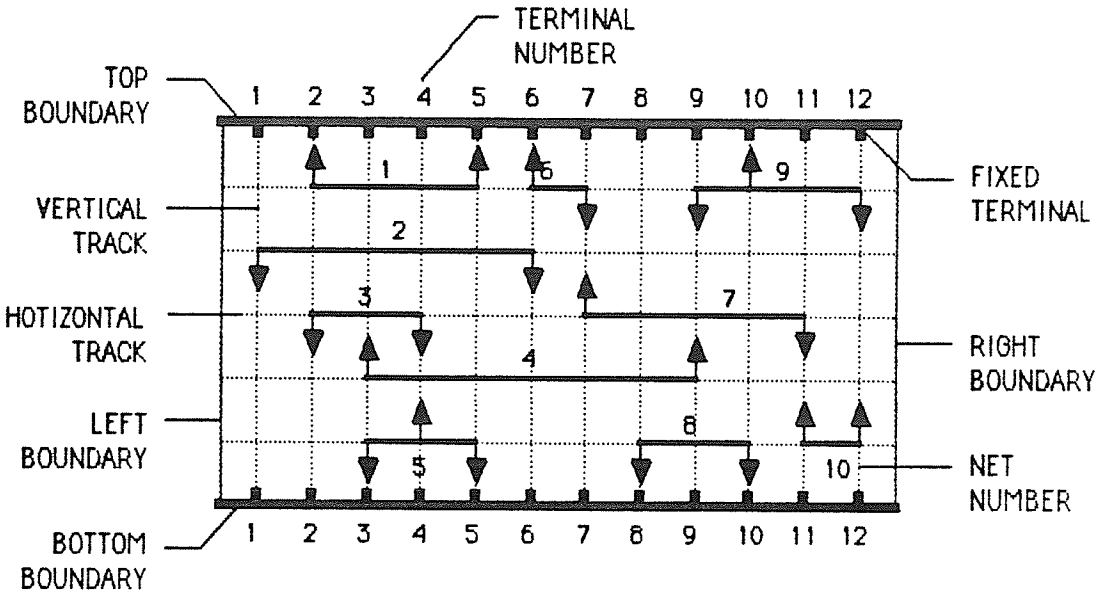
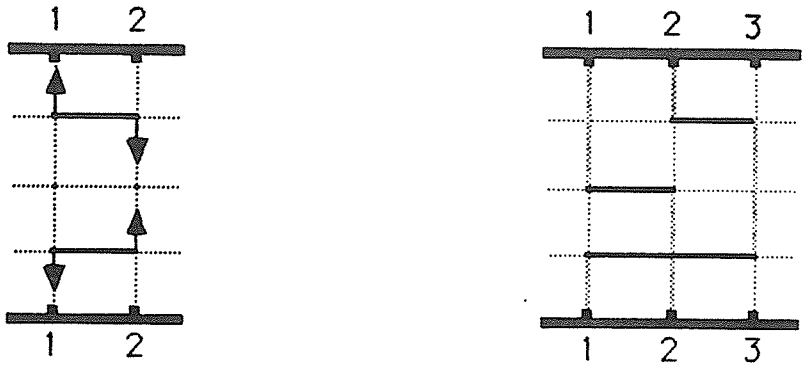
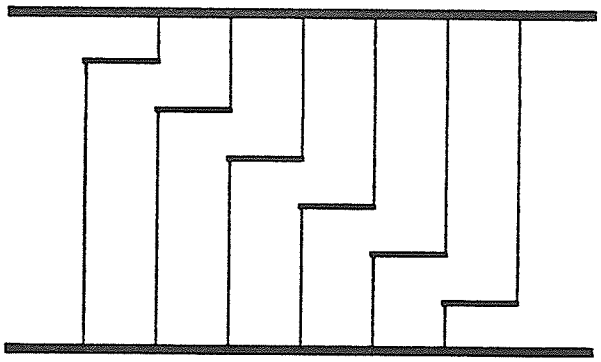


Fig. 14. An illustration of the channel routing model.



(a) (b)



(c)

Fig. 15. Channel routing illustrations. (a) An unroutable channel; (b) Addition of an extra vertical track; (c) A channel routing problem with a unique solution.

Generally, a channel routing solution must satisfy two basic routing constraints: the horizontal and the vertical contraints. A horizontal constraint requires that if two horizontal traces belonging to two different nets lay on the same layer and have one or more vertical tracks in common, they cannot overlap and must be assigned to different horizontal tracks. A vertical constraint, on the other hand, requires that if two vertical traces belonging to two different nets lay in the same vertical track, they cannot overlap and the lower endpoint of the upper vertical trace must be placed in a horizontal track above the upper endpoint of the lower vertical trace.

Since the channel routing problems are NP-complete [LAP80, SZY82, SHI86, KIN87], no algorithm that can guarantee an optimal solution in polynomial time has been found. The only way to produce a guaranteed optimal solution is by enumerative methods such as branch-and-bound techniques. For most but very small cases, enumerative methods require unacceptably long running times. Consequently, a large number of heuristics have been developed with the aim of producing good but not necessarily optimal solutions. Of particular interest is the class of generalized adaptive heuristics described in [KIN87].

To make the channel routing problem more manageable, some routing algorithms further restrict the problem to allow only one horizontal track per net [HAS71, KER73]. That is, no bending of the horizontal traces (doglegging) is allowing. An important advantage with such a restricted model is that the number of vias used is always the minimum. In this section, both algorithms that allow only one horizontal track per net (non-dogleg algorithms), and algorithms that allow multiple horizontal track per net (dogleg algorithms) will be described.

Line Packing/Left Edge Algorithm

If no bending of the horizontal traces or doglegging is allowed, each net can occupy at most one horizontal track, and the wire traces of a net can be view as consisting of one horizontal trace and two or more vertical traces branching off to the terminals. Each net can thus be associated with a lower bound and an upper bound according to the left and the right endpoints of its horizontal trace. The routing problem is now to assign horizontal track spaces to the horizontal traces such that the nets are all electrically isolated and the number of horizontal tracks required is the minimum.

Hashimoto and Stevens described a Line Packing algorithm in [HAS71]. For each horizontal track, the algorithm first searches the list of unrouted nets for the one having the highest upper bound, assigns it to the track, and eliminates it from the list. Then the list is searched for the net having the highest upper bound that is less than the lower bound of the previously routed net. The selected net is assigned to the track and eliminated from the list. The search continues until no net fulfills the requirement. Then the entire process is repeated for the next horizontal track until all the nets are routed.

A more efficient implementation of the Line Packing algorithm is the Left Edge algorithm [KER73]. It first sorts all the nets in ascending order of their left edges (lower bounds) and fills the horizontal tracks with nets that fit closest to the left of the available track space. For a channel with n nets, this algorithm requires O(n log n) running time.

These algorithms assume that there are no vertical constraints in the channel. When they are modified to observe the vertical constraints, they can fail to find the optimal solution even for very simple problems.

Net Merging Algorithm

There are certain instances of the non-dogleg channel routing problem that have obvious unique solutions. For example, the channel in Fig. 15c can only be routed as shown. Moving any one of those nets would result in overlaps in at least one vertical track. The vertical constraints have thus dictated a unique solution. This fact inspired the idea of reconstructing the original problem so that a unique configuration can be identified. The Net Merging algorithm by Yoshimura and Kuh [YOS82] is one such algorithm. The algorithm merges nets that occupy non-overlapping horizontal track spaces together so that their combined horizontal track spaces corresponds to the situation shown in Fig. 15c. Once such situation is established, the channel can be routed easily.

The Net Merging algorithm is usually capable of reducing the original problem to a much smaller size and requires less running time. But if two nets that are far apart were merged, the empty track space between the nets would be wasted. Yoshimura [YOS82, YOS84] suggested heuristics for merging nets together so that the reduced problem would not create situations that require more horizontal tracks than the original problem. In general, the net merging algorithm is capable of producing very good solutions in much shorter time than enumerative algorithms such as branch-and-bound searches.

Dogleg Channel Router

The Dogleg Channel Router of Deutsch [DEU76] was the first routing algorithm that relaxes the restriction of only one horizontal track per net. Here the nets are allowed to split between different horizontal tracks. Vertical

traces are used to connect the split horizontal traces together. The bending of an otherwise straight trace is called doglegging, and the bend is called a dogleg. The introduction of doglegs may allow the channel to be routed in fewer number of horizontal tracks. Moreover, situations as the one shown in Figs. 15a,b would be unroutable without doglegs. However, an immediate drawback of using doglegs is an increased number of vias. A more detailed discussion of the advantages and tradeoffs of using doglegs will be presented in Chapter V on the development of a new dogleg channel routing algorithm.

The idea of the Dogleg Router is to split every net into two-terminal subnets at terminal positions. If a net E has n terminals t_1, t_2, ..., t_n sorted in ascending order of their vertical track numbers, E is split into (n-1) two-terminal subnets E_1, E_2, ..., E_{n-1} such that E_i connects terminals t_i and t_{i+1}, for i = 1, 2, ..., n-1. A modified Left Edge algorithm is then applied to the resulting set of subnets with one modification: a subnet E_i and the next subnet of the same net E_{i+1} can be placed in the same horizontal track sharing a common terminal. When all the subnets are routed, those subnets belonging to same net but placed in different horizontal tracks are connected by doglegs.

The above described algorithm works reasonably well but often adds many more doglegs than it is necessary. Thus Deutsch introduced a control parameter called range to reduce the number of undesirable doglegs. Range is the minimum number of consecutive subnets that must be assigned to the current horizontal track. As the range increases, fewer doglegs will be introduced. But, a sequence of subnets shorter than the range will also be accepted if this will complete the routing of the net. Without this additional rule, two-terminal nets will never be routed.

The efficiency of the Dogleg Channel Router has been demonstrated in [DEU76] on several examples, one of which has become a benchmark test case for regular channel routing algorithms. In most cases, the Dogleg Router is capable of producing optimal, or near optimal solutions that are only a few tracks above the theoretical optimum. In fact, introduction of the Dogleg Router by Deutsch was at the time the breakthrough achievement and inspired further development of powerful heuristics for channel routing problems [BUR86].

Greedy Channel Router

The Greedy Channel Router of Rivest and Fiduccia [RIV82] further relaxes the Deutsch's Dogleg Router to allow doglegging in any vertical tracks not necessarily containing a net terminal. One interesting phenomenon of the Dogleg Router is that, by splitting every net into subnets connecting every two consecutive vertical tracks of the net span and applying the Left Edge algorithm would result in a left-to-right vertical-track-by-vertical-track scan of the entire channel [BUR86].

Using the above idea, the Greedy channel router scans the channel in a left-to-right vertical-track-by-vertical-track manner, completing the connections within a given vertical track before proceeding to the next. In each vertical track the algorithm tries to maximize the utilization of the track spaces in a greedy fashion. It may place a net on more than one horizontal track and have a vertical trace crossing more than one horizontal trace of the same net. Usually, the Greedy router is able to complete the routing within the channel, but sometimes it may requires vertical tracks be

added beyond the channel length to complete the problem.

The algorithm begins with a channel height equals to the theoretical minimum and adds more horizontal tracks when necessary. Following [RIV82], the Greedy Channel Router can be described as follows:

1. Bring the terminal connections from the top and bottom boundaries of the channel into the first horizontal track that is either empty or already contained a trace from the same net. If bringing in the nets would result in an overlap of their vertical traces, bring in just the one with a shorter trace. If the top and bottom terminals are from the same net, simple connect them with a vertical trace. If all the horizontal tracks are occupied, nothing is done in this stage.
2. Free up as many horizontal tracks as possible by adding doglegs that collapse nets occupying more than one horizontal track into one for the next vertical track. An exhaustive search is performed to find the best admissible pattern. This step may extends the vertical traces in step 1 from an intermediate empty horizontal track to a horizontal track containing the net.
3. Add doglegs to the remaining nets that are occupying more than one horizontal track to reduce the distance between their horizontal traces.
4. Add doglegs to move a net closer to the top if its next terminal is on the top of the channel, or closer to the bottom if its next terminal is on the bottom of the channel.
5. If a terminal could not be brought into the channel in step 1 because the channel was full, widen the channel by inserting an additional horizontal track in the center of the channel.
6. Extend the incomplete nets into the next vertical track. For each

connected piece of wire segment one horizontal trace is extended into the next vertical track. The above procedure is repeated for each vertical track until all the nets are routed.

The Greedy Router algorithm has been tested on numerous channels from actual designs and computer generated test cases [RIV82]. In most cases, the algorithm completes the problem using no more than one horizontal track above the theoretical minimum. In addition to this excellent performance, the algorithm has a very flexible control structure that allows variations in the heuristics to achieve different routing effects.

YACR-II

YACR-II (Yet Another Channel Router the Second) by Sangiovanni-Vincentelli [SAN84] is aimed at minimizing not only the channel height but also the number of vias required. YACR-II is basically a double-layer, grid based regular channel router. It normally uses one of the routing layers for vertical traces and the other for horizontal traces. But it also allows certain horizontal traces to be routed on the vertical layer.

Similar to the Greedy Router, YACR-II begins with a channel height equals to the theoretical minimum and inserts additional tracks when necessary. The basic idea is to place first all the nets in the channel with no doglegs, avoiding any horizontal overlaps while ignoring the vertical constraints. Then maze-running routers are used to complete the routing by connecting the terminals to the nets with doglegs. If the maze-running routers cannot connect some terminals, the number of horizontal tracks is increased by one and the entire process is repeated until all the nets are

routed. The algorithm consists of the following four phases:

1. Scan the channel from left to right to find the first vertical track that has the highest local density. The local density of a vertical track is the number of nets crossing that vertical track. All the nets that crossing that vertical track are then assigned to horizontal tracks so that (i) horizontal constraints are satisfied, (ii) vertical constraint violations are minimized, and (iii) resulting net assignments would facilitate the maze-running routing process.
2. Move to the right of the selected vertical track and assign the nets having their left endpoints in this part of the channel to horizontal tracks. While moving to the right, one vertical track at a time, the nets with their left endpoints in that vertical track are collected. The collected nets are assigned to horizontal tracks when (i) the number of available horizontal tracks is equal to the number of nets collected, or (ii) one of the nets collected at previous steps has its right endpoint in the vertical track currently processing. The assignment is performed according to the same criteria as in step 1. This step terminates when the end of the channel is reached.
3. This step processes the nets to the left of the initial vertical track selected, and is identical to step 2 except that the scanning is now towards the left.
4. Connect the terminals to the routed horizontal traces. For vertical tracks with no vertical constraint violations, no doglegs are necessary and the routing is straightforward. For vertical tracks with vertical constraint violations, a sequence of three maze-running routers are used. They attempt to complete the routing using doglegs with as few bends and vias as possible. The first router routes the terminal

connections using only the adjacent vertical tracks and the vertical layer. If the first router fails to complete the routing, the second router uses doglegs that span more than one vertical track. If the second router also fails, the third router attempts to complete the routing using all available space.

Excellent results have been obtained by YACR-II [SAN84, REE85]. In most cases, YACR-II is able to route a channel in a height equals to the minimum height attainable by classical routers. However, the YACR-II algorithm relaxed the classical channel routing model to allow horizontal doglegging on the vertical layer. This may results in wires running parallel on different layers, and the minimum channel height for the YACR-II channel routing model is no longer the same as that for the classical model. Nevertheless, since most technologies permit limited wire overlaps, YACR-II is a very fast and practical channel router.

Rectilinear Channel Routing Algorithms

The rectilinear channel routing problem can be defined as follows:

1. A rectilinear channel is a rectilinear routing region with fixed terminals located on all boundaries and has no initial interior obstructions such as pre-routed wires or interior routing voids.
2. Two routing layers are available with vertical traces exclusively on one layer and horizontal traces exclusively on the other. Traces located on different layers are connected by vias.
3. Routing is performed on a virtual grid. All wire traces must be routed inside the channel and on the grid lines. Net terminals are located on

the intersections of the channel boundaries and the grid lines, and are accessible on both routing layers.

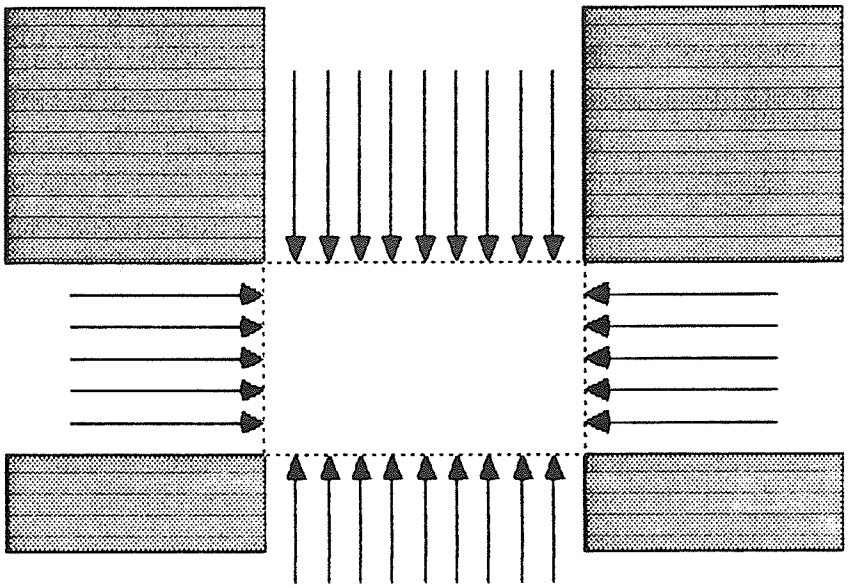
4. The objective is to route all the nets within the channel without violating any electrical or physical constraints.

Since the routing area is fixed, not all channels are routable. Pre-routing analysis similar to those used in PCB routing are commonly used to predict the routability of rectilinear channels. If a low routability is predicted, the routing area can be enlarged by modifying the placement configuration.

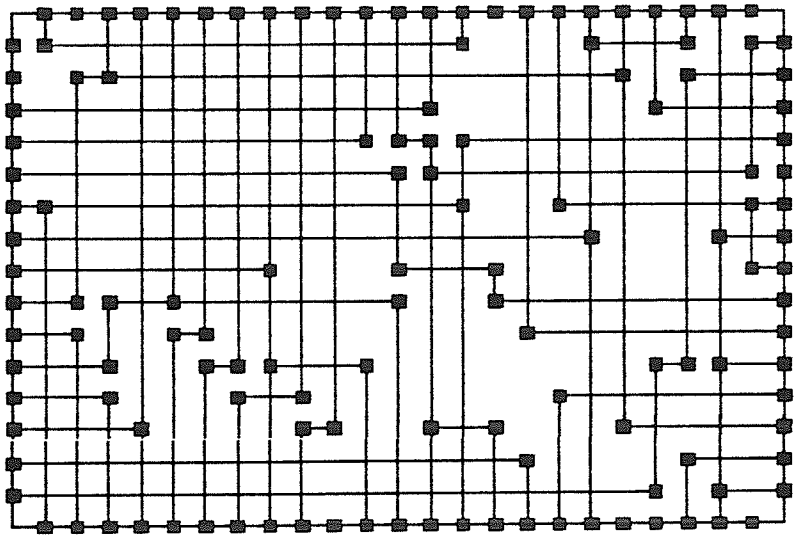
A special rectilinear channel that occurs very often in layout designs is the switchbox. A switchbox is a rectangular region with fixed terminals located on all four sides. It is commonly used at the interfaces of regular channels. Next to regular channels, switchboxes are the most desirable. However, in building block layout designs, there are often cases where the routing regions are so irregularly shaped that decomposing them into regular channels and switchboxes may not be practical. A number of routing algorithm have been published to deal with such general rectilinear channels. In this section, both switchbox routing and general rectilinear channel routing will be described.

Switchbox Routing

Figure 16 shows an example of a switchbox used at the interface of four regular channels and the routing of a switchbox. The switchbox routing problem is in general more difficult than the regular channel routing problem. The main reason being that, it is not clear as what to do in case of failures.



(a)



(b)

Fig. 16. Switchbox routing. (a) A switchbox used at the interface of regular channels; (b) An example of switchbox routing (Burstein's Difficult switchbox problem).

Unlike regular channel routing where the channel height can be increased, the routing area of a switchbox is fixed and no additional routing tracks can be inserted. Although the switchbox routing problem has not been proven NP-complete, it is very likely that it is [BUR86]. A number of heuristic algorithms have been developed for this problem. In this section the switchbox router, Detour, will be described.

Detour

Detour [HAM84] is a switchbox router based on the Greedy Channel Router of [RIV82]. It is capable of routing switchboxes and regular channels containing interior obstructions. It can dogleg nets around multi-layer obstructions such as vias, and route nets over single layer obstructions such as pre-routed wire traces. This ability to handle interior obstructions is one of the most remarkable advantage of the Detour switchbox router.

Like the original Greedy Router, Detour uses a heuristic based, left-to-right vertical track scan approach. In order to route switchboxes, Detour uses two strategies, one to allow nets to move into the tracks they need to make terminal connections, and another to allow nets to split in order to make multiple connections at the far edge of the switchbox. The basic steps of the Detour switchbox router are as follows:

1. As the first step in routing a vertical track, place a via in each unobstructed horizontal track if either the previous or the next vertical track has an obstruction in the horizontal layer. The via serves one of three purposes: (i) it switches the net from the horizontal layer to the vertical layer before the net enters an obstructed region, (ii) it

switches the net from the vertical layer back to the horizontal layer after the net leaves the obstructed region, or (iii) it switches the net to the vertical layer in preparation for doglegging the net to another horizontal track.

2. Bring the terminal connections from the top and bottom boundaries of the switchbox into the first horizontal track that is not blocked in the next vertical track. If the net is brought into an obstructed horizontal track, the next step will attempt to dogleg the net out of the obstructed region.
3. Find horizontal traces in obstructed regions and dogleg them to the nearest empty horizontal track while giving preference to doglegging towards the next terminal.
4. Free up as many horizontal tracks as possible by adding doglegs that collapse nets occupying more than one horizontal track into one for the next vertical track.
5. Add doglegs to the remaining nets that are occupying more than one horizontal track to reduce the distance between their horizontal traces.
6. Add doglegs to move a net closer to the top if its next terminal is on the top of the switchbox, or closer to the bottom if its next terminal is on the bottom of the switchbox. Do not move a net into an obstructed horizontal track.
7. If the net is within a certain number of vertical tracks of the right boundary, attempt to split the nets to make multiple terminal connections.
8. Extend the incomplete nets into the next vertical track. For each connected piece of wire segment one horizontal trace is extended into the next vertical track. The above procedure is repeated for each

vertical track until all the nets are routed, or a net is prevented from extending into the next vertical track by the presence of a multi-layer obstruction, in which case the algorithm terminates with no solution.

The Detour has been considered to be one of the best practical switchbox routers [BUR86]. Its performance in routing benchmark test cases, though not the best, was satisfactory [HAM84] while its most distinctive advantage is the ability to handle interior obstructions such as pre-routed wires. This obstruction avoidance capability gives designers the option of pre-routing special nets such as clock, power and ground lines, either manually or using special purpose routers.

General Rectilinear Channel Routing

The general rectilinear channel routing problem arises commonly in building block layout designs, where the routing regions are irregular in shape and terminals are located on all sides of the regions. An example of general rectilinear channels is shown in Fig. 17. The general rectilinear channel routing problem is usually more difficult than the switchbox routing problem. Like the switchbox routing problem, the general rectilinear channel routing problem has not been proven NP-complete, but it is also very likely to be [BUR86]. In this section, the MIGHTY router will be described.

MIGHTY

The MIGHTY router of Shin and Sangiovanni-Vincentelli [SHI86] is a general rectilinear channel router that routes incrementally the nets in the channel,

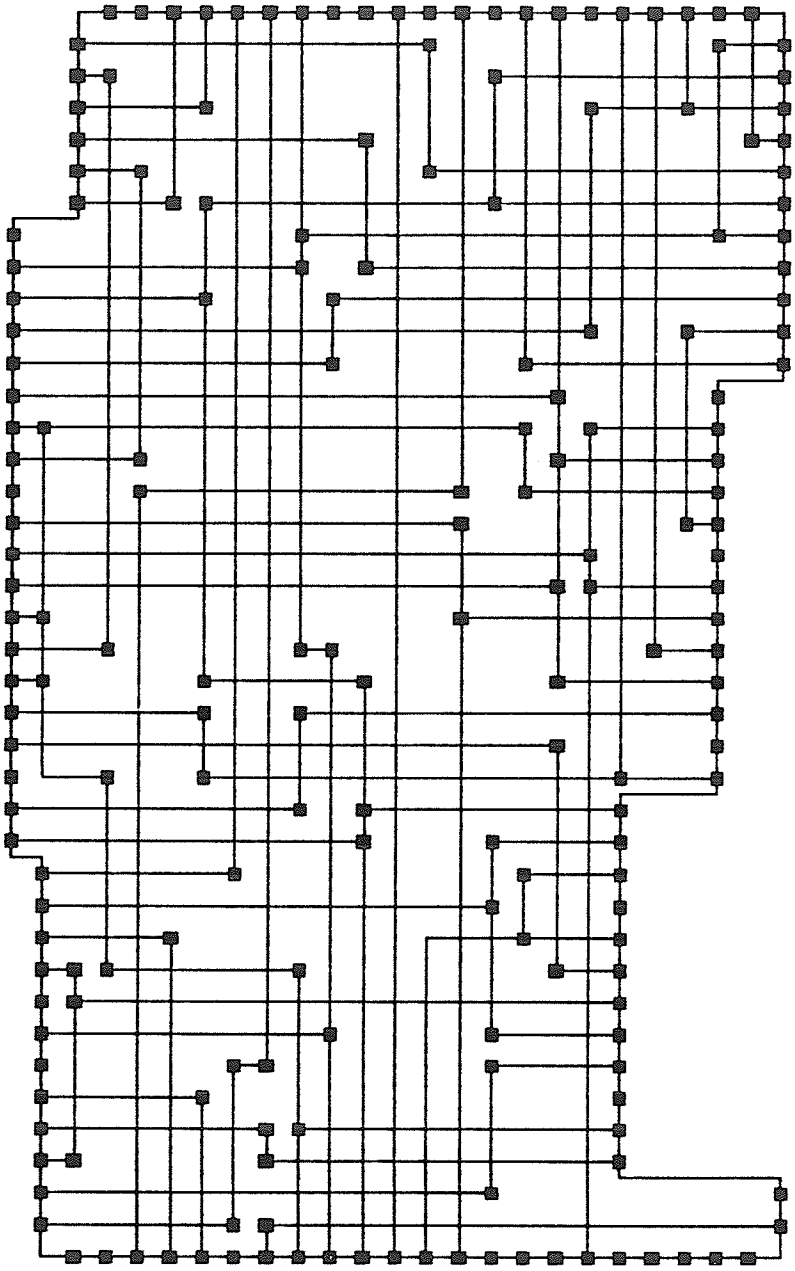


Fig. 17. General rectilinear channel routing [HSU82].

and allows modification and rip-up of nets that may impede the routing of other nets. It consists of four main parts: (i) a path finder that searches for minimum cost paths among subnets, (ii) a path conformer that implements a path proposed by the path finder, (iii) a weak modifier that pushes existing wire segments aside to make better connections, and (iv) a strong modifier that removes subnets to allow the completion of a blocked net.

The algorithm begins by extending all the terminals on the boundaries of the channel inside. The path finding phase then processes the nets in the order they are entered. From each terminal of the net, a maze-running router is used to search for the minimum cost path that connects two of the terminals of the net while ignoring other nets. As soon as a path connecting two terminals of the net is found the search is stopped, and the path is recorded in an ordered list organized in increasing cost. When all the nets have been processed by the path finder, the path conformer takes over. One by one, the paths in the ordered list are examined. If a path does not have any conflicts with the existing paths, the path is implemented. Otherwise, the path finder is invoked again to find a feasible minimum cost path between any two unconnected terminals or subnets of the net. Note that in finding this path, the nets already routed are taken into consideration. When a path is found, if its cost is within a certain prescribed range from the minimum cost of the path found when no other nets are present in the region, the path is entered into the ordered list. Otherwise, the modification phase is entered. The weak modifier is first called to move other nets around to make a feasible path that satisfies the prescribed cost range. If no solution is found, the strong modifier is called to remove some routed wire segments. In both modification phases, a variety of alternate paths are examined and the one

with the minimum cost is selected. If no path satisfying the prescribed cost range is found, the search is terminated with no solution; otherwise the process is repeated for the next net in the list.

The MIGHTY router has shown excellent results in a number of benchmark test cases [SHI86]. But, it generally requires much more running time. For example, in one of the benchmark test case, the Detour switchbox router took 3.9 seconds while MIGHTY required 176 seconds. However, the versatility and performance of the MIGHTY router has made it one of the best general rectilinear channel router today.

2.2.3 Summary

Routing of high-density chips and boards can be divided into two stages: loose routing and detailed routing. The loose routing stage relies on a global routing plan that partitions the routing problem into smaller detailed routing problems. Then the detailed routing stage assigns locally the locations of the wire traces inside the channels. This approach greatly simplifies the complex routing problem into manageable subproblems while maintaining a very high level of global efficiency.

An extensive survey of channel routing algorithms from simple non-dogleg regular channel routing algorithms to elaborate rip-up and re-route general rectilinear channel routing algorithms has been presented in this section to show the most important achievements in channel routing theory and algorithms. An understanding of those developments is essential to any further studies of the channel routing problem. In the next section, other approaches to the VLSI routing problem will be described.

2.3 Other VLSI Routing Approaches

As noted in the previous discussions, the VLSI routing problems are known to be NP-complete. The optimal solutions to these problems require running times that could grow exponentially with the size of the problems. Practical algorithms therefore use heuristic techniques with polynomial complexity that lead to near optimal solutions. Unfortunately, to obtain even those suboptimal solutions would still require tremendous computational effort. With the extreme complexity of VLSI routing, it is not unusual that a small number of overflows occurs. But, even 1% of overflows for a ten thousand gate layout amounts to hundreds of interconnections. Manually editing existing wires and routing overflows may takes days, weeks, or even months. Iterative algorithms such as rip-up and re-route techniques require much longer running time than conventional algorithms; still they may not be able to eliminate overflows. Many other techniques are thus studied with the aim of more efficient routing techniques. In this section, hardware routers, expert routers and the simulated annealing technique will be described.

2.3.1 Hardware Routers

Most hardware routers are parallel implementations of maze-running algorithms. The processing elements are usually arranged in a mesh with one element per cell. Basically, each element should be capable of performing the following maze-running operations in parallel:

1. Receive wavefront tokens from neighbouring elements, if any.
2. Ignore the token if the cell is occupied or visited.
3. Mark the cell visited and mark the direction from which the token is received.

4. If the cell is the target, signal completion; otherwise send wavefront token to all four neighbouring elements simultaneously.

CLearly, all cells in the wavefront during propagation can be processed in parallel. This is true whether the source is a single cell or a set of cells in a subnet. The routers are capable of processing an entire wavefront of propagation simultaneously. Thus, the parallel propagation time is proportional to the path length.

Breuer and Shamsa's L-machine [BRE80] is the first published design of this nature. The L-machine consists of a control unit that communicates with the host computer, and sequences the operations of the array of processing elements, called L-cells. It is designed specifically to implement the Lee algorithm. The L-cells are fairly simple (about 75 gates), and many of them can be laid out on a VLSI forming a subarray. The machine is capable of performing the following tasks:

1. Initialization: This involves the loading of source, target, and blockage information into the L-cells.
2. Parallel Propagation: This essentially implements the parallel cell processing elements described earlier. In addition, a BUSY status signal is sent by those L-cells that are active during an expansion step. The control unit receives a wired-OR BUSY signal from all the L-cells. Thus, the BUSY signal is high as long as some cells are active. If the control unit detects that the BUSY signal goes low before the target is reached, an overflow is indicated.
3. Backtrace: This process determines the path by following the stored direction flags from an activated target back to the source. The coordinates of each element on the path of the wire are output to the

host computer.

4. Clear: Cells along the backtraced path are marked as blockages for subsequent routing. The internal status of all other L-cells are cleared to an idle state.

Each L-cell communicates with its four neighbours through bidirectional lines, one per neighbour. For double layer routing, each L-cell would have five neighbours and five connection lines. These lines are used during the propagation and the backtrace processes. In addition, there are seven more signal lines and a clock input line per cell.

In general, for each routing layer of size N×N, it requires N^2 L-cells, each of which consists of about 75 gates. If the array were to be implemented in VLSI chips, the total number of pins of the array including two power lines and the row and column decoder and encoder is $4\log_2 N + 8$. Thus a 64×64 array would require 300K gates and 32 pins, and a 256×256 array would require 5M gates and 40 pins. Since the array size of the L-machine must be as large as the routing region (multiplied by the number of layers for multi-layer routing), for even a moderately sized layout the number of L-cells required could still be prohibitively large. However, since the L-cells are hardwired for the Lee algorithm, it takes only one clock cycle to process one complete wavefront. In general, the L-machine is much faster than conventional algorithmic routers; but, it is inflexible and limited to finding the shortest path between two points only.

In addition to the L-machine, the SAM (Synchronous Active Memory) machine proposed by Blank, Stefik and van Cleemput is also aimed at a compact design suitable for subarray packing in a VLSI chip. One main

difference between this machine and the L-machine is that this was designed with a somewhat broader range of applications in mind. The node processors are called SAM-cells. They support 20 assembly level instructions operating mainly on the data width of 1 bit. Each SAM-cell consists of a local control unit, a 2-bit Boolean logic unit, a multiplexer feeding a 1-bit accumulator, a neighbour masking unit, and sixteen 1-bit registers. One of the most powerful SAM instructions is called NEIB. This instruction enables a fast implementation of node processing for the Lee algorithm.

The SAM-machine is a SIMD (Single Instruction Multiple Data) parallel processing construct. The program control and storage can be provided either by the host computer or by the SAM system depending on implementation. Blank et al. proposed packaging a 16×16subarray of SAM-cells in a VLSI. To process a 1000×1000grid, it would require about 4000 such ICs in the SAM system. Unlike the L-machine where the L-cells are hardwired for the Lee algorithm, the SAM-machine now takes many clock cycles to process one wavefront.

Conceptually, the SAM machine can be used for general purpose applications. However, the limited instruction set, small data width (1 bit), and SIMD operation together restrict the effective range of applications to bit-map problems arose in certain image processing, bit-vector operations, and simple design rule checking.

In summary, hardware routers gain speed over conventional routers by special purpose hardware and by use of parallel computation. High speed routing in VLSI would allow fast feedback to the designers and even enable the designer to interactively improve the design through a series of

applications. However, like other parallel processing systems, much of the issues concerning data width, instruction capability, neighbour communication, MIMD/SIMD, and local memory organization need further research and development.

2.3.2 Expert Routers

In addition to straightforward parallel hardware implementation of conventional routing algorithms, another approach to the VLSI routing problem is through the use of expert systems. It has been observed that a human layout expert can actually perform wire routing better than conventional algorithmic routers. It is because an experienced designer can understand the design and find a solution using his knowledge and intuitions while an algorithmic router with a small number of heuristics finds a solution without a complete knowledge of the entire problem. A number of expert routers have been proposed [MIT84, JOS85, JOO85], among which the most successful one is the WEAVER channel/switchbox knowledge based expert router by Joobbani and Siewiorek [JOO85]. In this section, the WEAVER expert router will be briefly described.

WEAVER is a knowledge based channel/switchbox routing program that considers several important routing criteria such as 100% completion, minimum routing area, minimum wire length, and minimum number of vias simultaneously. WEAVER is a grid based router that utilizes two routing layers and can be extended to route regions of any shapes. It allows pre-routed wires, and user interaction throughout the entire routing process.

WEAVER uses a set of knowledge based experts organized around a communicating medium called a blackboard. Each expert decides, based on its knowledge and metric criteria, what should be done next. A focus of attention module decides which expert should be allowed to give advice at a given time. The WEAVER experts consists of the following experts:

1. Constraint Propagation Expert: This expert is the most frequently used expert. It propagates the constraints resulted from the routing of the current net to totally or partially adjust the routing of the other nets.
2. Wire Length Expert: This expert decides which nets should be routed closer to which side of the channel based on the minimum wire length criterion.
3. Vertical/Horizontal Constraint Expert: This expert decides the ordering of the nets from bottom to top or from left to right of the channel based on the vertical and horizontal constraints.
4. Merging Expert: This expert decides which nets can be routed on the same row or column.
5. Congestion Expert: A congestion factor is defined for each row and column in the channel which is equal to the number of nets crossing that row or column. This expert restricts each net to cross the most congested area of the channel at most once.
6. Common Sense Expert: This expert uses common rule of thumbs employed by human experts when there are no clear best choice based on the advice of the other experts.
7. Focus of Attention Expert: This expert decides, based on the current active expert and the decision arrived at by the active expert, which expert should be activated next. It maintains a priority list for the experts.

In general, WEAVER is a complex knowledge based expert system utilizing a total of 436 rules. It usually requires much longer running times than algorithmic routers. For example, the Burstein's switchbox benchmark case required 3933 rule applications (rule firings) and 1390 seconds of processing time. However, WEAVER has many advantages that are not easily achieved with algorithmic routers: (i) WEAVER is fairly general and can be easily extended to route regions of any shapes; (ii) while most conventional routers consider only one or two routing criteria, WEAVER considers simultaneously the routing area, the completion rate, the wire lengths, and the number of vias; (iii) although WEAVER is a double-layer router, it uses both routing layers for all directions, thus allowing critical nets to be routed on a single layer avoiding the use of vias; and (iv) since human designers are the best expert, WEAVER allows user interaction throughout the entire routing process. The user can stop the program at any time, edit the wire patterns and tell the system to continue.

As demonstrated by WEAVER, an expert system is a feasible routing approach. But, the complexities of the experts and the huge demand on computing resources make expert routers a rather expensive alternative.

2.3.3 Simulated Annealing

In many practical instances of VLSI routing, the problems can be viewed as large scale optimization problems involving the routing of many nets. The method of simulated annealing recently introduced by Kirkpatrick, Gelatt and Vecchi [KIR83] is especially suited in solving such problems. This method is intended for problems with very many degrees of freedom and objective

functions that combine conflicting goals. The problems of finding the optimal solution to such problems may be NP-complete; but in practice, one often needs only a good solution and an assurance that there are no solutions significantly better than the one found.

The simulated annealing technique makes use of the following analogies between a multivariate optimization problem and a hypothetical fluid consisting of many interacting atoms:

Hypothetical Fluid	Optimization Problem
internal energy	objective function
atomic positions	parameter values
cooling into a stable, low energy state	finding a near optimal configuration

To bring the fluid into a low energy state (for example, in growing large single crystals), the most effective procedure is careful annealing. First, one melts the fluid, then lowers the temperature slowly, spending more time at temperature near the freezing point to allow defects to anneal out of the growing crystal, then cools the crystals more rapidly to bring the atoms to rest. The same sequence can be followed in optimization by introducing a pseudo-temperature as a control parameter.

In each step of the optimization process, a new feasible solution is generated from the previous solution. The new solution is accepted with probability 1 if $\Delta E<0$, and with probability $\exp(-\Delta E/T)$ if $\Delta E>0$, where ΔE is the change in energy and T is the pseudo-temperature. This probabilistic measure of acceptance has the feature of allowing uphill moves in the process of searching for a solution. In algorithms that do not allow uphill

moves, the search process may be trapped in a local minimum with no chance of progressing towards a lower minimum. The annealing optimization process produces a metastable state of the fluid, which is not necessarily the true ground state or the global optimum. But, as T decreases, it gets closer to the optimum. In this section, the basic simulated annealing algorithm of the TimberWolf Placement and Routing Package [SEC84] will be briefly described.

TimberWolf is an integrated set of placement and routing programs developed at the University of California, Berkeley. It consists of a standard cell placement program, a standard cell loose routing program, a generalized gate array placement program, and a macro/custom cell placement program. All these programs use the same basic simulated annealing algorithm to arrive at a final solution, or improve upon an initial solution. The algorithm can be stated in pseudo-codes as follows:

[TimberWolf Simulated Annealing Algorithm]:

```
SimulatedAnnealing( state, temp );
    initial_state:   given initial state;
    initial_temp:    given initial temperature;

    (variable definitions)
    prev_state:      previous state;
    new_state:       new generated state;
    prev_temp:       previous temperature;
    new_temp:        new generated temperature;

    (subroutine definitions)
    cost();          (given a state this function returns the cost value)
    accept();        (given the new state cost and the old state cost
                     decides whether to accept or reject the new state)
```

```
begin
old_state := initial_state;
prev_temp := initial_temp;
while stopping criteria not satisfied
   begin
   generate new_temp < old_temp;
   old_temp = new_temp;
   while inner loop criterion not satisfied
      begin
      generate new_state;
      evaluate the new_state cost c(new_state);
      if accept( c(new_state),c(old_state),old_temp ) then
         new_state := old_state;
      end;
   end;
end.

subroutine accept( new_state_cost, old_state_cost, T )
   new_state_cost:   cost value of the new state;
   old_state_cost:   cost value of the old state;
   T:                current temperature;

   (variable definitions)
   Δc:               change in cost;

   (subroutine definitions)
   random()          (returns a random number with uniform distribution)

begin
cost_change := new_state_cost - old_state_cost;
if Δc ≤ 0 then
   return true;
else
   return random(0,1) < exp(- Δc / T);
end.                                                          □
```

In the above algorithm, the most important part is the function accept. Note that if the new state has a $\Delta c \leq 0$, the new state is always accepted. However, if the new state has a $\Delta c > 0$, then the parameter T plays a fundamental role. If T is large, the random number generated is very likely to be less than $\exp(-\Delta c/T)$ and the new state is almost always get accepted regardless of Δc. If T is small (close to 0), then only new states with Δc slightly greater than 0 have any chance of getting accepted. Thus, in general, all states with $\Delta c > 0$ have smaller chances of getting accepted for smaller values of T.

In addition, the "stopping criterion" used by the TimberWolf package is based on the cost of the new state at the end of each annealing stage. If the cost does not change for four consecutive stages, the "stopping criterion" is met and the process terminates. The "inner loop criterion", on the other hand, specifies the number of new states generated for each annealing temperature. Depending on the problem, this criterion is different. For example, for the gate array placement and standard cell loose routing programs, 20 new states are generated for each temperature. This is necessary because the TimberWolf system does not continuously adjust its annealing temperature, it generates a new temperature by multiplying the previous temperature with a parameter α. For each temperature T, an α is specified to control the temperature change or the annealing schedule. By generating, for example, 20 new states for each temperature effectively approximates a continuous annealing schedule function by a staircase function.

As can be seen, the TimberWolf package is in fact fairly rudimentary. Recent research on convergence and acceleration issues have provided much more understanding on the simulated annealing technique as an adaptive

heuristic technique where parameters may be modified. Moreover, specialized simulated annealing algorithms that tailored for specific classes of problems have great promise in solving NP-complete problems while reducing the massive computing resources usually required by general-purpose adaptive heuristics [KIN87].

2.3.4 Summary

Hardware routers, expert routers and the simulating annealing technique have been described in this section. Hardware routers are fast but the problem size is usually limited by the system size. Expert routers are capable of producing very good solutions while the simulated annealing technique is capable of producing even the optimal solution. But both of these approaches require massive computing resources.

This concludes the survey on VLSI routing techniques. Through the development of routing techniques from early maze-running and line-search algorithms that sequentially route one net at a time, to channel routing algorithms that maintain a high level of global cohesiveness by dividing the problem into loose and detailed routing, to more recent research in hardware routers, expert routers and the simulated annealing technique, the complexities, considerations and tradeoffs in VLSI routing should be clear. Applying the knowledge gained through this extensive survey, a channel routing algorithm is developed. In the next two chapters, the algorithm will be introduced.

CHAPTER III

A NON-DOGLEG CHANNEL ROUTER

A large number of detailed routing algorithms have been developed since the channel routing concept was introduced by Hashimoto and Stevens in 1971 [HAS71]. In order to cope with the complexity of the channel routing problem, which is NP-complete, practical algorithms must employ heuristics. The heuristics are embedded in a mathematical model of the routing process. Such models include graphs [YOS82] and probabilistic hill climbing [ROM84]. The graph-based model has been selected because of its relative simplicity and fairly accurate representation of the routing process. In this chapter, the development and implementation of a graph based heuristic non-dogleg channel routing algorithm will be described.

3.1 Definitions

Before the algorithm can be described, the pertinent concepts and definitions must be introduced. In this section, the following will be described: (i) the definition of the channel routing problem considered, (ii) the definition of doglegs, (iii) a net list representation of the channel routing problem, (iv) the definition and construction of a vertical constraint graph, (v) the definition and construction of a horizontal constraint graph, and (vi) the definition of channel density, channel ordering and channel height lower bounds.

3.1.1 Non-Dogleg Channel Routing Problem

The channel routing problem considered is basically the regular channel routing problem described in Section 2.2.2. The definition is repeated here with the minor modifications for the sake of completeness.

1. A channel is an open-ended rectangular routing region. If a horizontal orientation is assumed, two rows of fixed terminals are located at the top and bottom boundaries, while the left and right boundaries are open and interconnections may exit through floating terminals. The location of the floating terminals are not specified but decided by the router.
2. The channel has no initial interior obstructions such as pre-routed wires or interior routing voids.
3. Routing is performed on a virtual rectilinear grid consisting of vertical and horizontal grid lines called vertical and horizontal tracks, respectively. All wire traces are routed inside the channel and on the tracks, that is, no routing outside the channel and no diagonal wire traces. The track spacing are such that the proper clearance between features is ensured.
4. Two routing layers are available with vertical wire traces exclusively on one layer and horizontal wire traces exclusively on the other layer. Wire traces located on different layers are connected by vias.
5. Net terminals are located on the intersections of the top and bottom channel boundaries and the vertical tracks, and are accessible on at least the vertical routing layer.
6. Each net is restricted to have at most one horizontal trace; that is, no doglegs are allowed. This restriction will be relaxed to allow doglegging at terminal positions as an extension to the algorithm.
7. The channel height is defined as the number of horizontal tracks

between the top and bottom boundaries. The channel height is assumed to be indefinitely extendable. That is, the channel area can be made as large as necessary.

8. The objective is to route all the nets in minimum channel height, and hence minimum area.

In the above definition, the channel height is allowed to extend indefinitely, thus guaranteeing 100% routing completion. However, in practice, the allowable channel height (channel capacity) is determined by the placement of modules. The question is whether the routing of nets could be completed.

In such cases, it is up to the loose router to ensure that a channel must not be assigned more nets than its capacity, and that the channel router would be able to complete the final routing. the efficiency of the channel router, therefore, has a very important impact on the entire layout process.

Note that if it is known to the loose router that the channel router can complete the routing in the minimum theoretical channel height, the loose router can assign the maximum number of nets to each channel so that the placer can further compact the distribution of modules. If, however, the channel router cannot complete the routing in the minimum theoretical channel height, the loose routing process must be repeated to re-assign the channels. If the channel router still cannot complete the routing with the new channel assignment, the placement of modules must be modified to increase the channel capacities. These three processes, placement, loose routing, and channel routing, must be iterated until a complete layout is produced. However, such an iterative process may be impractical.

3.1.2 Dogleg

In the definition of the channel routing problem, the number of horizontal traces per net is limited to one (point 6). However, there are situations where dividing the horizontal traces into more than one horizontal segments may allow the channel to be routed in lower channel height. The dividing of a horizontal trace into two or more horizontal segments on different horizontal tracks, or the bending of an otherwise straight wire trace is called doglegging. A more detailed discussion on the advantages and tradeoffs of doglegging will be presented in the next chapter on the dogleg extension of the basic algorithm.

3.1.3 Net List Representation of a Channel Routing Problem

The channel routing problem can be represented graphically as shown in Fig. 14. For computational purposes, however, the problem would be more conveniently represented as a matrix or a net list. In a matrix representation, a matrix $A=[a_{ij}]$ of dimension $N \times M$ is used to represent a channel routing problem consisting of N nets and M terminals. For each matrix element a_{ij},

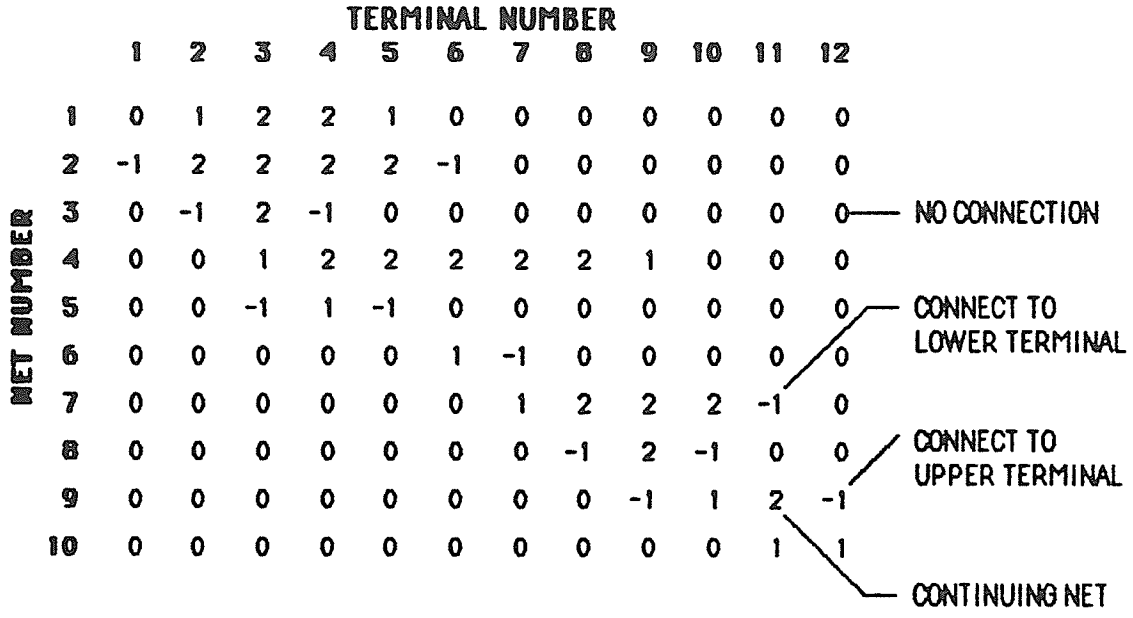
$a_{ij} = +1$ if net i is connected to terminal j on the upper boundary;

$a_{ij} = -1$ if net i is connected to terminal j on the lower boundary;

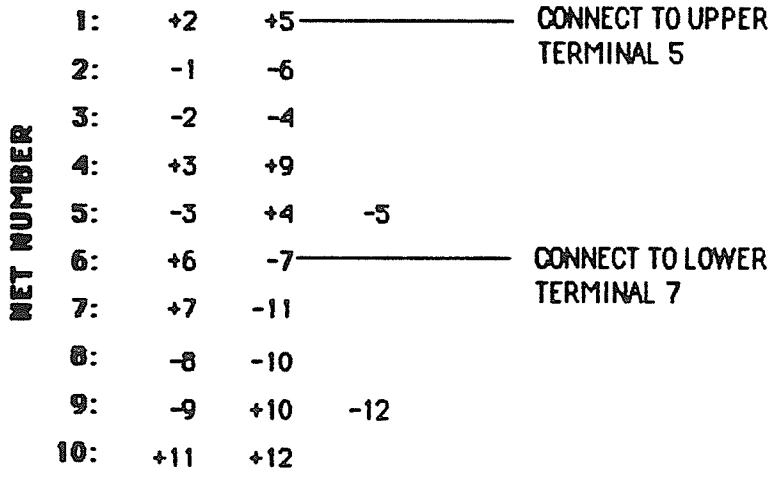
$a_{ij} = 2$ if net i crosses vertical track j but is not connected to terminal j on either side of the channel;

$a_{ij} = 0$ otherwise.

A matrix representation of the channel in Fig. 14 is shown in Fig. 18a. Note that although the problem does not call for any net numbering, in a matrix



(a)



(b)

Fig. 18. Channel representations. (a) Matrix representation; (b) Net list representation.

representation, a net is numbered according to its row number in the matrix. However, the placement of the nets in the matrix is not unique and does not necessarily represent any particular order of processing.

In addition to the above matrix representation, another possible representation of the channel routing problem is a net list representation. For each net in the channel, a set of terminal connections (t_i) is specified, such that

$t_i = +j$	if the connection is to the j^{th} terminal on the upper boundary;
$t_i = -j$	if the connection is to the j^{th} terminal on the lower boundary;
$t_i = 0$	if the net has a floating terminal on the left end of the channel;
$t_i >$ channel length	if the net has a floating terminal on the right end of the channel.

A net list representation of the channel in Fig. 14 is shown in Fig. 18b. Here the nets are numbered in the order they were entered.

Theoretically, both the matrix representation and the net list representation are equivalent and complete. In practice, however, a net list representation has several advantages over the matrix representation.

1. A computer representation of the matrix would require N×M storage, while the net list representation requires only T, where T is the total number of terminal connections, and $T \ll N \times M$.
2. Using a ±1 to represent a terminal connection and a 2 to represent a net continuation complicate the length and distance calculations. For

example, to calculate the length of a net in a matrix representation would require a scan of the corresponding row in the matrix to find the left and right endpoints of the net. A net list representation, on the other hand, lends itself naturally to more efficient linked list data structures. Using a pointer to the last terminal connection the length and distance calculations become simple additions and subtractions.

3.1.4 Vertical Constraint Graph

Since wire traces are restricted to their respective routing layers only, if two vertical wire traces belonging to two different nets must be placed in the same vertical track, the lower endpoint of the upper vertical wire trace must be placed in a horizontal track above the upper endpoint of the lower vertical wire trace. This requirement is called a vertical constraint. In the basic non-dogleg algorithm, each net is limited to at most one horizontal wire trace. A vertical constraint thus requires that the horizontal wire trace of the upper net be routed above the horizontal wire trace of the lower net. This is not necessarily true if doglegs are allowed, since some nets may have more than one horizontal wire trace.

The vertical constraints can be represented by a directed graph $G_v(E,V)$, where a vertex $v \in V$ corresponds to a net, and a directed edge $e \in E$ emanating from vertex a to vertex b indicates a vertical constraint and that net a must be placed above net b. Vertex a is said to be an ancestor of vertex b (vertex b is a descendent of vertex a), if there is a directed edge from a to b in $G_v(E,V)$. Also, each vertex is assigned an ordering number according to the following recursive equation:

$$\text{ord}(v) = \begin{cases} 1 & \text{if } U = \phi \\ \max_{u \in U} \text{ord}(u) + 1 & \text{if } U \neq \phi \end{cases}$$

where $U \in V$ is the set of descendent vertices of vertex v. Note that a cyclic constraint occurred if there is a directed cycle in $G_v(E,V)$. In this case, ordering numbers cannot be defined on the vertical constraint graph, and the channel is unroutable, since each net in the cycle requires another net to be routed above it. Doglegs would be required to break such cyclic constraints. On the other hand, if the vertical constraint graph is acyclic, ordering numbers can always be defined and the routing specification is always realizable even without dogleg. The vertical constraint graph for the channel in Fig. 14 is shown in Fig. 19a.

3.1.5 Horizontal Constraint Graph

If two horizontal wire traces belonging to two different nets lay on the same horizontal track and have one or more vertical tracks in common, they cannot overlap and must be assigned to different horizontal tracks. This requirement is called a horizontal constraint. The distance between two nets i and j, denoted as d_{ij}, is defined as the minimum number of horizontal grid spaces the nets must be moved to overlap. For example, in Fig. 14, $d_{16}=d_{61}=1$, $d_{12}=d_{21}=0$, $d_{19}=d_{91}=4$, and $d_{67}=d_{76}=0$.

The horizontal constraints can be represented as a weighted undirected graph $G_h(V,E)$, where a vertex $v \in V$ corresponds to a net, and an undirected edge e between vertices a and b exists if and only if the nets a and b do not overlap. An edge weight d_{ab} is assigned to the edge e representing the

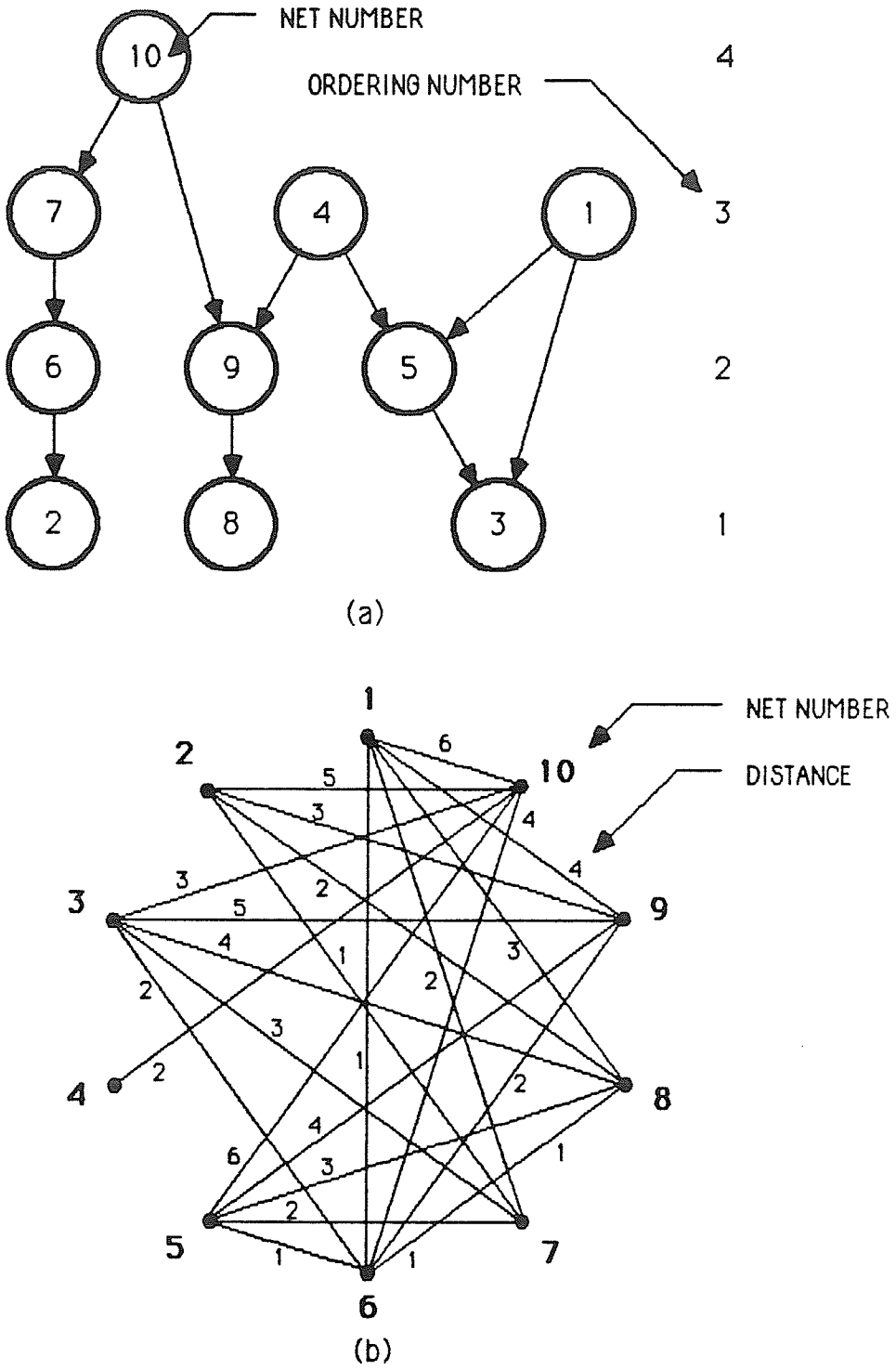


Fig. 19. Constraint graphs for the channel shown in Fig. 14.
(a) Vertical constraint graph; (b) Horizontal constraint graph.

distance between the vertices a and b. The horizontal constraint graph for the channel in Fig. 14 is shown in Fig. 19b.

3.1.6 Density, Ordering, and Channel Height Lower Bounds

The number of horizontal traces that crosses a vertical track is the local density of that vertical track. The maximum local density in the channel is the density of the channel. For example, for the channel in Fig. 14, the local density of vertical track 1 is 2, vertical track 2 is 3, vertical track 3 is 5, and so on, and the channel density is 5 (occurred at vertical tracks 3 and 4).

Moreover, the highest ordering number defined in the vertical constraint graph of a channel is the ordering of the channel. From the vertical constraint graph shown in Fig. 19a, the ordering of the channel in Fig. 14 is 4.

For the channel routing problem defined in Section 3.1.1, the lower bound on the channel height is determined by at least two factors: the channel density and the channel ordering. Without providing any detailed discussion on this topic here, it can be observed that the channel density is an obvious lower bound on the channel height, since every horizontal trace crossing the vertical track where the maximum local density occurs requires one horizontal track. In fact, the channel density is the least lower bound on the channel height. As long as horizontal traces are restricted to one routing layer, no channel can be routed in fewer horizontal tracks than its density.

If each net in a channel is limited to have at most one horizontal trace, that is, if doglegging is not allowed, another lower bound on the channel height is the channel ordering, which is the length of the longest constraint

chain in the vertical constraint graph. Since the horizontal trace of each net in the constraint chain must be routed in a different horizontal track, the channel height must be at least equals to the channel ordering.

Though effort has been devoted to finding the greatest lower bound in channel height (necessary and sufficient channel height) [GAM81, RIC84], no such lower bounds have been find for general channel routing problems. Without any further discussion on the subject of the necessary and sufficient channel height, the channel density and the channel ordering are noted here as two of the lower bounds.

3.2 A Graph Based Heuristic Channel Router

The algorithm routes the channel one horizontal track at a time starting from the top of the channel. For each horizontal track, it examines the nets that can be routed on that track using the vertical constraint graph. Those nets with no vertical constraints that require other nets to be routed first are selected and assigned Mother net priorities. The one with the highest priority is selected as the Mother net and routed in the track. Then among the unrouted nets, a Ready net set is formed consisting of nets that (i) do not require other nets to be routed first (not vertically constrained), and (ii) are not in horizontal conflict with the Mother net. Each of the nets in the Ready net set is assigned a net priority with respect to the Mother net. At this point, any one of those Ready nets can be routed with the Mother net in the same track without any vertical or horizontal constraint violations. To maximize the utilization of the track, however, a subset of the Ready net set is found such that, (i) there are no horizontal conflicts between all the nets

in the subset, (ii) the combined priority of all the nets in the subset is the maximum. Since the finding of such a subset is NP-complete, a heuristic algorithm is used (discussed later). Once the subset is determined, the nets in the subset are routed with the Mother net. The channel height is then increased by one horizontal track and the above process is repeated until all the nets are routed. A more precise definition of the algorithm is given in the following pseudo-code.

[Non-dogleg Channel Routing Algorithm]:

NonDoglegChannelRouter;

{Variable Definitions}

S_0:	set of nets to be routed;
S_1:	set of nets assigned to the current horizontal track;
S_m:	set of candidates for the mother net;
G_v:	vertical constraint graph;
TrackNumber:	current horizontal track number;
ReadyNetSet:	set of nets that can be routed with the MotherNet;

Begin

1. {initialization}
   S_0 := unrouted nets;
   S_1 := ϕ;
   TrackNumber := 0;

   {main loop}
   repeat
2. {process next horizontal track}
   TrackNumber := TrackNumber + 1;

3. {Mother net selection}
   S_m := nets in S_0 with no ancestors in G_v;
   MotherNet := net in S_m with maximum Mother net priority;

4. {Ready net set creation}
ReadyNetSet := all nets in S_0 with no ancestors in G_v and have non-zero distances with MotherNet;

5. {Maximal subset S_1 selection}
choose a subset S_1 of nets in ReadyNetSet such that
(i) all the nets in S_1 have no overlaps
(ii) the combined priority for nets in S_1 is the maximum;

6. {Track assignment}
assign MotherNet and S_1 to TrackNumber;

7. {Graph update}
delete MotherNet and S_1 from S_0;
delete vertices corresponding to MotherNet and nets in S_1 from G_v;

8. {Repeat until all nets are routed}
until $S_0 = \phi$;

end; {NonDoglegChannelRouter} □

3.2.1 Mother Net Selection

The first condition for any net to be considered as a potential Mother net is that its routing must not require other nets to be routed first. In other words, the net must not be vertically constrained by any other nets. This implies that the net must have no ancestors in the vertical constraint graph. A Mother net candidate set S_m is thus constructed consisting of unrouted nets that have no ancestors in the vertical constraint graph. A Mother net priority function is used to give a quantitative measure on the goodness of a candidate as the Mother net. The net with the highest priority is selected as the Mother net.

After experimented with a number of Mother net priority functions, two factors, the length and the ordering number of a net, are identified to give a fair measure of the goodness of a potential Mother net. Functions of various forms have been tested, and the following is the final Mother net priority function selected.

$$f_m(\text{Mother}) = M_{length} * \text{Length(Mother)/MaxMotherLength} + \text{ChannelOrder/ChannelDensity} * M_{ordering} * \text{Order(Mother)/MaxMotherOrder}$$

where f_m is the Mother net priority function, Mother is the Mother net candidate, Length() gives the length of a net, Order() gives the ordering number of a net, MaxMotherLength is the maximum net length among all the Mother net candidates, ChannelOrder is the order of the channel, ChannelDensity is the density of the channel, MaxMotherOrder is the maximum ordering number among all the Mother net candidates, and M_{length} and $M_{ordering}$ are the weighting factors for the length and the ordering terms, respectively.

Basically, the above function is a linear combination of the length and the ordering of the Mother net candidate. Both of those terms are normalized, so that the relative emphasis on the length and the ordering can be controlled by the weighting factors, and the influence by the absolute values of the terms are reduced. Since the net lengths various considerably in different channels, in order to have an algorithm that is capable of routing channels with diverse characteristics, the normalization is necessary.

Moreover, experiments have shown that for channels with a relatively high density, a higher M_{length} usually produce better results, and for channels with a relatively high ordering, a higher $M_{ordering}$ usually produces better

results. In light of this observation, and with the aim of developing an algorithm that can produce good solutions for the general class of channel routing problems described, an adaptive factor, ChannelOrder/ChannelDensity, was included in the Mother net priority function. This allows the algorithm to adapt to the density and ordering of the channel automatically.

3.2.2 Ready Net Set Creation

After a Mother net is selected, a Ready net set is form with respect to the selected Mother net. The basic condition that a net must not have any ancestors in the vertical constraint graph is still required. In addition, among the nets with no ancestors, a Ready net must have a non-zero distance with the Mother net. This latter condition corresponds to having an edge connecting the potential Ready net and the Mother net in the horizontal constraint graph.

Once the Ready net set is formed, each net in the set is given a priority with respect to the selected Mother net. After experimented with a number of priority functions, the following function is chosen as the net priority function, f_n:

$$f_n(\text{Mother,Net}) = N_{ordering} * \text{Order(Net)} / \text{MaxReadyNetOrder} + N_{length} * \text{Length(Net)} / \text{MaxReadyNetLength} + N_{distance} * (\text{MaxReadyNetDistance} - \text{Distance(Mother,Net)}) / \text{MaxReadyNetDistance}$$

where Mother is the Mother net, Net is the Ready net, MaxReadyNetOrder is the maximum net ordering among the Ready nets, MaxReadyNetLength is the

maximum net length among the Ready nets, and MaxReadyNetDistance is the maximum distance between the Mother net and the Ready nets, Order() gives the ordering number of a net, Length() gives the length of a net, Distance() gives the horizontal distance between two nets, $N_{ordering}$, N_{length} and $N_{distance}$ are the weighting factors for the ordering, length and distance terms, respectively.

Similar to the Mother net priority function, terms in the net priority function are also normalized to reduce the effect of diverse channel characteristics. In addition to the length and ordering terms, another term is included here, namely the distance between the Ready net and the Mother net. It was found that the closer the Ready net to the mother net, the better the track utilization.

3.2.3 Maximal Subset Selection

With the above Ready net set, any one net in the set can be routed with the Mother net without any vertical or horizontal conflicts. But since there may be horizontal conflicts between the Ready nets themselves, not all of them can be assigned to the same horizontal track. The problem is now to find a subset of nets from the Ready net set so that (i) the subset of nets can all be routed with the Mother net in the same horizontal without conflicts, (ii) the resulting assignment will result in an optimal or near optimal solution.

Using the described net priority function f_n the best subset can be considered as the one that has the maximum combined net priority among all subsets that satisfy the subset requirements. This maximal subset selection problem is equivalent to finding the maximal subset in the weighted

horizontal constraint graph. Such a maximal subset problem is known to be NP-complete. Thus, a heuristic technique is used here.

First, the Ready net set is ordered in descending order of net priorities. Then, a small number of test subsets are created (for example, five), and each test subset is initialized to contain one of the highest priority nets. Then each test subset is expanded by adding as many nets as possible from the Ready net set in descending ordering of priorities provided that the addition would not create any horizontal conflicts among the nets already in the test subset. When all the nets are examined and none of the test subsets can be further expanded, the nets in the test subset with the highest combined net priority are selected to be routed with the Mother net.

3.2.4 Track Assignment and Graph Update

After the maximal subset is found, all the nets in that subset are routed in the same horizontal track with the Mother net. The vertical constraint graph is then updated by deleting those vertices corresponding to the routed nets. If the set of remaining unrouted nets is not empty, the channel height is increased by one horizontal track and the routing process is repeated until all the nets are routed.

3.3 Implementation

The above algorithm has been implemented in Domain C Revision 3.12 on an Apollo Domain DN660 workstation under the Aegis operating system Version 8.0. Structured charts and program listing of the channel router with the

dogleg extension are included in Appendices A and B. As shown in the structured chart of the main program, DOGLEG.C, in Appendix A, Fig. A1, the program is consisted of three main sections: (i) Filer, (ii) Doglegger, and (iii) Non-Dogleg Router. The first section, Filer, performs basic input error checking and creates a linked list data structure for the input netlist. The second section, Doglegger, modifies the input netlist so that dogleg routing can be achieved using the described non-dogleg routing algorithm. Finally, the third section, Non-Dogleg Router, performs the described non-dogleg routing. For the non-dogleg algorithm, only the Filer and the Non-Dogleg Router sections are used. Since the Doglegger section is independent of the other two sections and pertinent only to the dogleg algorithm, its description is postponed until the next chapter on the dogleg extension of the algorithm. Here, the Filer and the Non-Dogleg Router sections will be described.

When the program is invoked, the Filer section reads in the specified weighting factors and netlist files. The weighting factors are stored in a set of global variables accessible by the priority functions. A list representation of the netlist as described in Section 3.1.3 is then created. The head nodes of the nets are arranged in an array for more efficient reference by net numbers. Each head node contains: (i) two pointers, FIRST_TERM and LAST_TERM, pointing to the first and last nodes of a linked list of terminal connections, and (ii) two integers, LEFT and RIGHT, containing the vertical track numbers of the left and right endpoints of the net.

After the netlist data structure is created, the Non-Dogleg Router takes over. It first creates the vertical and horizontal constraint graphs for the netlist. The vertical constraint graph is represented as an adjacency list embedded in the netlist structure. Each head node in the netlist structure

contains: (i) a pointer, FIRST_SON, pointing to a linked list of descendent nodes of the net in the vertical constraint graph, (ii) an integer, ORD_NUM, containing the ordering number of the net, and (iii) an integer, PARENT, containing the number of ancestor nodes the net has. After the structure of the vertical constraint graph is created, a recursive procedure is called to calculate the ordering numbers of the nodes. Then the horizontal constraint graph is created. The horizontal constraint graph is represented as an adjacency matrix for more efficient access to distances and horizontal conflicts between nets.

With the graphs created, the Non-Dogleg Router proceeds to perform non-dogleg track assignments. As described in the pseudo-codes of the algorithm, the track assignment process consists of four major steps: (i) Mother net selection, (ii) Ready net set creation, (iii) maximal subset selection, and (vi) graph update. All of these steps have been implemented as they were described in Sections 3.2.1 to 3.2.4.

The program listing of the channel router with the dogleg extension, Doglegger, is included in Appendix B. By removing the procedure call to the subroutine Doglegger, the program would perform non-dogleg channel routing on the input netlist. Such a modular approach would allow the user to conveniently select between dogleg and non-dogleg routing using a simple command line switch.

The program listing in Appendix B was transferred directly from the Apollo workstation to a Macintosh microcomputer, where the program was printed with page numbering. Correctness of the program has been verified by comparing the partial results of several examples generated by the program

and by manual traces of the algorithm.

3.4 Efficiency of the Non-Dogleg Routing Algorithm

In this section, efficiency of the non-dogleg algorithm will be illustrated through detailed traces of two examples and through an analysis of the CPU times required versus the complexity of the channels. First, partial results generated at each major step of the algorithm are shown for two examples.

3.4.1 Example 1

Consider the channel routing problem shown in Fig. 14. The channel is 12 terminals long, and consists of 10 nets. The vertical constraint graph and the horizontal constraint graph of the channel are shown in Fig. 19. From the graphs, the channel density is 5 and the channel ordering is 4. Thus, the minimum channel height is 5.

A step by step illustration of the algorithm is presented here. At each stage of the routing process, reduced vertical constraint graphs are used to show the progress made in each step (Fig. 20). The weighting factors used in the Mother net priority function f_m are M_{length}=10 and $M_{ordering}$=40. The weighting factors used in the net priority function f_n are $N_{ordering}$=15, N_{length}=10 and $N_{distance}$=5.

Step 1: $S_0 = \{1,2,3,4,5,6,7,8,9,10\}$

$S_1 = \phi$

TrackNumber = 0

Step 2: TrackNumber = 1

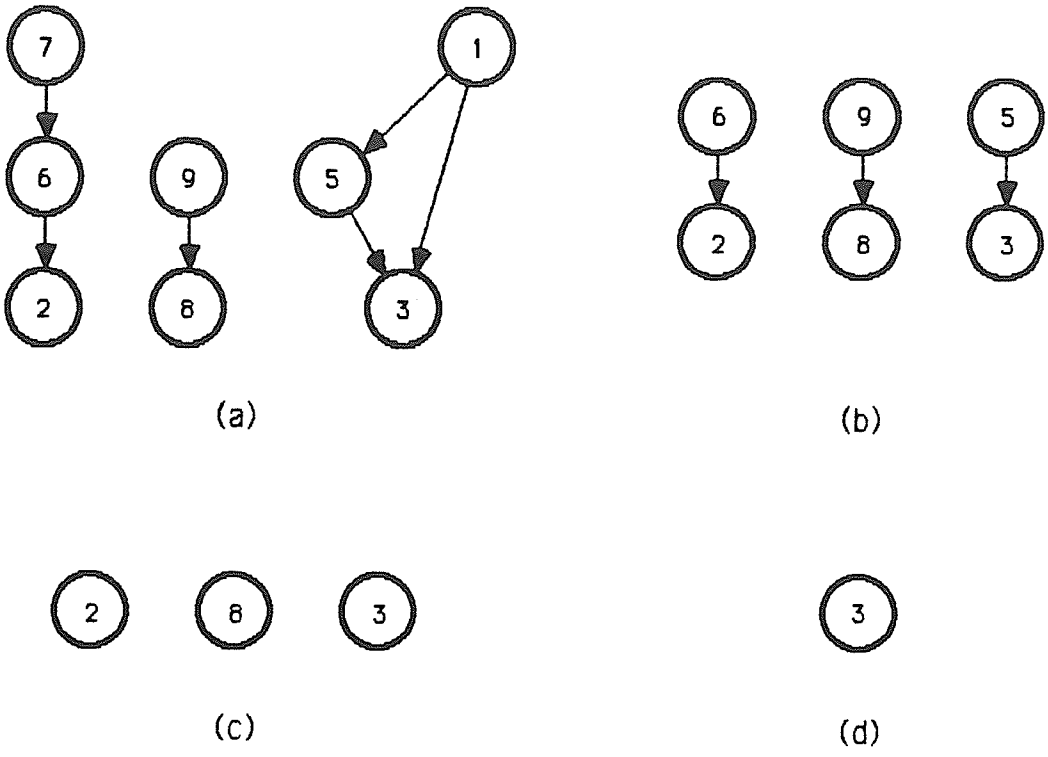


Fig. 20. Reduced vertical constraint graphs for Example 1.

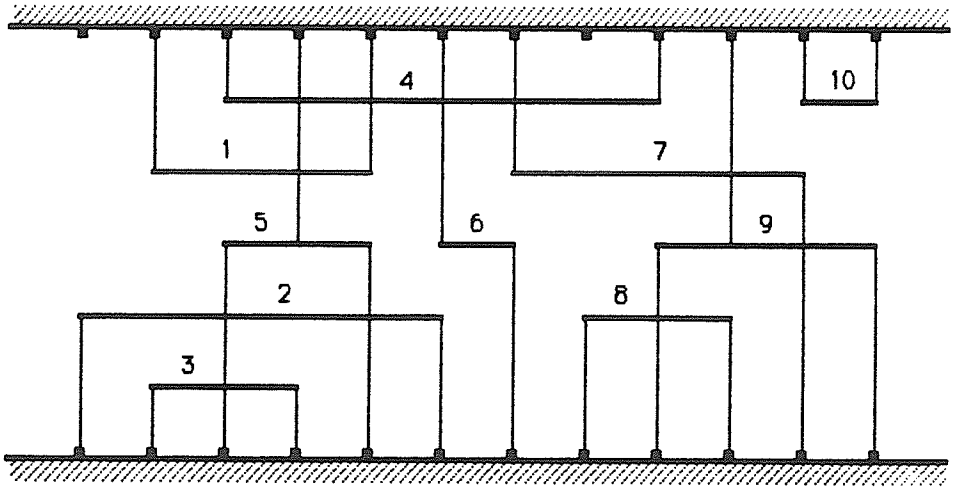


Fig. 21. Realization of Example 1.

Step 3: S_m = {1,4,10}; $f_m(1) = 29$, $f_m(4) = 34$, $f_m(10) = 33$

MotherNet = 4

Step 4: ReadyNetSet = {10}; $f_n(10) = 25$

Step 5: S_1 = {10}

Step 6: Assign net 4 and 10 to track 1

Step 7: Delete net 4 and 10 from S_0; S_0 = {1,2,3,5,6,7,8,9}

Delete vertices 4 and 10 from G_v (Fig. 20a)

Step 8: $S_0 \neq \phi$ repeat from step 2

Step 2: TrackNumber = 2

Step 3: S_m = {1,7,9}; $f_m(1) = 39$, $f_m(7) = 42$, $f_m(9) = 28$

MotherNet = 7

Step 4: ReadyNetSet = {1}; $f_n(1) = 25$

Step 5: S_1 = {1}

Step 6: Assign net 1 and 7 to track 2

Step 7: Delete net 1 and 7 from S_0; S_0 = {2,3,5,6,8,9}

Delete vertices 1 and 7 from G_v (Fig. 20b)

Step 8: $S_0 \neq \phi$ repeat from step 2

Step 2: TrackNumber = 3

Step 3: S_m = {5,6,9}; $f_m(5) = 38$, $f_m(6) = 35$, $f_m(9) = 42$

MotherNet = 9

Step 4: ReadyNetSet = {5,6}; $f_n(5) = 25$, $f_n(6)=22$

Step 5: S_1 = {5,6}

Step 6: Assign net 5, 6 and 9 to track 3

Step 7: Delete net 5, 6 and 9 from S_0; S_0 = {2,3,8}

Delete vertices 5, 6 and 9 from G_v (Fig. 20c)

Step 8: $S_0 \neq \phi$ repeat from step 2

Step 2: TrackNumber = 4

Step 3: $S_m = \{2,3,8\}$; $f_m(2) = 42$, $f_m(3) = 36$, $f_m(8) = 36$

MotherNet = 2

Step 4: ReadyNetSet = {8}; $f_n(8) = 25$

Step 5: $S_1 = \{8\}$

Step 6: Assign net 2 and 8 to track 4

Step 7: Delete net 2 and 8 from S_0; $S_0 = \{3\}$

Delete vertices 2 and 8 from G_v (Fig. 20d)

Step 8: $S_0 \neq \phi$ repeat from step 2

Step 2: TrackNumber = 5

Step 3: $S_m = \{3\}$; $f_m(3) = 42$

MotherNet = 3

Step 4: ReadyNetSet = ϕ

Step 5: $S_1 = \phi$

Step 6: Assign net 3 to track 5

Step 7: Delete net 3 from S_0; $S_0 = \phi$

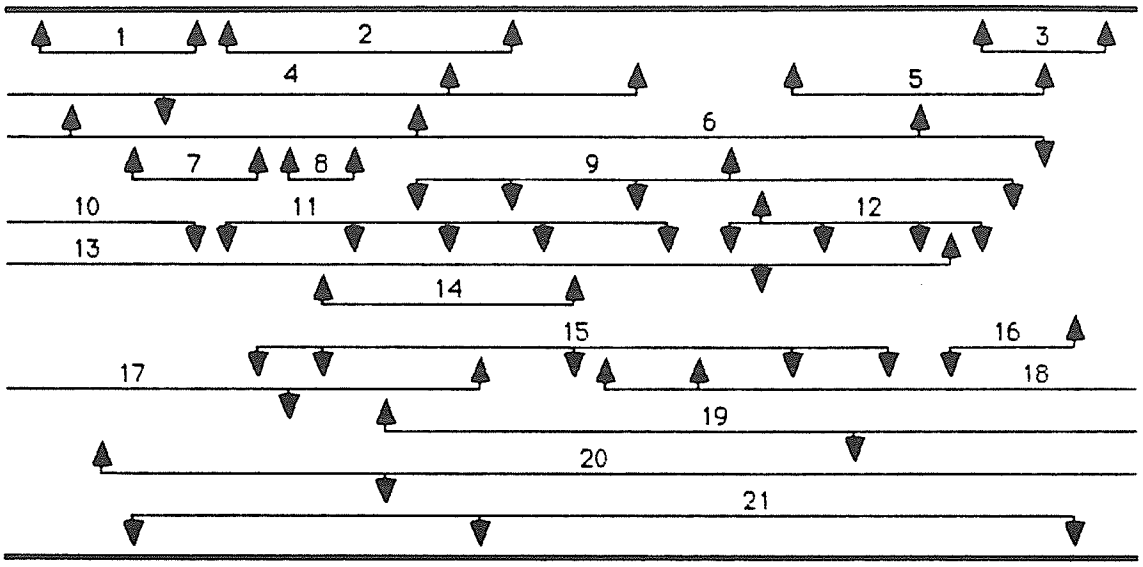
Delete vertex 3 from G_v; $G_v = \phi$

Step 8: $S_0 = \phi$; Stop. Routing completed.

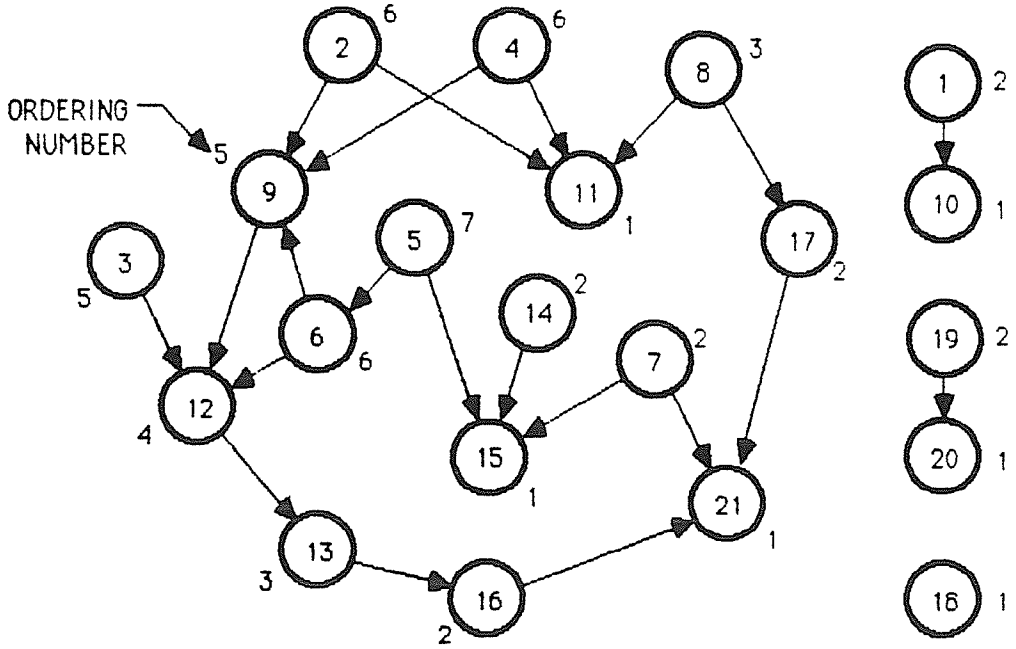
The final track assignment is shown in Fig. 21. A total of 5 horizontal tracks were used in the realization. This channel although small in size, has a fairly high ordering (4) compared to the density (5). However, the algorithm was still able to complete the routing in minimum channel height.

3.4.2 Example 2

A second example is the channel shown in Fig. 22a. The channel is 35 terminals long and consists of 35 nets. As can be seen from the graphical



(a)



(b)

Fig. 22. Representation and vertical constraint graph of Example 2.

(a) Graphical channel representation of example 2;

(b) Vertical constraint graph of example 2.

representation of the channel in Fig. 22a, the channel density is 12. The vertical constraint graph of the channel is shown in Fig. 22b. From the graph, the order of the channel is 7. The lower bound on the channel height is therefore 12. The weighting factors for the Mother net priority function and the net priority function are the same as those used in Example 1.

Table 1 shows the track by track routing of the channel. For each track, the Mother net candidates are listed in the second column. The Mother net priorities are listed in parenthesis after the net numbers. The Mother net selected is shown in bold. The Ready nets with respect to the selected Mother net are listed in the last column and the Ready net priorities are listed in parenthesis. The Ready nets selected are shown in bold. A total of 12 horizontal tracks are used in the realization as shown in Fig. 23. The channel height is again equals to the minimum.

3.4.3 Execution Time Versus Channel Complexity

The current implementation of the algorithm, as a study of the algorithm, was designed for clarity rather than speed. The program is highly modularized utilizing over 55 modules, including 16 diagnostic and messaging subroutines. By optimizing the data structures and coding of the algorithm higher speed is achievable.

Since the algorithm operates on units of nets, the number of nets in the channel gives a fairly good measure of the channel complexity. A plot of the CPU time required on the Apollo workstation versus the number of nets for twelve examples is shown in Fig. 24. The curve labelled NON-DOGLEG shows the CPU time required by the non-dogleg router. The other two curves

Table 1. Routing of Example 2.

Track	Mother Net Candidates	Ready nets
1	1(8) 2(22) 3(17) **4(27)** 5(26) 7(7) 8(9) 14(9) 18(10) 19(16)	3(15) **5(27)**
2	1(8) 2(25) 3(20) **6(33)** 7(8) 8(11) 14(9) 18(8) 19(14)	
3	1(9) **2(26)** 3(20) 7(8) 8(11) 14(10) 18(10) 19(17)	**1(11)** 3(14) **18(16)**
4	3(24) 7(10) 8(13) **9(30)** 10(6) 14(12) 19(19)	**7(13)** **8(15)** 10(13)
5	**3(24)** 10(6) 11(9) 14(12) 17(15) 19(19)	**10(7)** **11(15)** 14(13) 17(17)
6	**12(26)** 14(14) 17(17) 19(21)	14(13) **17(17)**
7	**13(33)** 14(17) 19(23)	
8	14(26) 16(24) **19(33)**	
9	**14(25)** 16(24) 20(21)	**16(25)**
10	15(29) **20(33)** 21(32)	
11	15(29) **21(33)**	
12	**15(33)**	

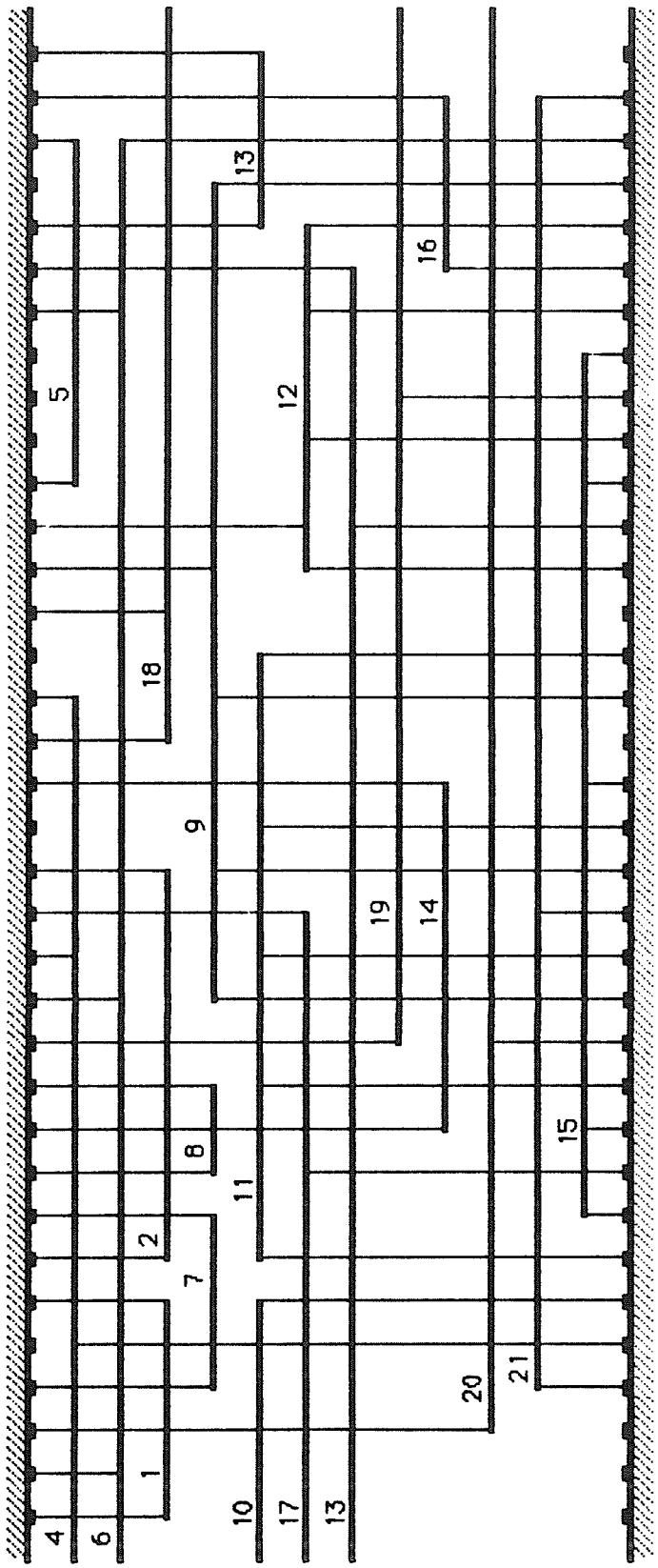


Fig. 23. Realization of Example 2.

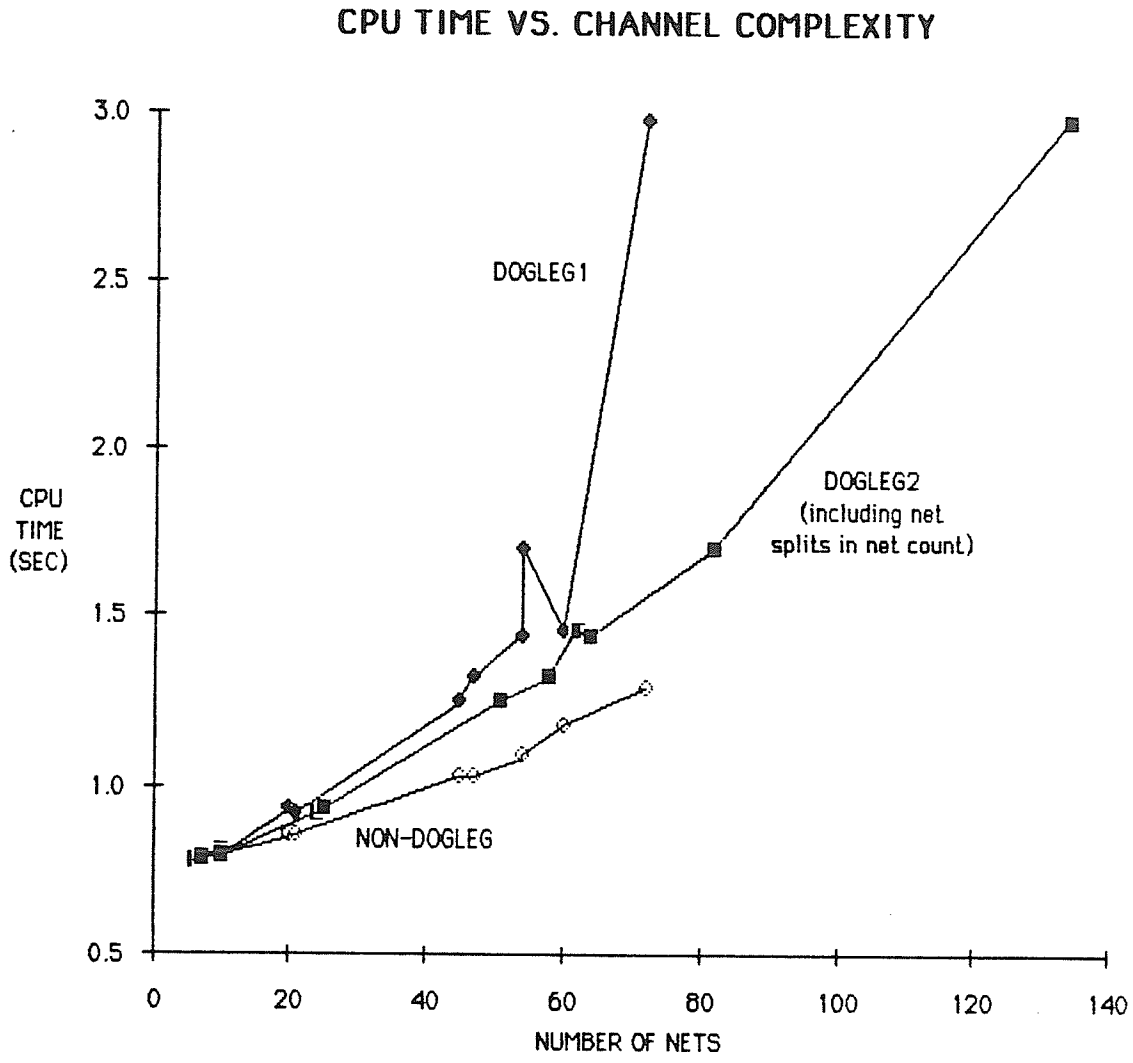


Fig. 24. CPU time vs. channel complexity for the Dogleg and Non-Dogleg channel routers.

labelled DOGLEG1 and DOGLEG2 are CPU times for the dogleg router, which will be discussed in the next chapter on the dogleg extension of the algorithm.

From the curve NON-DOGLEG, it can be seen that the CPU time required by the non-dogleg router is almost linear, increasing only gradually with the number of nets. In particular, the routing of a 72-net channel (Deutsch's Example) required only about 1.3 seconds.

3.5 Experimental Results

A total of twelve examples have been obtained from previously published papers. The collection of examples covered a wide range of channel types from relatively small channels consisting of about 10 nets and 10 terminals to fairly complex channels consisting of 72 nets and 174 terminals. As discussed, the lower bound on the channel height depends on both the channel ordering and the channel density. The selected examples covered channels with ordering numbers lower than density numbers, and channels with ordering numbers higher than density numbers. The twelve examples thus represent a fairly good mix of practical channel routing problems.

Table 2 lists the characteristics of the twelve examples and the resulting channel heights obtained with the non-dogleg routing algorithm. Columns two and three (Nets and Terms) are the number of nets and number of terminals in the channel, respectively. Columns four and five (Density and Order) are the density number and ordering number of the channel, respectively. Column six (Ord/Den) is the ratio of the ordering number to the density number. This ratio is used in the Mother net priority function, f_m, to adapt the algorithm to channels of different characteristics. The last two

Table 2. Characteristics and resulting channel heights of the Examples.

Example	Characteristics					Channel Height	
	Nets	Terms	Density	Order	Ord/Den	Optimal	Result
1	10	12	5	4	0.80	5	5
2	6	7	4	2	0.50	4	4
3	7	17	3	3	1.00	3	3
4	10	18	5	5	1.00	5	5
5	54	79	18	6	0.33	18	19
6	72	174	19	23	1.21	28	28
7	60	119	20	3	0.15	20	20
8	20	35	12	7	0.58	12	12
9	54	119	17	13	0.76	17	18
10	21	35	12	7	0.58	12	12
11	45	63	15	4	0.27	15	15
12	47	63	17	9	0.53	17	18

columns (Optimal and Result) are the optimal channel height and resulting channel height obtained by the algorithm, respectively. Realizations of the twelve examples are shown in Figs. 25-36.

3.6 Discussions

As discussed in Section 3.1.6, the lower bound on the channel height is determined by both the ordering and the density of the channel. In the twelve examples, ordering to density ratios (Ord/Den) from as low as 0.15 to as high as 1.21 are covered (column 6 of Table 2). For channels with high orderings, nets with high ordering numbers should be routed as early as possible, since the maximum ordering in the vertical constraint graph at any one time determines the minimum number of additional tracks required. For channels with high densities, routing the longer nets at the beginning usually results in realizations requiring fewer tracks. Since the goal was to develop a non-dogleg channel routing algorithm for the entire class of problems defined in Section 3.1.1, the algorithm is judged on its performance in the routing of all twelve examples.

As can be seen from Table 2 and Figs. 25-36, the algorithm was able to route all twelve examples in no more than one horizontal track above the optimal channel heights. Furthermore, 75% (nine out of twelve) of the examples were routed in their minimum channel height, and the remaining 25% (three out of twelve) were routed in one track above their minimum channel height. These results were obtained using a single set of weighting factors. By adjusting the weighting factors slightly, all twelve examples were routed in their minimum channel height.

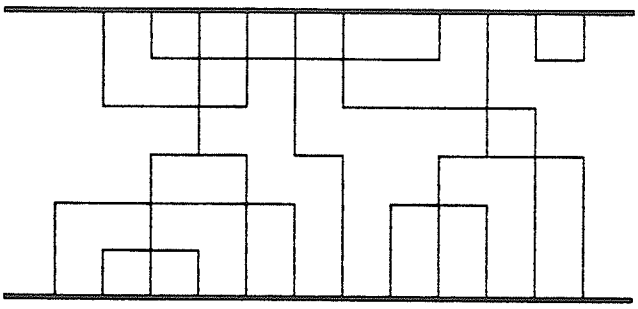


Fig. 25. Non-dogleg realization of Example 1 (channel height: 5).

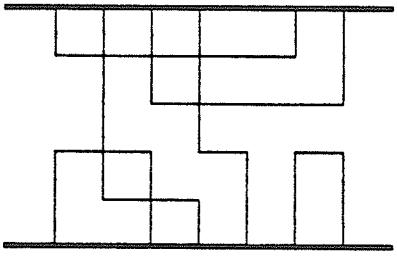


Fig. 26. Non-dogleg realization of Example 2 (channel height: 4).

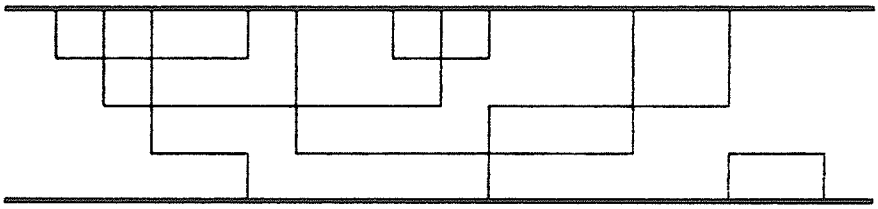


Fig. 27. Non-dogleg realization of Example 3 (channel height: 3).

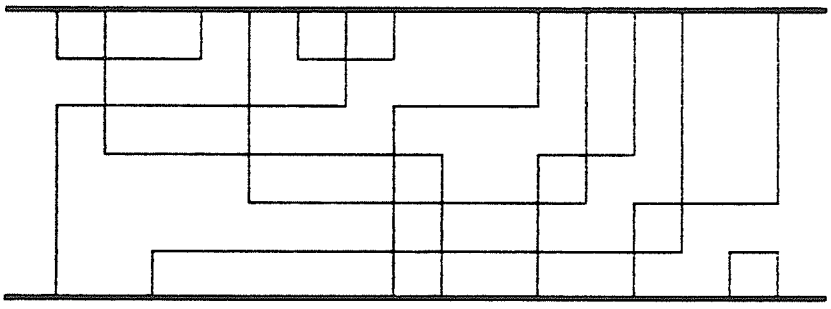


Fig. 28. Non-dogleg realization of Example 4 (channel height: 5).

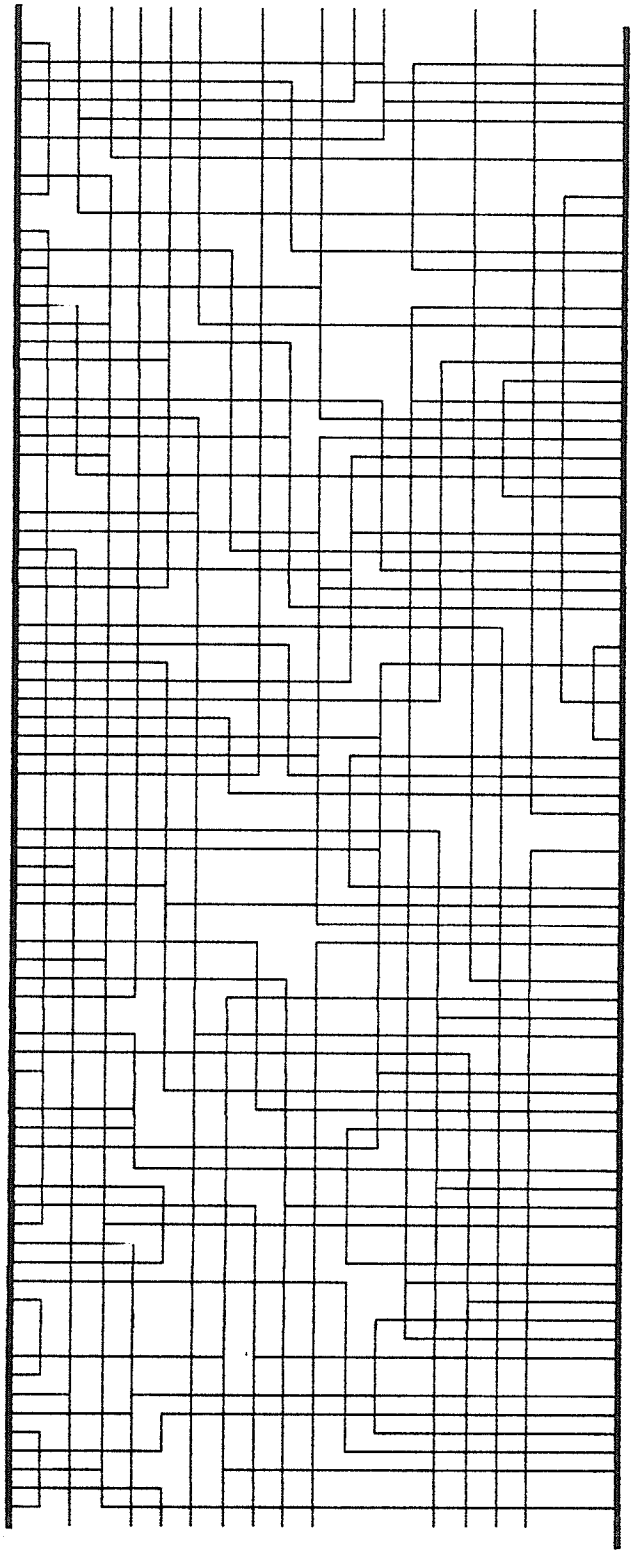


Fig. 29. Non-dogleg realization of Example 5 (channel height: 19).

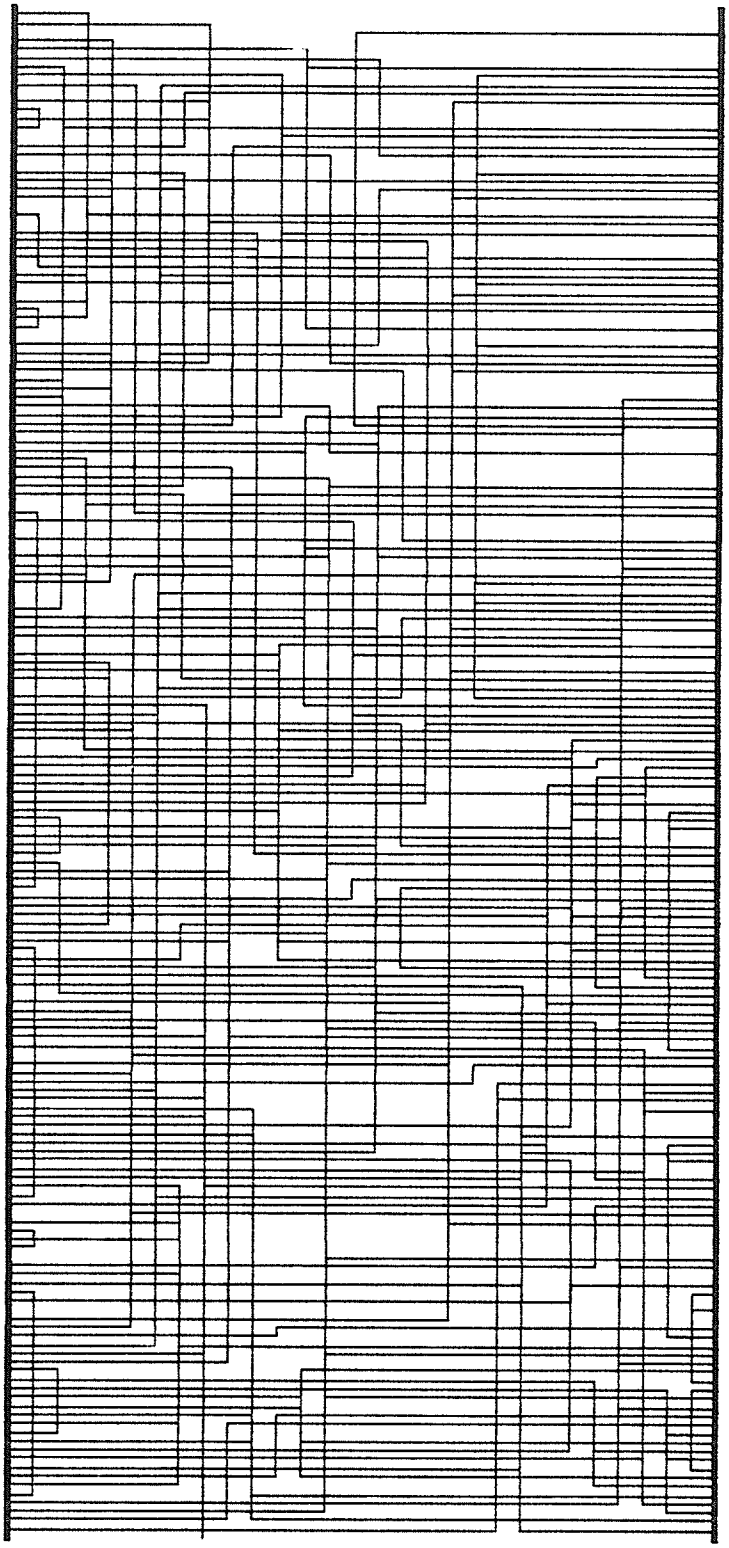


Fig. 30. Non-dogleg realization of Example 6 (channel height: 28).

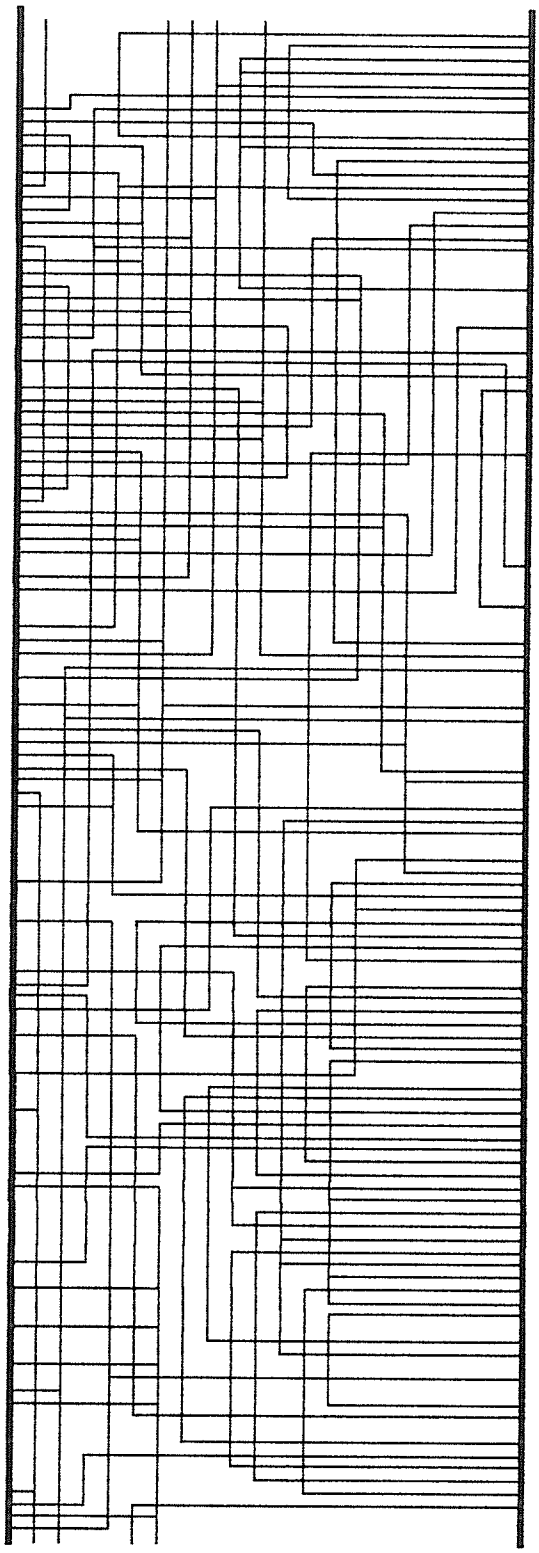


Fig. 31. Non-dogleg realization of Example 7 (channel height: 20).

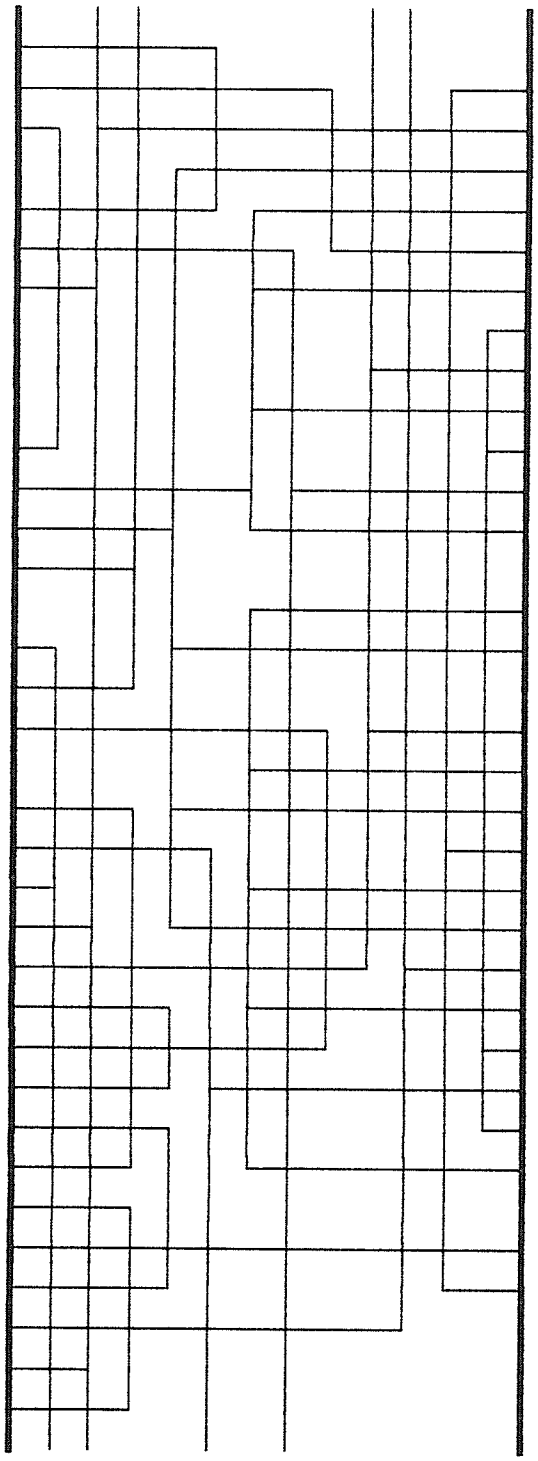


Fig. 32. Non-dogleg realization of Example 8 (channel height: 12).

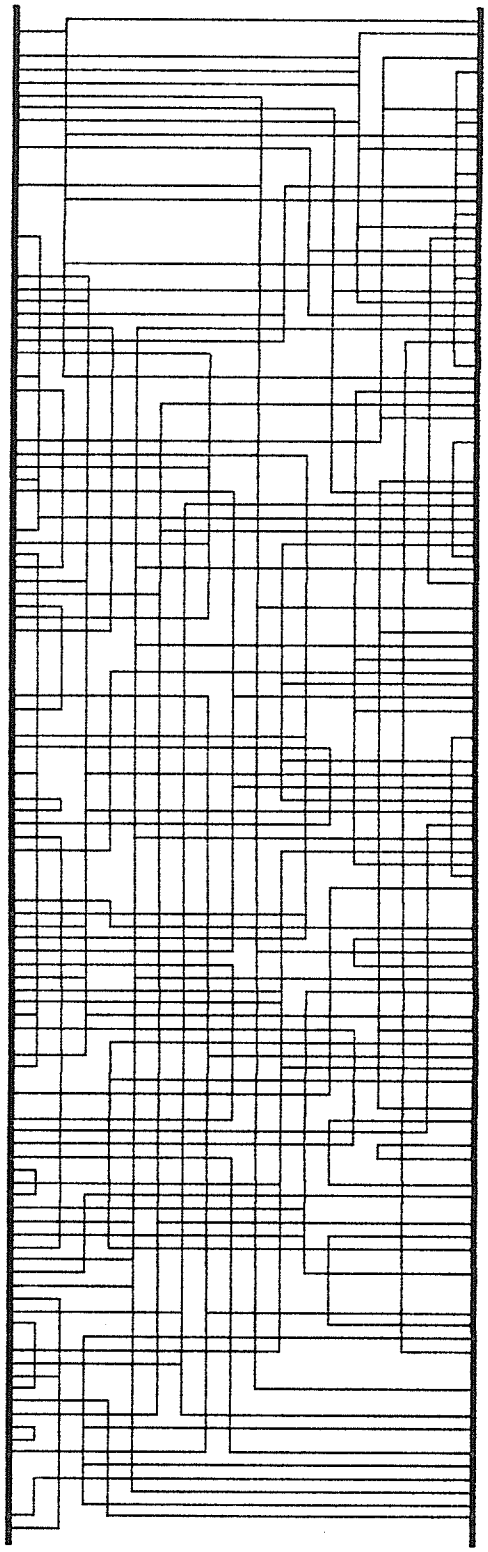


Fig. 33. Non-dogleg realization of Example 9 (channel height: 18).

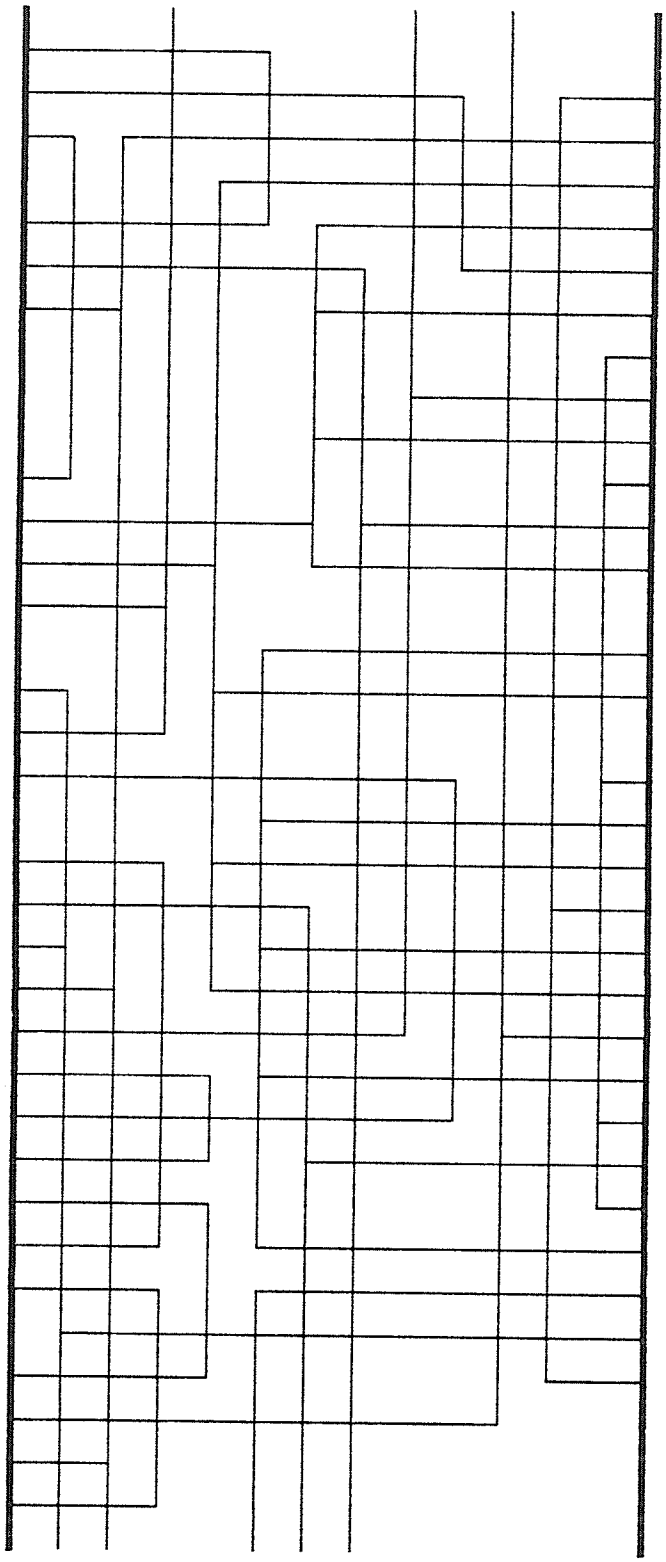


Fig. 34. Non-dogleg realization of Example 10 (channel height 12).

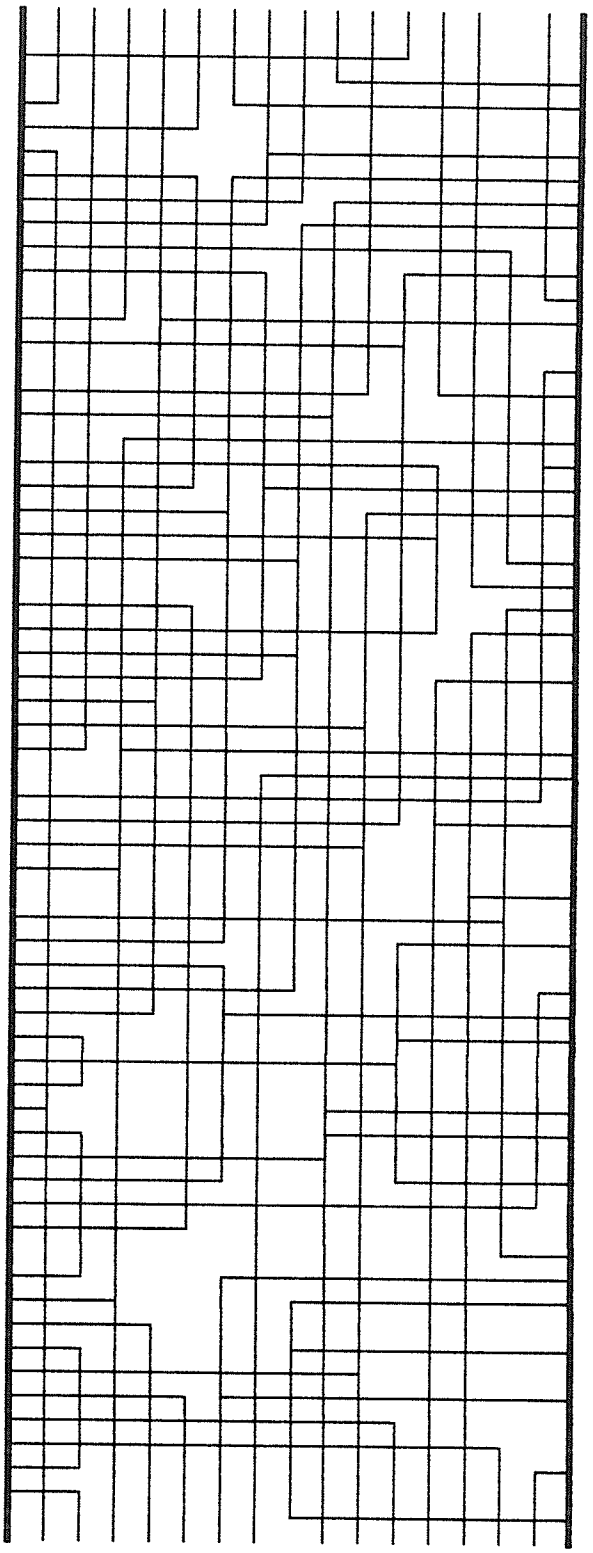


Fig. 35. Non-dogleg realization of Example 11 (channel height: 15).

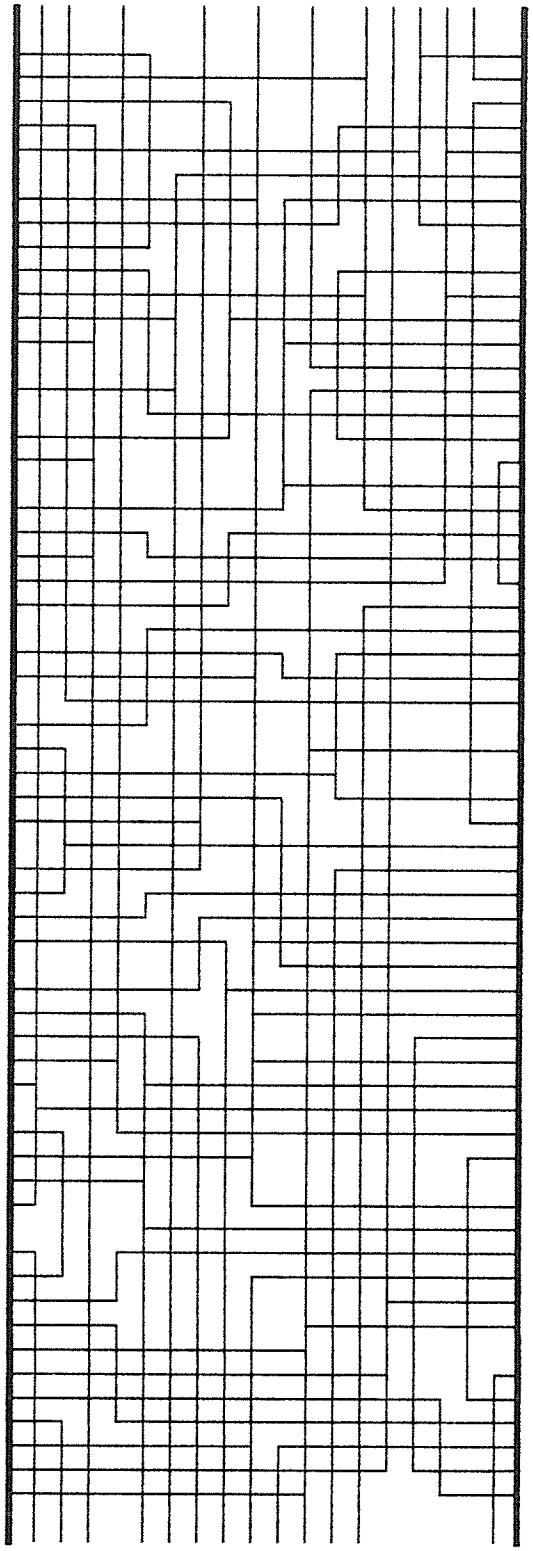


Fig. 36. Non-dogleg realization of Example 12 (channel height: 18).

Among the twelve examples, Example 6 is commonly known as Deutsch's Difficult Example [DEU76] and has been widely used in literature as a benchmark test case for channel routing. Although Deutsch's Difficult Example contains fewer than the typical number of nets (72 vs. about 100), it is difficult because of its severe vertical constraints. It has a channel ordering of 23, while the channel density is only 19. Furthermore, the minimum non-dogleg channel height was found to be 28 tracks, which is far above the ordering and density of the channel. The 28-track minimum non-dogleg realization was first obtained by Deutsch using a branch-and-bound technique [KER73] after four hours of computation on an HP2100 minicomputer [DEU76]. But, the new algorithm was able to route this difficult example in minimum channel height in less than 1.3 seconds (including the reading of the net list and the writing of the routing results). The 28-track non-dogleg solution has only been obtained by one other routing algorithm [YOS82] that does not require an explicit or implicit exhaustive search (branch-and-bound techniques are considered as implicit exhaustive searches).

Moreover, since the algorithm is able to adapt the priority function according to the characteristics of the channels, the algorithm is relatively insensitive to the values of the weighting factors. For example, varying the weighting factor $N_{distance}$ from 2 to 10 has no effect on the routing results. This feature allows the algorithm to route different channels without user intervention.

3.7 Summary

Experimental results presented in this chapter have shown that the non-dogleg channel routing algorithm has very good performance. Since the algorithm adapts itself according to the characteristics of the channel, the algorithm is able to produce optimal or very near optimal results for a wide range of practical channel routing problems. In all twelve examples tested, the algorithm was able to route the channels in no more than one track above the optimal channel heights. Furthermore, the algorithm performs equally well for channels with ordering above or below density, and is fairly insensitive to the values of the weighting factors. A comprehensive description of the algorithm and the corresponding results is provided in [TSK87a]

CHAPTER IV
DOGLEG EXTENSION

In Chapter III, the non-dogleg channel routing algorithm was introduced. In the definition of the channel routing problem in Section 3.1.1, each net in the channel is limited to have at most one horizontal trace, and hence occupy at most one horizontal track. Very often, however, such requirement is too restrictive. In this chapter, an extension to the non-dogleg algorithm will be described. The extension relaxes the restriction to allow doglegging of the horizontal traces at terminal positions. A net is thus allowed to span several horizontal tracks. In the following sections, the motivations and tradeoffs of introducing doglegs, the development and implementation of the dogleg channel routing algorithm, and the experimental results will be described.

4.1 Motivation and Tradeoffs of Introducing Doglegs

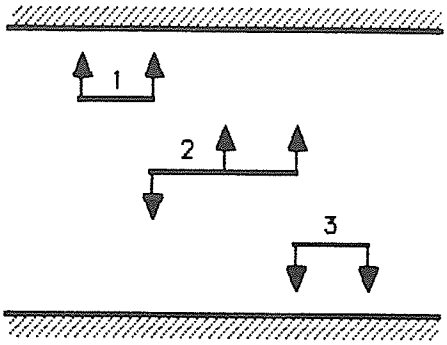
As described in Section 3.1.2, a dogleg splits the horizontal trace of a net into two or more horizontal segments on different horizontal tracks. The motivation of introducing doglegs is twofold. First, since the lower bound on the channel height is determined by both the channel density and the channel ordering, a long constraint chain in the vertical constraint graph may prevent the channel from being routed at or near optimal channel height. Introducing doglegs would allow those long vertical constraint chains to be broken. This is particularly important in channels where the ordering is higher than the density.

Consider the channel shown in Fig. 37a. Without doglegs the 3-track realization shown in Fig. 37b is already the optimal, because the channel ordering of 3 is higher than the channel density of 2, which bounded the minimum channel height at 3 tracks. However, when doglegs are used, the same channel can be routed in 2 tracks as shown in Fig. 37c, where the channel ordering is reduced to 2, and the lower bound on the channel height is determined by the channel density of 2.

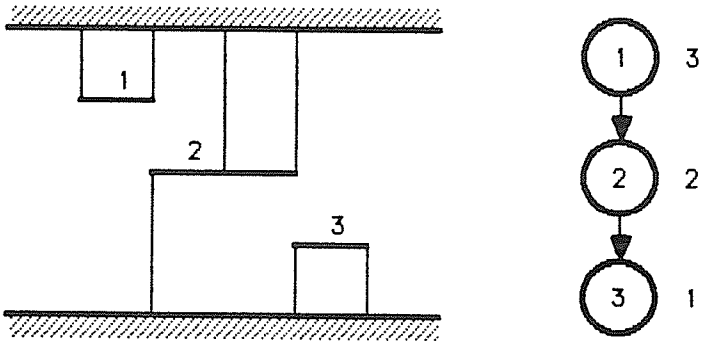
Another motivation of using doglegs is that, doglegs would enable routing of channels with vertical constraint loops, which would otherwise be unroutable. In Fig. 38a, each net in the channel requires the other net to be routed above and below it at the same time. This situation corresponds to a directed cycle in the vertical constraint graph. Without doglegs, this channel would be unroutable. Using doglegs, the directed cycle in the vertical constraint graph can be broken, and the channel is routed as shown in Fig. 38b.

The advantages of introducing doglegs are thus fairly clear. Doglegs enable us to (i) reduce long vertical constraint chains, and (ii) break vertical constraint loops. However, the use of doglegs also involves tradeoffs that must be considered. First, without doglegs, the channel realization always uses the minimum number of vias, while the introduction of each dogleg would add one or two additional vias. Since vias increase the interconnection resistance and capacitance, and reduce the circuit reliability, the number of vias, and hence, the number of doglegs, should be as few as possible.

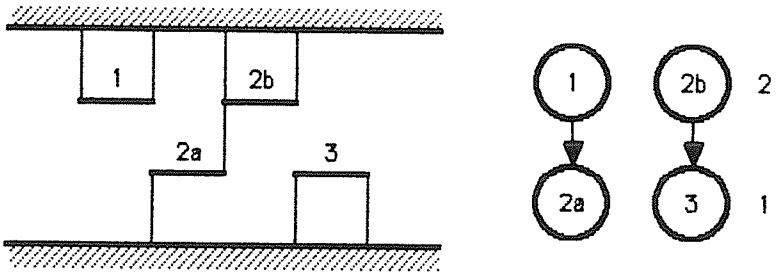
Another tradeoff involved in the use of doglegs is that, each dogleg increases the local density of the channel by one at the vertical track where the dogleg is placed. Since the routing process depends not only on the



(a)

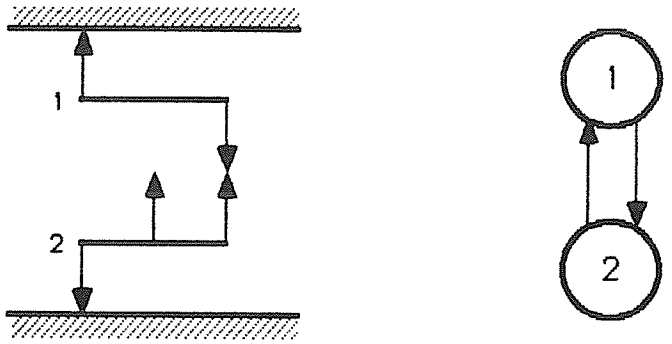


(b)

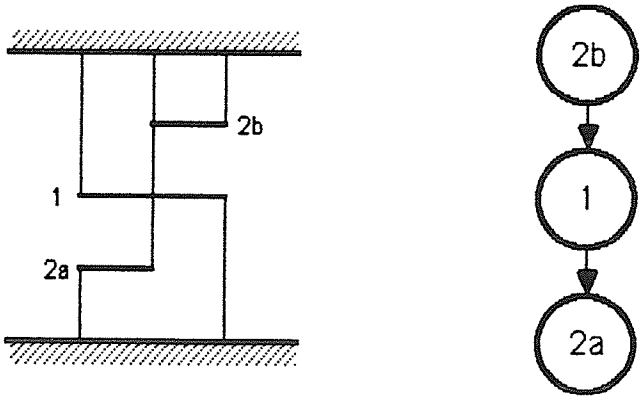


(c)

Fig. 37. An example illustrating the advantage of doglegs. (a) Graphical representation; (b) Non-dogleg realization; (c) Dogleg realization.



(a)



(b)

Fig. 38. An example unroutable without dogleg.
(a) A channel with a constriant loop; (b) Realization and vertical constriant graph of the channel using doglegs.

maximum density of the channel, but also on the local densities and the span (the number of vertical tracks having a local density equals to the channel density), increasing the local densities could complicate the routing process and result in sub-optimal realizations. Therefore, although doglegs could produce better solutions for certain channels and even allow channels with constraint loops to be routed, they should be used only when necessary.

4.2 Dogleg Channel Routing Algorithm

In light of the above considerations, the new dogleg channel routing algorithm doglegs only at the terminal positions of a net. That is, the algorithm would not dogleg a net unless there is already a via. This way, the number of vias is limited to at most 2T vias, where T is the total number of terminal connections in the channel, excluding the endpoint connections. Moreover, the algorithm would not extend a net beyond its original endpoints or permit its horizontal segments on different tracks to overlap. Such routing style can only increase the local densities of the channel. While sometimes beneficial, the incidence of such situations does not justify the complication of the routing process.

In general, doglegging at terminal positions alone has the following advantages: (i) it eliminates the need of additional vertical tracks for the doglegs, (ii) the resulting horizontal trace is always the shortest, and (iii) it avoids such cases as routing off the ends of the channel as with the Greedy Channel Router [RIV82].

4.2.1 Basic Dogleg Channel Routing Algorithm

The following is the basic structure of the dogleg channel routing algorithm.

[Basic Dogleg Channel Routing Algorithm]:

```
BasicDoglegRouter;
begin
    order the nets to be split;
    for each net meeting selection criteria do
        begin
        for each terminal in the net (excluding endpoints) do
            if terminal should be split then
                split the net at the selected terminal position;
        assign doglegging priority to the resulting subnets;
        end;
    non-dogleg channel router;
end; {BasicDoglegRouter}
```

□

The basic idea of the above algorithm is to first split the nets into subnets at selected terminal positions where doglegging may improve the solution, then use the previously developed non-dogleg channel routing algorithm to complete the routing. The main advantage of this approach is that, rather than developing an entirely new algorithm from scratch, previous research can be capitalized.

In the above dogleg channel routing algorithm, the nets are first ordered. Then one by one the nets are examined to determine if doglegging the net would reduce the ordering of the channel. If so, each terminal in the net (excluding the endpoints) is examined to decide whether the net should be split at that terminal position. This process continues until all the nets are examined. The result is a modified netlist that is equivalent to the original netlist but with certain nets divided into two or more subnets.

After a net is split, doglegging priorities are assigned to the resulting subnets as negative distance values. A negative value is used since the net priority function contains a term MaxReadyNetdistance-Distance(Mother,Net). When the distance value between two subnets is large and negative, the subnets would have higher net priorities compared with the others. Thus, in the net selection processes those subnets would be more likely to be routed together, and be re-merged back into a longer subnet. This distance value is therefore equivalent to a priority factor for doglegs. The more negative this doglegging priority, the less doglegs would be used. When this value become very negative, the dogleg algorithm reverts back to the non-dogleg algorithm.

4.2.2 Net Ordering

As noted previously, each dogleg increases the local density of the vertical track where the dogleg occurs by one. In order to keep the minimum height of the modified channel to no greater than that of the original channel, doglegging of a net is not allowed if the doglegging would increase the channel density above the original channel density. Thus the number of doglegs that can be placed in any vertical track is limited to the channel density minus the original local density of that vertical track. Being limited by the number of possible doglegs, the order the nets are doglegged becomes important. If the nets were doglegged indiscriminately, the channel density could be reach easily, leaving some long vertical constraint chains unbroken.

Two net ordering schemes were tested in reducing the ordering of the channel. The first scheme arranges the nets in descending order of their ordering numbers, so that nets with higher ordering numbers are doglegged

first. The second scheme arranges the nets in descending order of their number of ancestors in the vertical constraint graph. The reason is that, doglegging a net results in subnets having lower ordering numbers. The reduced ordering numbers are then propagated to the ancestors of the subnets. Thus, nets with more ancestors when reduced would result in more nets having reduced ordering numbers. Using a simplified dogleg router that sequentially splits all nets at all terminal positions as long as the local densities do not exceed the original channel density, the two net ordering schemes were tested. The channel orderings and channel heights of the twelve example channels were compared. The results of no net ordering (columns two and three), ordering by net ordering numbers (columns four and five), and ordering by number of ancestors (column six and seven) are listed in Table 3.

From the table, it can be seen that the channel orderings were different with and without net ordering (columns two and four). It thus showed a need for net ordering. Moreover, although the channel orderings were the same with no ordering (column 2) and with ordering by the number of ancestors (column six), the resulting channel heights were better in the latter case. Thus, ordering by the number of ancestors showed superior results. In fact, even with the simplified algorithm using no net and terminal selections, ten of the twelve examples were routed in density, and the remaining two examples were routed within two tracks of densities. In particular, the Deutsch's Difficult Example (Example 6) was routed in 21 tracks, which is the same as that of the Dogleg Channel Router of [DEU76], and showed a great improvement over the 28 track optimal realization without doglegs.

Table 3. Comparison of net ordering schemes.

Example	No Net Ordering		By Ordering Numbers		By Number of Ancesters	
	Ordering	Height	Ordering	Height	Ordering	Height
1	4	5	4	5	4	5
2	2	4	2	4	2	4
3	3	3	3	3	3	3
4	5	5	5	5	5	5
5	6	16	6	19	6	18
6	6	21	8	22	6	21
7	3	20	3	20	3	20
8	6	12	6	12	6	12
9	6	17	6	18	6	17
10	6	12	6	12	6	12
11	4	17	4	16	4	16
12	7	17	7	17	7	17

4.2.3 Net and Terminal Selection

As described in the structure of the dogleg channel routing algorithm in Section 4.2.1, after the nets are ordered each nets is examined to determine whether doglegging the net would likely to reduce the channel ordering, and if so which terminal should be doglegged. After experimented with several net and terminal selection schemes the following scheme was chosen.

for each vertex in the vertical constraint graph **do**
  if the vertex has at least one ancestor and one descendent **then**
    for $t_i = t_2$ **to** t_{n-1} **do**
      if t_{i-1} and t_{i+1} are connected to opposite boundaries **and**
        Density(t_i) < ChannelDensity **then**
        split the net at terminal position t_i;

The above selection scheme examines the corresponding vertex of each net in the vertical constraint graph. If the net has at least one ancestor and one descendent, the net is in a situation similar to that of net 2 in Fig. 37. By splitting the net into subnets, the ordering numbers of the ancestors are reduced by one. Furthermore, the dogleg should occur in a terminal position t_i where the previous terminal connection t_{i-1} and the next terminal connection t_{i+1} are to opposite channel boundaries. Such a terminal position separates the portion of the net that should be placed above the descendent net and the portion that should be placed below the ancestor net. Doglegging the net at such terminal positions would break the constraint chain into two disjoint chains, and reduce the ordering.

4.2.4 Complete Dogleg Channel Routing Algorithm

The complete dogleg channel routing algorithm including the net ordering scheme, and the net and terminal selection scheme can be described in pseudo-code as follows.

[Complete Dogleg Channel Routing Algorithm]:

```
CompleteDoglegChannelRouter;
begin
    order the nets in descending number of ancestors;
    for each vertex in the vertical constraint graph do
        if the vertex has at least one ancestor and one descendent then
            begin
            for t_i = t_2 to t_{n-1} do
                if t_{i-1} and t_{i+1} are connected to opposite boundaries and
                    Density(t_i) < ChannelDensity then
                    split the net at terminal position t_i;
            distance between all subnets in the same net = DoglegPriority;
            end;
    non-dogleg channel router;
end; {CompleteDoglegChannelRouter}                                    □
```

4.3 Implementation

As described, the only different between the dogleg routing algorithm and the non-dogleg routing algorithm is in the Doglegger section. After the Filer read in the input netlist, the Doglegger modifies the netlist splitting the nets into subnets at selected terminal positions, then the Non-Dogleg Router takes the modified netlist and performs a non-dogleg routing. As shown in the structured chart of the Doglegger in Appendix A, Fig. A2, the Doglegger is consisted of four main processes. First, the Doglegger finds the local density

at each vertical track to ensure that the net splits would not increase the channel density. Then, the vertical constraint graph is created to determine the number of ancestors for each net and the input netlist is ordered by their number of ancestors in the vertical constraint graph. After the netlist is ordered, the net and terminal selection process determines which nets should be split and at which terminal positions. When a net is split, the subnets created are appended to the end of the netlist structure. After all the nets are examined and all the subnets are created, the modified netlist is passed on to the Non-Dogleg Router section to complete the routing.

4.4 Efficiency of the Dogleg Routing Algorithm

In this section, the efficiency of the dogleg router will be discussed in terms of the CPU time required versus the complexity of the channel. Two curves labelled DOGLEG1 and DOGLEG2 were shown in Fig. 24 along with the curve NON-DOGLEG for the non-dogleg router. The curve labelled DOGLEG1 showed the CPU time required versus the number of nets in the original netlist while the curve labelled DOGLEG2 showed the CPU time required versus the number of subnets in the modified netlist.

An analysis of the DOGLEG1 curve showed that the complexity of the algorithm is about $O(n^2)$ to (n^3), where n is the number of nets in the original netlist. However, when doglegs are used, the complexity of a channel depends not only on the number of nets in the channel, but also on the number of doglegs. The curve DOGLEG2 shows the CPU time required when the complexity of the channel is measured by the number of subnets, which is equal to the number of nets in the channel plus the number of possible

doglegs. As can be seen from Fig. 24, the DOGLEG2 curve is fairly linear. The algorithm is thus able to handle fairly complex channels with reasonable running time.

4.5 Experimental Results

Using mother net priorities of M_{length}=10 and $M_{ordering}$=25, net priorities of N_{order}=10, N_{length}=15 and $N_{distance}$=10, and a dogleg priority of -20, the results obtained by the new dogleg channel routing algorithm are summarized in Table 4, and the channel realizations are shown in Figs. 39-50. In Table 4, columns two and three are the densities and orderings of the original examples. Column four is the reduced ordering numbers obtained from the simplified algorithm repeated here for comparison. The remaining columns are results obtained using the new implementation. Note that the ordering numbers were further reduced using the new algorithm (examples 5, 8 and 10). More importantly, the number of net splits and the number of doglegs used were greatly reduced. Since each dogleg requires an additional via, the reduced number of doglegs showed that the net and terminal selection schemes were effective in determining where the doglegs should be placed.

From Table 4, the dogleg channel routing algorithm was able to route 75% (nine out of twelve) of the examples in a channel height equals to their density, 17% (two out of twelve) in a channel height equals to one track above their density, and 8% (one out of twelve) in a channel height equals to two tracks above their density. These results were obtained using a single set of weighting factors. By adjusting the weighting factors slightly, eleven examples were routing in a channel height equals to their density, while

Table 4. Results of the new dogleg channel routing algorithm.

Example	Original Statistics		Order by Ancestors	Final Routing Results			
	Density	Ordering		New Ord	Net Split	Dogleg	Height
1	5	4	4	4	0	0	5
2	4	2	2	2	0	0	4
3	3	3	3	3	0	0	3
4	5	5	5	5	0	0	5
5	18	6	6	5	10	6	18
6	19	23	6	6	62	44	20
7	20	3	3	3	2	2	20
8	12	7	6	3	5	2	12
9	17	13	6	6	28	16	18
10	12	7	6	3	3	2	12
11	15	4	4	4	6	3	17
12	17	9	7	7	11	8	17

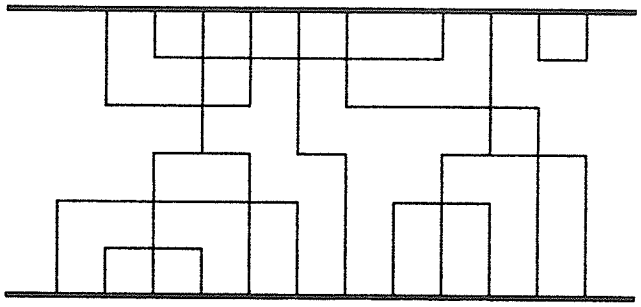


Fig. 39. Dogleg realization of Example 1 (channel height: 5).

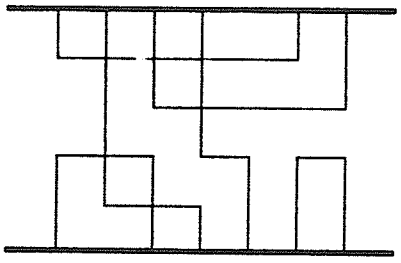


Fig. 40. Dogleg realization of Example 2 (channel height: 4).

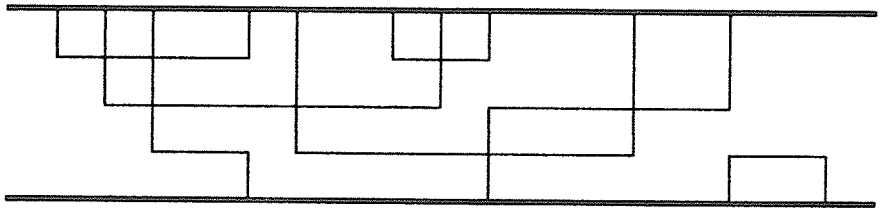


Fig. 41. Dogleg realization of Example 3 (channel height: 3).

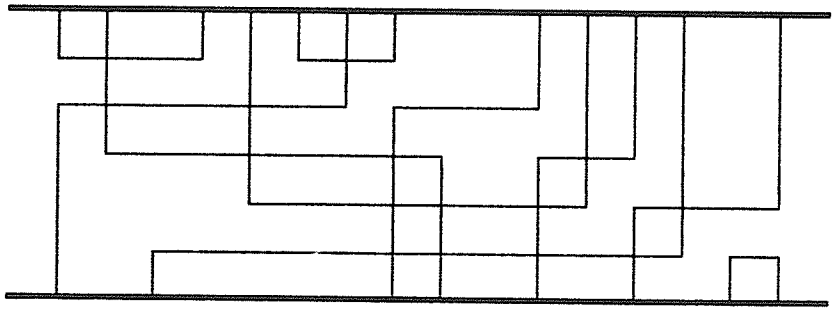


Fig. 42. Dogleg realization of Example 4 (channel height: 5).

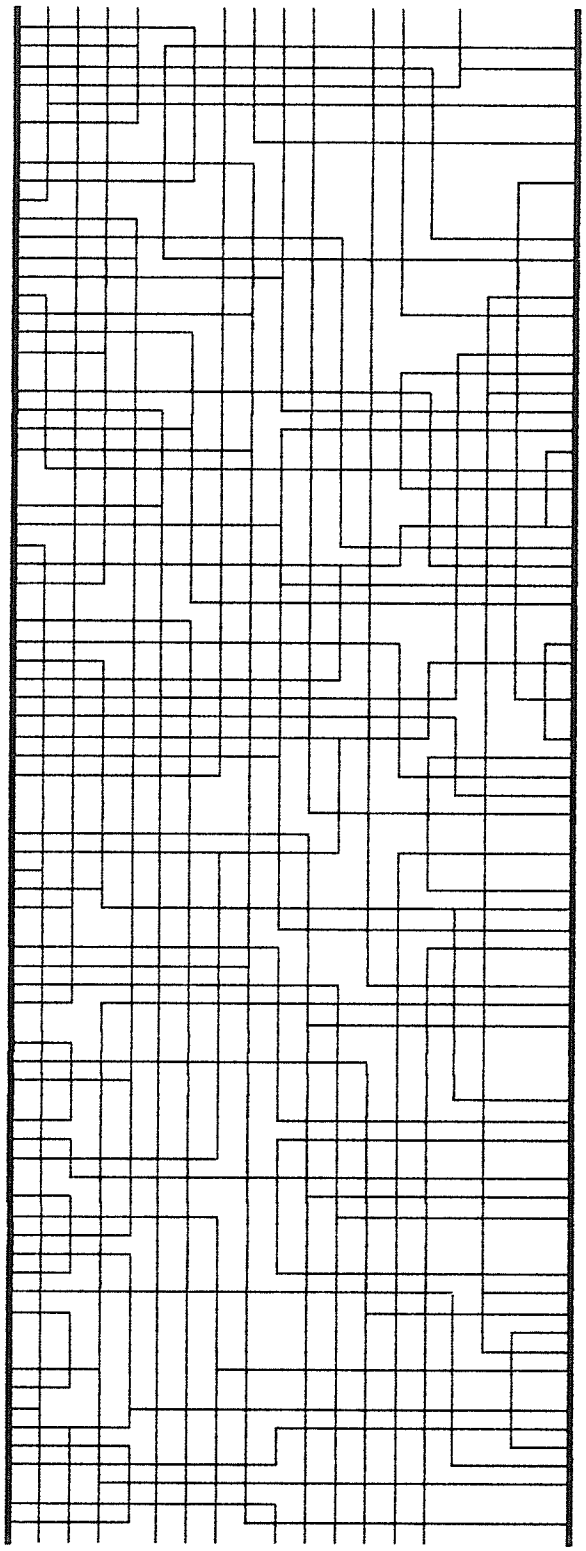


Fig. 43. Dogleg realization of Example 5 (channel height: 18).

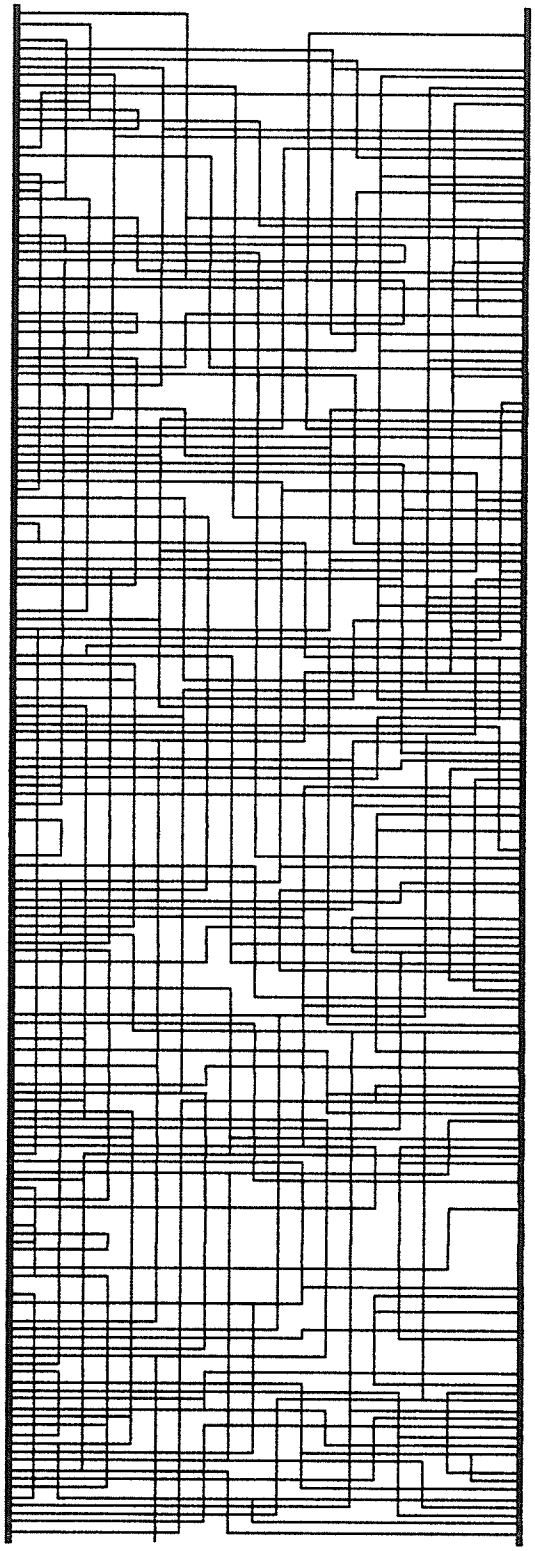


Fig. 44. Dogleg realization of Example 6 (channel height: 20).

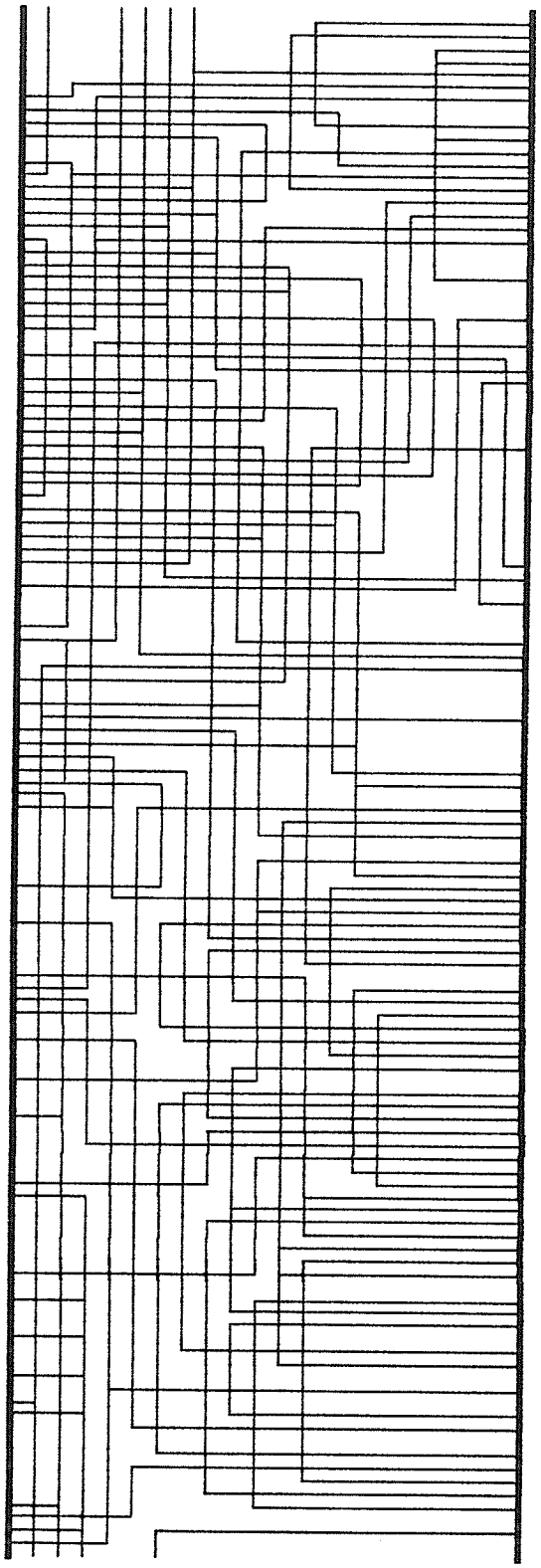


Fig. 45. Dogleg realization of Example 7 (channel height: 20).

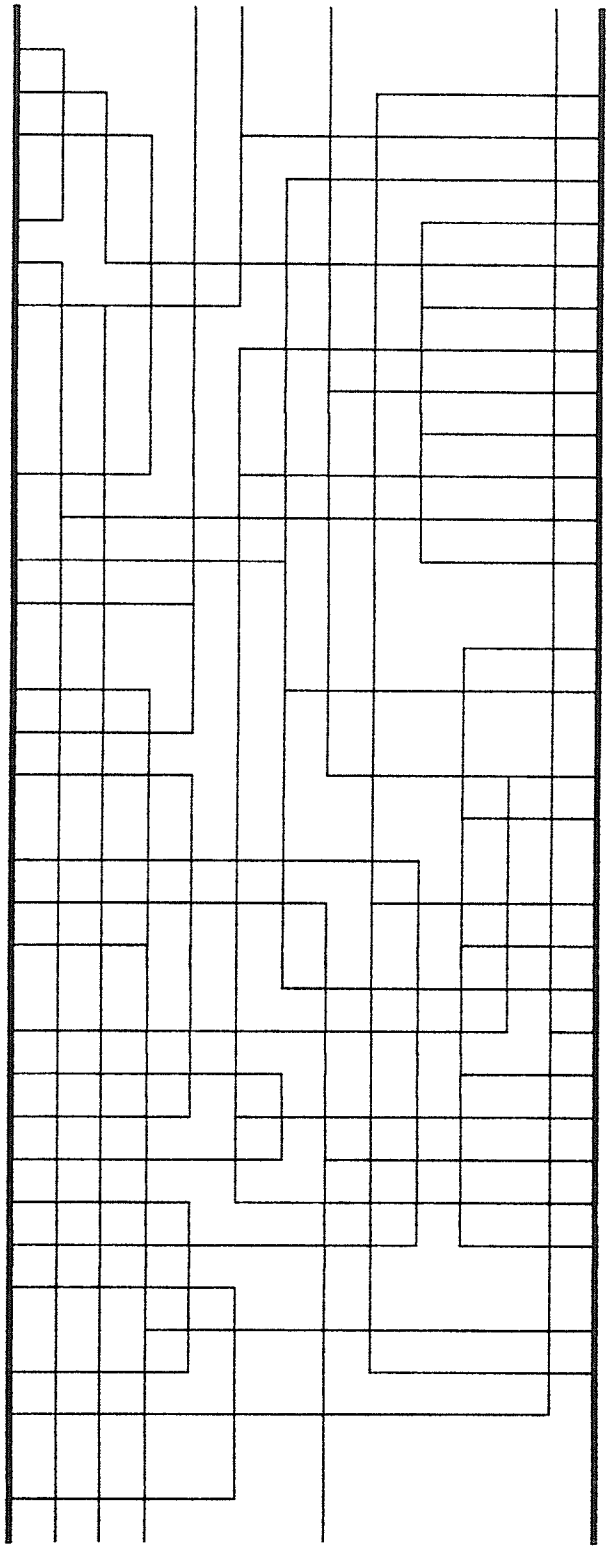


Fig. 46. Dogleg realization of Example 8 (channel height: 12).

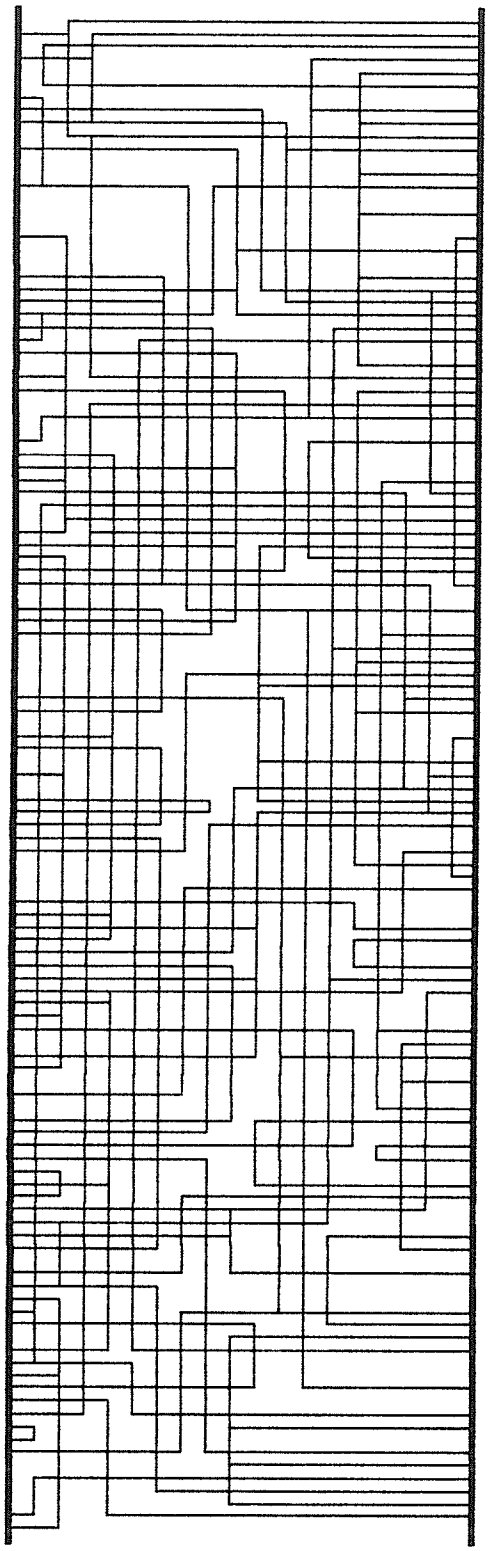


Fig. 47. Dogleg realization of Example 9 (channel height: 18).

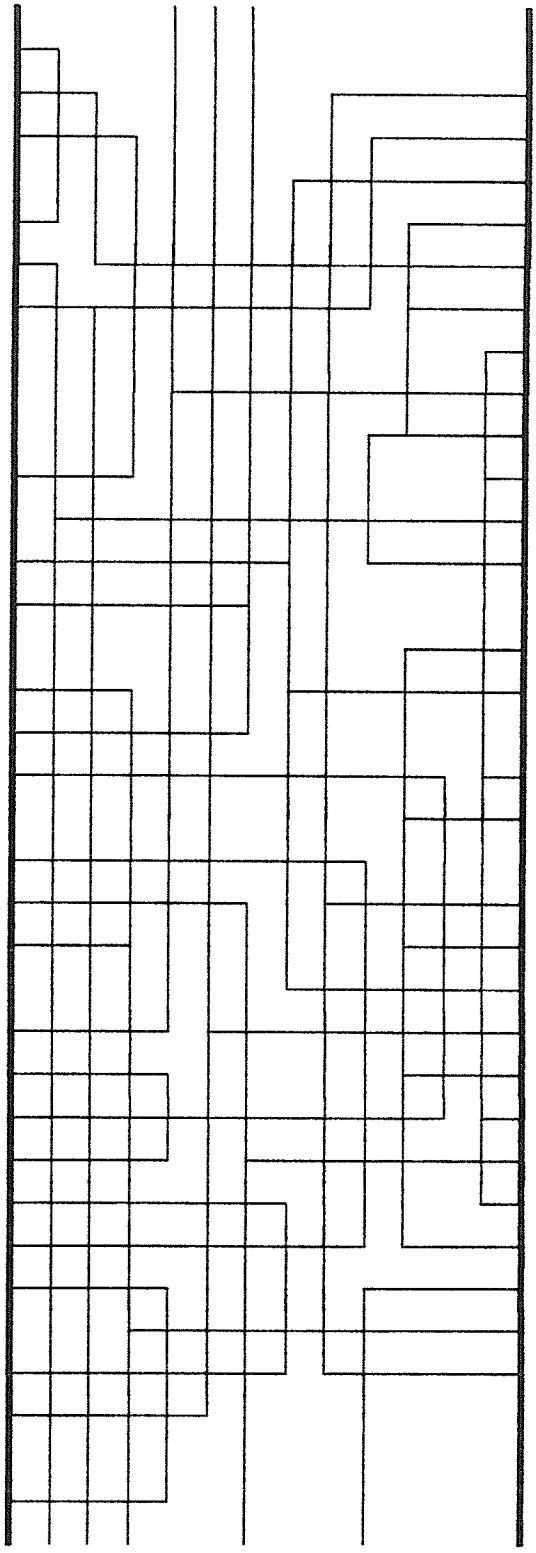


Fig. 48. Dogleg realization of Example 10 (channel height: 12).

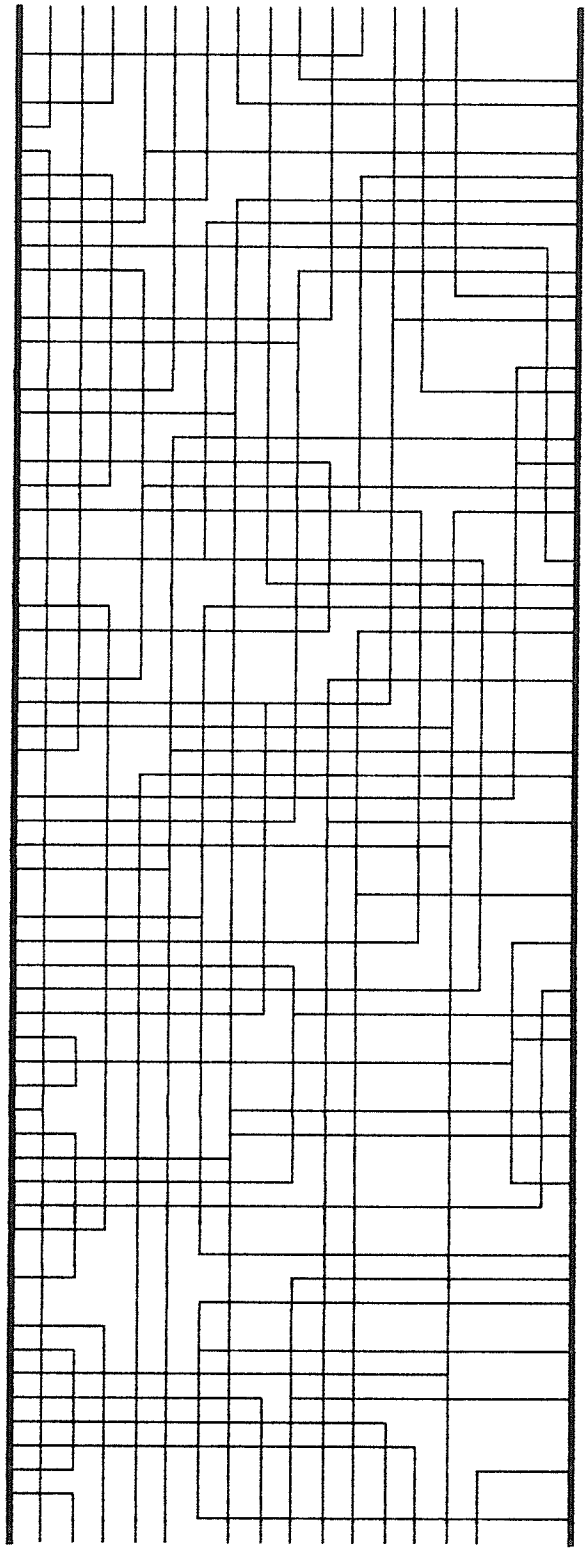


Fig. 49. Dogleg realization of Example 11 (channel height: 17).

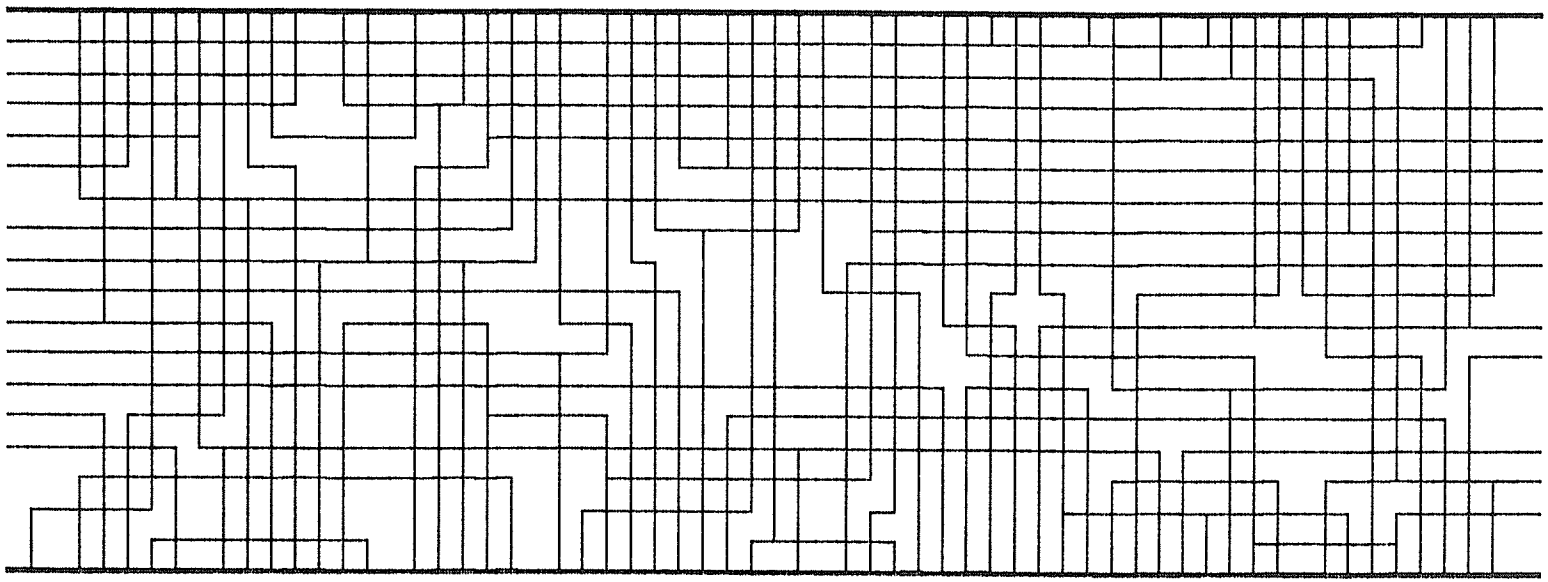


Fig. 50. Dogleg realization of Example 12 (chanel height: 17).

Deutsch's Difficult Example was routed in 20 track, only one track above the minimum channel height. This result is comparable to any published dogleg channel routers of the same nature, that is, routers that dogleg only at terminal positions. Although it has not been proven, the 20-track realization may be the minimum channel height achievable by routers that place dogleg at terminal positions only. More specifically, the Dogleg Channel Router of [DEU76] required 21 tracks, the Efficient Channel Router of [YOS84] required 20 tracks, and the Greedy Channel Router of [RIV82] also required 20 tracks but it allows doglegging at all available vertical tracks and routing off the end of the channel. Thus, the performance of the new dogleg channel routing algorithm is at least as good as, and in many cases better than, other published algorithms of the same class.

4.6 Vertical Constraint Loop Handling

As discussed in Section 4.1, the use of doglegs would enable us to route channels with constraint loops that would otherwise be unroutable. In the new dogleg channel routing algorithm described in Section 4.2.4, a net is split only if the split would not increase the channel density. For channels with vertical constraint loops, however, it may be necessary to introduce doglegs that would increase the channel density. For example, the channel shown in Fig. 38 has density and ordering both equal to 2, but the minimum channel height is 3. Therefore, the requirement that a net split must not increase the channel density needs to be dropped in order to cope with channels with vertical constraint loops.

Two additional examples, Example 13 and 14, have been obtained from [MAT72]. Both examples contain a vertical constraint loop that would be unroutable without doglegs. Using the new dogleg routing algorithm with the density check (Density(t_i)<ChannelDensity) dropped, Mother net priorities of N_{length}=10 and $N_{ordering}$=25, net priorities of N_{order}=10, N_{length}=15 and $N_{distance}$=20, and a dogleg priority of -50, the two examples have been routed. Moreover, Examples 1-12 have also been re-routed without the density check for comparison. The results are summarized in Table 5 and the realizations of Example 13 and 14 are shown in Figs. 51 and 52.

In Table 5, columns two and three are the original densities and orderings of the channels; columns four and five are the new densities and orderings of the channels when doglegs are introduced; column seven is the number of net splits, which determines the maximum number of doglegs that can occur; column eight is the actual number of doglegs used; and column nine is the resulting channel height. The difference between the number of net splits and the number of dogleg used is the number of subnet re-merges.

From the results shown in Table 5, it can be seen that the channel densities for four of the fourteen examples were increased above the original channel densities due to the relaxed terminal selection scheme (dropping of the density check). For channels with no vertical constraint loops (Example 1-12), the algorithm was able to re-merge the unnecessary net splits, and only the channel height of the Deutsch's Difficult Example (Example 6) was slightly increased by one track. For channels with vertical constraint loops (Example 13 and 14), Example 13 was routed in 19 tracks and Example 14 was routed in 17 tracks.

Table 5. Results of the new dogleg channel router with no density check

(Example 13 and 14 contain vertical constraint loops).

Example	Original Statistics		New Statistics		Results		
	Density	Ordering	Density	Ordering	Net Split	Dogleg	Height
1	5	4	5	4	0	0	5
2	4	2	4	2	0	0	4
3	3	3	3	3	0	0	3
4	5	5	5	5	0	0	5
5	18	6	18	5	10	6	18
6	19	23	20	6	67	40	21
7	20	3	20	3	2	2	20
8	12	7	12	3	5	2	12
9	17	13	18	6	29	16	18
10	12	7	12	3	3	2	12
11	15	4	15	4	6	3	17
12	17	9	18	7	12	8	17
13	18	-	20	6	16	9	19
14	17	-	17	4	12	9	17

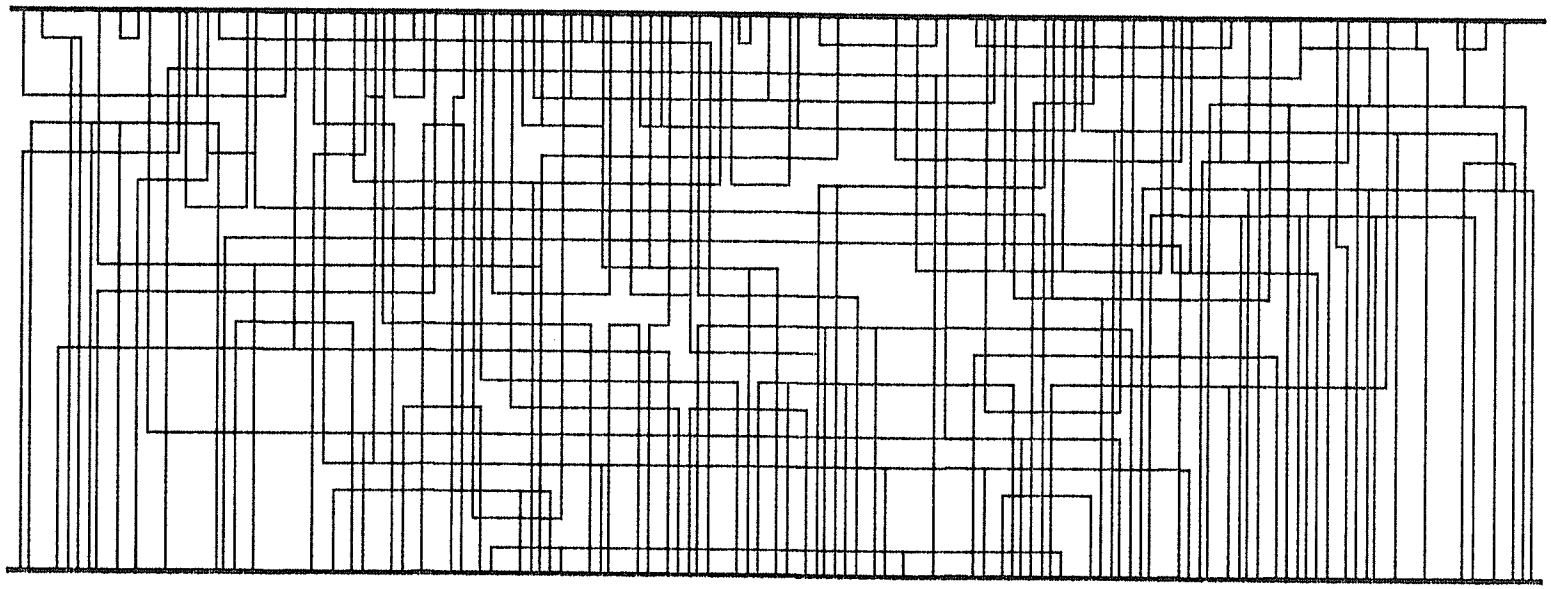


Fig. 51. Dogleg realization of Example 13,
a channel with a vertical constraint loop (channel height: 19).

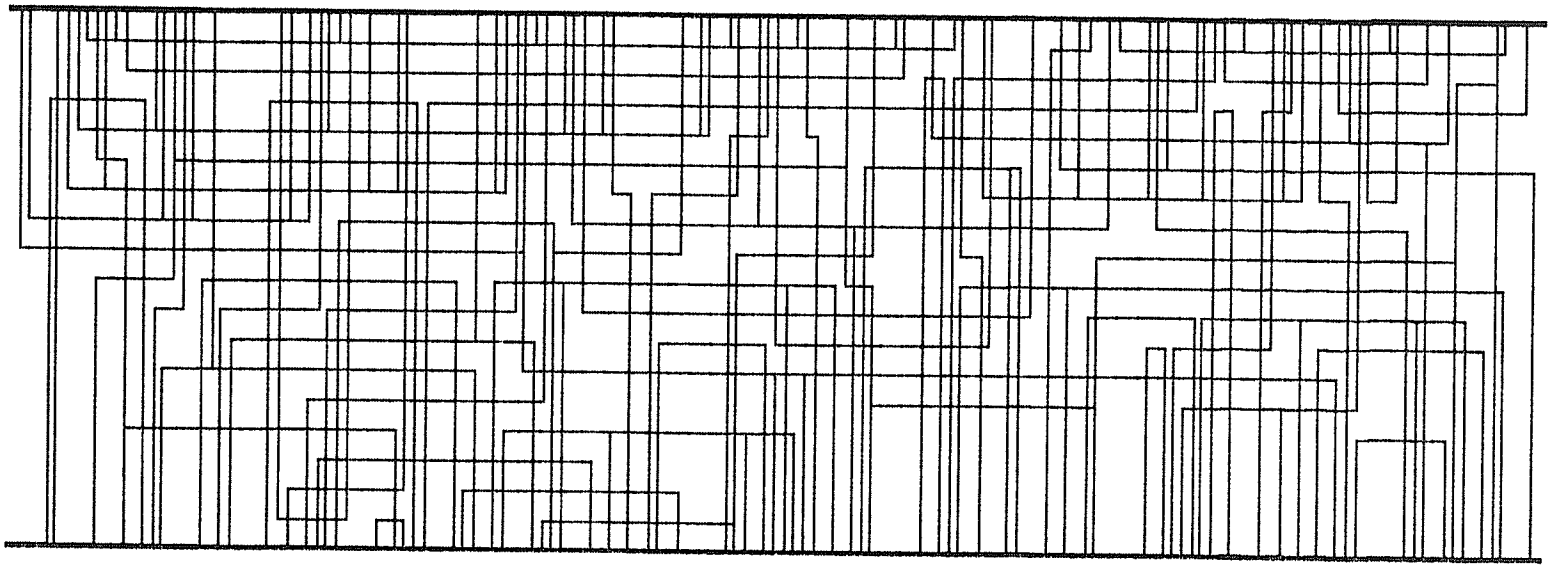


Fig. 52. Dogleg realization of Example 14,
another channel with a vertical constraint loop (channel heihgt: 17).

Comparing the results of Example 13 and 14 obtained by the new dogleg channel router with other routers, the channel router of Mattison [MAT72] routed the two examples in 23 and 20 tracks respectively, and the Dogleg Channel Router [DEU76] required 19 and 17 tracks, respectively. The new algorithm has thus shown a significant improvement over the channel router of Mattison and performed as well as the Dogleg Channel Router.

4.6.1 Applicability of the Algorithm

The new dogleg channel routing algorithm is able to handle a large class of channels with vertical constraint loops. However, there are certain cases that the algorithm cannot handle. The limitation is due to the fact that the nets are doglegged at terminal positions only and are not allowed to be extended beyond their original endpoints. This approach has its advantages as discussed in Section 4.2, but as an example, the channel shown in Fig. 53 would be unroutable because it requires an additional vertical track off the right end of the channel, and requires a net to be extended from vertical track 1 to vertical track 3, then "turn back" to vertical track 2, doglegging at non-terminal positions and extending beyond its original endpoints.

The class of channels handled can be defined more precisely as follows: if a channel contains a vertical constraint loop of n nets, at least n-1 of those nets must each has at least one unconstraining terminal in between every pair of constraining terminals. An unconstraining terminal is a terminal that if removed would not alter the constraint loop. That is, the constraint loop would still exist regardless of the unconstraining terminals.

Fig. 53. A channel unroutable by the new channel router.
(a) A channel not handled by the new Dogleg routing algorithm;
(b) A realization requiring doglegs at non-terminal positions.

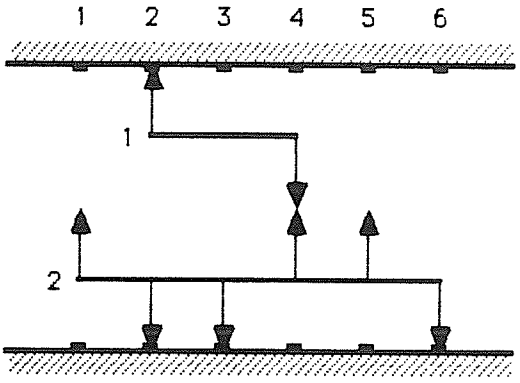


Fig. 54. An illustration of constraining
and unconstraining terminals.

In Fig. 54, terminals 2 and 4 of nets 1 and 2 are constraining terminals, removing any one of those terminals would disrupt the constraint loop. The rest of the terminals, terminals 1, 3, 5 and 6 of net 2, are unconstraining terminals, removing them would have no effect on the constraint loop. Furthermore, terminal 3 of net 2 is an unconstraining terminal in between two constraining terminals (2 and 4). If a dogleg is inserted at this terminal position, the constraint loop would be reduced into two disjoint constraint chains. It is such terminals that allow the channel to be routed with doglegs at terminal positions alone.

4.7 Summary

The dogleg channel routing algorithm presented in this chapter is capable of routing channels with no vertical constraint loops at or near optimal channel heights. Moreover, the algorithm is able to handle a large class of channels with vertical constraint loops. Among all the examples tested, the algorithm was able to complete the routing in no more than two tracks above the minimum channel height. In particular, the Deutsch's Difficult Example was routed in 20 tracks, which is same as the best published result obtained by routers of the same nature (dogleg at terminal positions only). The algorithm has thus demonstrated the efficiency of a graph based heuristic approach to the NP-complete channel routing problem. A description of this algorithm and the corresponding results is provided in [TSK87b].

CHAPTER V
CONCLUSIONS AND RECOMMENDATIONS

Routing is a very challenging problem in VLSI layout design because of its extreme complexity and its tremendous influence on the quality and performance of the resulting circuit. Since over half of the die area can be occupied by interconnections, and since the cost and performance of the circuit rely heavily on the die size and the interconnection length, an efficient router is critical to the success of the circuit.

Many routing algorithms have been developed for VLSI layout designs. Maze-running and line-search algorithms have been widely used in layout design systems, particularly in printed circuit boards (PCBs) design workstations. Their popularity can be attributed to their generality and their ability to find the optimal interconnection path if such a path exists. These algorithms, however, are not suitable for LSI and VLSI layout designs. They are not only inefficient in both space and time, but their inherent sequential nature of routing one net at a time may also result in undesirable routing patterns or excessive overflows. Due to the regular shapes of modules used in VLSI layout designs, particularly in gate array and standard cell designs, channel routing algorithms are much preferred.

A number of channel routing algorithms have been developed since the channel routing concept was introduced by Hashimoto and Stevens [HAS71]. To cope with the complexity of the channel routing problem, which is NP-complete, practical algorithms must employ heuristics. The heuristics are embedded in a mathematical model of the routing process. Such models

include graphs [YOS82] and probabilistic hill climbing [ROM84]. The graph-based model has been selected because of its relative simplicity and fairly accurate representation of the routing process. An attempt has been made in this thesis to develop a heuristic channel routing algorithm based on the graph model. The algorithm was aimed at the routing of a general class of channels that frequently arises in gate array, standard cell, and building block layout designs.

As demonstrated in the previous chapters, this thesis has contributed to general and technical knowledge by achieving the following results:

1. Studied the VLSI layout design problem, in particular, the VLSI channel routing problem and channel routing algorithms.

2. Developed a non-dogleg channel routing algorithm that is capable of finding near optimal solution for a general class of channels that arises very often in gate array, standard cell, and building block layout designs. The algorithm is applicable to channels with ordering above or below density. The heuristic used is flexible and allow different routing criteria to be incorporated.

3. Developed a dogleg channel routing algorithm that is capable of routing regular channels at or near density using doglegs. The dogleg channel routing algorithm is also capable of routing channels with a class of vertical constraint loops that would be unroutable without doglegs.

4. The performance of both channel routing algorithms have been demonstrated through fourteen examples obtained from previously published papers. The algorithms have been found to be better than or

comparable with most published channel routing algorithms. It has thus demonstrated that the concept of a graph based heuristic channel routing algorithm can be used efficiently in solving the NP-complete VLSI channel routing problem.

5. The experimental results have shown that dogleg routing is not universally better than non-dogleg routing. The percentage of examples routed in their minimum theoretical channel height (75%) is the same with and without doglegs. Moreover, for those examples that did not reach minimum channel height without doglegs, only those containing severe vertical constraints are routed in smaller channel heights with the use of doglegs. Therefore, whenever possible non-dogleg routing should be used because: (i) it is simpler and requires shorter running time than dogleg routing, and (ii) it may produces realizations that are electrically superior to those produced with dogleg routing.

The research done and the conclusions drawn from it are important to the theory and practice of CAD/CAE of VLSI layout designs. Through the study and research, our understanding of the channel routing problem has increased considerably. Many other important questions have been discovered through out the course of the research, the following areas are recommended to improve the research performed and presented in this thesis:

1. Other routing criteria such as minimum number of vias and minimum wire lengths could be incorporated into the heuristic to produce better routing patterns. Adaptive heuristic techniques should be used to reduce the sensitivity of the algorithm to channel characteristics.

2. With a given routing area, the more evenly the wires are distributed the more clearance the wires would have. Thus, the routing algorithms could be modified to produce more evenly distributed routing patterns. This may be achieved by routing from both the top and the bottom of the channel at the same time.

3. Although the number of vias should be kept to the minimum, the dogleg channel routing algorithm could be modified to allow doglegging at non-terminal positions in cases where minimum routing area is the main objective.

4. The dogleg channel routing algorithm should be modified to handle other types of vertical constraint loops. This problem can be partially solved by allowing doglegs at non-terminal positions. However, a more robust approach would be to find the constraint loops in the vertical constraint graph and develop an algorithm to break the loops. The cycle breaking algorithm should also try to keep the lengths of the resulting constraint chains to the minimum.

5. The channel routing algorithms can now handle channels with regular shapes only. However, the possibility of applying the concept of a graph based heuristic algorithm to general routing areas such as switchboxes should be investigated.

6. The algorithms are now limited to two routing layers and the wire traces on a layer are allowed to run in only one of two perpendicular directions. The possibility of extending the algorithms to allow three or more routing layers and relaxing the restriction on wiring directions should be investigated.

REFERENCES

[AHO83] A. V. Aho, J. E. Hopcroft and J. D. Ullman, Data Structures and Algorithms. Reading, Massachusetts: Addison-Wesley, 1983.

[AKE67] S. B. Aker, "A modification of Lee's path connection algorithm," IEEE Transactions on Electronic Computers, vol. EC-16, January, 1967, pp. 97-98.

[AND85] H. Andou, I. Yamamoto, Y. Koike, K. Shouji and K. Hirakawa, "Automatic Routing Algorithm for VLSI," 22nd Design Automation Conference, 1985, pp. 785-788.

[BRE76] M. A. Breuer, Design Automation of Digital Systems, Theory and Techniques, vol. 1. New York: Prentice-Hall, 1976.

[BRE80] M. A. Breuer and K. Shamsa, "A hardware router," Journal of Digital Systems, vol. 4, no. 4, Computer Science Press, 1980, pp. 393-408.

[BRE83] M. A. Breuer and A. Kumar, "A methodology for custom VLSI layout," IEEE Transactions on Systems, Man, and Cybernetics, vol. SMC-13, no. 4, July/August 1983, pp. 470-475.

[BUR85] M. Burstein and M. N. Youssef, "Timing influenced layout design," 22nd Design Automation Conference, 1985, pp. 124-130.

[BUR86] M. Burstein, "Channel routing," in Layout Design and Verification, Advances in CAD for VLSI, vol. 4, ed. T. Ohtsuki. New York: North-Holland, 1986. pp. 133-168.

[CHA86] K. C. Chang and H. C. Du, "A preprocessor for the via minimization problem," 23rd Design Automation Conference, 1986, pp. 702-707.

[CHE86] H. H. Chen and E. S. Kuh, "Glitter: A gridless variable-width channel router," IEEE Transactions on Computer-Aided Design, vol. CAD-5, no. 4, October 1986.

[CHO85] S. Chowdhury and M. A. Breuer, "The construction of minimal area power and ground nets for VLSI circuits," 22nd Design Automation

Conference, 1985, pp. 794-797.

[DEU76] D. N. Deutsch, "A 'dogleg channel router," 13th Design Automation Conference, 1976, pp. 425-433.

[FOS75] J. Foster, "Prerouting analysis programs," 12th Design Automation Conference, 1975, pp. 306-310.

[GAM81] A. Gamal and Z. Syed, "A stochastic model for interconnections in custom integrated circuits," IEEE Transactions on Circuits and Systems, vol. CAS-28, September 1981, pp. 888-894.

[GAR79] M. R. Garey and D. S. Johnson, Computers and intractability: A Guide to the Theroy of NP-Completeness, 2nd ed. San Francisco, CA: W. H. Freeman, 1979.

[GEY71] J. Geyer, "Connection routing algorithm for printed circuit boards," IEEE Transactions on Circuit Theory, vol. CT-18, January 1971, pp. 95-100.

[GOT86] S. Goto and T. Matsuda, "Partitioning, assignment and placement," in Layout Design and Verification, Advances in CAD for VLSI, vol. 4, ed. T. Ohtsuki. New York: North-Holland, 1986. pp. 55-98.

[HAM84] G. T. Hamachi and J. K. Ousterhout, "A switchbox router with obstacle avoidance," 21st Design Automation Conference, 1984, pp. 173-179.

[HAN66] M. Hanan, "On Steiner's problem with rectilinear distance," SIAM Journal of Applied Mathematics, 14, March 1966, pp. 255-265.

[HAN72] M. Hanan and J. M. Kurtzberg, "Placement techniques," in Design Automation of Digital Systems: Theory and Techniques, vol. 1, ed. M. A. Breuer. New York: Prentice-Hall, 1972. pp. 213-282.

[HAR82] N. Harada, "A new average interconnection length prediction method for masterslice LSI," IEEE International Symposium on Circuits and Systems, vol. 3, 1982, pp. 760-763.

[HAS71] A. Hashimoto and J. Stevens, "Wire routing by optimal channel assignment within large apertures," 8th Design Automation

Workshop, 1972, pp. 15-169.

[HAS82] J. E. Hassett, "Automated layout in ASHLAR: an approach to the problem of 'general cell' layout for VLSI," 19th Design Automation Conference, 1982, pp. 777-784.

[HAY83] F. Hayes-Roth, D. A. Waterman and D. B. Lenat, Building Expert Systems. New York: Addison-Wesley, 1983. pp. 169.

[HEL77] W. R. Heller, W. F. Mikhail and W. E. Donath, "Prediction of wiring space requirements for LSI," 14th Design Automation Conference, 1977, pp. 32-42.

[HEY80] W. Heyns, W. Sansen, and H. Beke, "A line-expansion algorithm for the general routing problem with a guaranteed solution," 17th Design Automation Conference, 1980, pp. 243-249.

[HIC83] P. J. Hicks, ed., Semi-Custom IC Design and VLSI. London, UK: Peter Peregrinus, 1983.

[HIG69] D. W. Hightower, "A solution to the line-routing problems on the continuous plane," 6th Design Automation Conference, 1969, pp. 1-24.

[HOE76] J. Hoel, "Some variations of Lee's algorithm," IEEE Transactions on Computers, vol. C-25, January 1976, pp. 19-24.

[HOR81] C. S. Horng & M. Lie, "An automatic/interactive layout planning system for arbitrarily-sized rectangular building blocks," 18th Design Automation Conference, 1981, pp. 293-300.

[JOO85] R. Joobbani and D. Siewiorek, "WEAVER: a knowledge-based routing expert," 22nd Design Automation Conference, 1985, pp. 266-272.

[JOS85] R. L. Joseph, "An expert systems approach to completing partially routed printed circuit boards," 22nd Design Automation Conference, 1985, pp. 523-528.

[KAJ83] Y. Kajitani, "Order of channel for safe routing and optimal compaction of routing area," IEEE Transactions on Computer-Aided Design, vol. CAD-2, no. 4, October 1983, pp. 293-300.

[KAT85] F. Kato and H. Shiraishi, "Efficient compaction technique for LSI layout," IEEE International Conference on Computer Design, 1985, pp. 646-649.

[KER73] B. W. Kernighan, D. G. Schweikert and G. Persky, "An optimum channel-routing algorihtm for polycell layouts of integrated circuits," 10th Design Automation Conference, 1973, pp. 50-59.

[KIN85] W. Kinsner, Computer-Aided Engineering of Printed Circuit Boards, Course notes, Microelectronics Centre and University of Manitoba, Winnipeg, Manitoba, Canada, July1985, 300pp.

[KIN86a] W. Kinsner, Computer-Aided Engineering of Electronic Circuits: An Introduction, MC86-2, Microelectronics Centre and University of Manitoba, Winnipeg, Manitoba, Canada, October 5, 1986, 82pp.

[KIN86b] W. Kinsner and X. Kong, Schematic Capture, Placement and Routing of PCBs and SMBs: Examples, MC86-1, Microelectronics Centre and University of Manitoba, Winnipeg, Manitoba, Canada, August 20, 1986, 80pp.

[KIN86c] W. Kinsner, Design Considerations in PCBs and SMBs, MC86-4, Microelectronics Centre and University of Manitoba, Winnipeg, Manitoba, Canada, October 3, 1986, 66pp.

[KIN86d] W. Kinsner and X. Kong, Geometry Extraction of Placed Nets and Routed Wires in PCBs and SMBs, MC86-3, Microelectronics Centre and University of Manitoba, Winnipeg, Manitoba, Canada, September 4, 1986, 44pp.

[KIN86e] W. Kinsner, Semicustom Integrated Circuit Design, Course notes, Microelectronics Centre and University of Manitoba, Winnipeg, Manitoba, Canada, December, 1986.

[KIN87] W. Kinsner, "Solution to NP-complete problems in VLSI placement and routing," Miconex Processings, 1987, 10 pp.

[KIR83] S. Kirkpatrick, C. D. Gelatt, Jr. and M. P. Vecchi, "Optimization by Simulated Annealing," Science, vol. 220, no. 4598, 13 May 1983, pp. 671-680.

[KON86] X. Kong, A Study of Routing Algorithms for Printed Circuit Boards and VLSI, M. Sc. Thesis, University of Manitoba, Winnipeg, Manitoba, Canada, 1986.

[KOR82] R. Korn, "An efficient variable-cost maze router," 19th Design Automation Conference, 1982, pp. 425-431.

[KUH86] E. S. Kuh and M. Marek-Sadowska, "Global routing," in Layout Design and Verification, Advances in CAD for VLSI, vol. 4, ed. T. Ohtsuki. New York: North-Holland, 1986. pp. 169-198

[LAP80] A. S. LaPaugh, "Algorithms for integrated circuit layout: an analytic approach," Technical Report MIT-LCS-TR-248, Ph. D. Thesis, Dept. of Electrical Engineering and Computer Science, MIT, Cambridge, MA, 1980.

[LEE61] C. Y. Lee, "An algorithm for path connections and its application," IRE Transaction on Electronic Computers, vol. EC-10, 1961, pp. 346-365.

[LI83] J. Li, "Algorithms for gate matrix layout," IEEE International Symposium on Circuits and Systems, 1983, vol. 3, pp. 1013-1016.

[LI84] J. T. Li, C. K. Cheng, M. Turner, E. S. Kuh and M. Marek-Sadowska, "Automatic layout of gate arrays," IEEE Custom Integrated Circuits Conference, 1984, pp. 518-521.

[LIA83] Y. Z. Liao and C. K. Wong, "An algorithm to compact a VLSI symbolic layout with mixed constraints," 20th Design Automation Conference, 1983, pp. 107-111.

[LOO79] K. J. Loosemore, "Automated layout of integrated circuits," IEEE international Symposium on Circuits and Systems, 1979, pp. 665-668.

[LOS80] P. Losleben, "Computer aided design for VLSI," in Very Large Scale Integration: VLSI, ed. D. F. Barde. Berlin: Springer-Verlag, 1980.

[LSI86] Introduction to Application Specific Integrated Circuits. Canada: LSI Logic Corporation of Canada, Inc., 1986.

[LUD83] J. A. Ludwig, P. Lowy and R. H. McClug, "A hierachical approach to VLSI chip design and verification," International Symposium on Circuits and Systems, vol. 1, 1983, pp. 16-19.

[MAT72] R. L. Mattison, "A high quality, low cost router for MOS/LSI," 9th Design Automation Workshop, 1972, pp. 94-103.

[MEA80] C. Mead and L. Conway, Introduction to VLSI Systems. Reading, MA: Addison-Wesley, 1980.

[MEN84] IDEA System User's Manual. U.S.A.: Mentor Graphics Corporation, 1984.

[MIT84] K. Mitsumoto, H. Mori, T. Fujita and S. Goto, "AI approach to VLSI routing problem," IEEE International Symposium on Circuits and Systems," 1984, pp. 449-452.

[MLY86] D. A. Mlynski and C. H. Sung, "Layout compaction" in Layout Design and Verification, Advances in CAD for VLSI, vol. 4, ed. T. Ohtsuki. New York: North-Holland, 1986. pp. 199-236.

[MOO59] E. F. Moore, "The shortest path through a maze," Annals of the Havard Computation Laboratory, vol. 30, pt. II, 1959, pp. 185-292.

[NAK83] K. Nakajima and M. Sun, "On graph theorotic models for the circuit layout problem," IEEE International Symposium on Circuits and Systems, 1983, vol. 3, pp. 1022-1025.

[OHT86] T. Ohtsuki, "Maze-running and line-search algorithm," in Layout Design and Verification, Advances in CAD for VLSI, vol. 4, ed. T. Ohtsuki. New York: North-Holland, 1986. pp. 99-132.

[PAP82] C. H. Papadimetriou and K. Steiglitz, Combinatorial Optimization: Algorithms and Complexity. Ch. 15. Englewood Cliffs, New York: Prentice Hall, 1982. 496pp.

[POL86] J. Poltz and A. Wexler, "Transmission line analysis of PC boards," VLSI System Design, May 1986, pp. 38-43.

[REE85] J. Reed, A. Sangiovanni-Vincentelli and M. Santomauro, "A new

symbolic channel router: YACR2," IEEE Transactions on Computer-Aided Design, vol. CAD-4, no. 3, July 1985, pp. 208-219.

[RIC80] R. Rice, VLSI: The Coming Revolution in Applications and Design. New York: IEEE Computer Society, 1980.

[RIC84] D. Richards, "Complexity of single-layer routing," IEEE Transaction on Computers, vol. c-33, March 1984, pp. 286-288.

[RIV82] R. L. Rivest and C. M. Fiduccia. "A greedy channel router," 19th Design Automation Conference, 1982, pp. 418-424.

[ROM84] F. Romeo and A. Sangiovanni-Vincentelli, "Probabilistic hill climbing algorithms: Properties and applications," in 1985 Chapel Hill Conference on VLSI, ed. F. Fuchs. Chappel Hill: Computer Science Press, 1985.

[RUB74] F. Rubin, "The Lee path connection algorithm," IFFF Transactions on Computers, vol. C-23, September 1974, pp. 907-914.

[SAN84] A. Sangiovanni-Vincentelli, "A new gridless channel router: Yet Another Channel Router the Second (YACR-II)", IEEE International Conference on Computer-Aided Design, 1984, pp. 72-75.

[SAS86] S. Sastry and A. C. Parker, "Stochastic models for wireability analysis of gate arrays," IEEE Transactions on Computer-Aided Design, vol. CAD-5, no. 1, January 1986.

[SAT83] S. Satry and A. Parker, "The complexity of Two-Dimensional Compaction of VLSI Layout," Proc. IEEE Intrnational Conference on Circuit Theory and Design, 1983, pp. 263-265.

[SCH83a] M. Schlag, Y. Z. Liao and C. K. Wong, "An algorithm for optimal two-dimensional compaction of VLSI layouts," INTEGRATION, VLSI Journal 1, 1983, pp. 179-209.

[SCH83b] W. L. Schiele, "Improved compaction by minimized length of wires," 20th Design Automation Conference, 1983, pp. 121-125.

[SEC84] C. Sechen and A. Sangiovanni-Vancentelli, "The TimberWolf Placement and Routing Package," IEEE Custom Integrated Circuits

Conference, 1984, pp. 522-527.

[SHI86] M. T. Shing and T. C. Hu, "Computational complexity of layout problems," in Layout Design and Verification, Advances in CAD for VLSI, vol. 4, ed. T. Ohtsuki. New York: North-Holland, 1986. pp. 267-294.

[SO73] H. C. So, "Pin assignment of circuit cards and the routability of multilayer printed circuit wiring backplanes," 10th Design Automation Conference, 1973, pp. 33-43.

[SOU78] J. Soukup, "Fast maze router," 15th Design Automation Conference, 1978, pp. 100-102.

[SOU79] J. Soukup, "Global router," 16th Design Automation Conference, 1979, pp. 484-484.

[SOU81] J. Soukup, "Circuit layout," Proceedings of IEEE, vol. 69, pp. 1281-1304, October 1981.

[SZE86] A. A. Szepieniec, "Integrated placement/routing in sliced layouts," 23rd Design Automation Conference, 1986, pp. 300-307.

[SZY82] T. G. Szymanski, "Dogleg channel routing is NP-complete," unpublished manuscript, Bell Laboratories, Murray Hill, 1982.

[TAD80] F. Tada, K. Yoshimura, T. Kagata, and T. Shirakawa, "A fast maze router with iterative use of variable search space restriction," 17th Design Automation Conference, 1980, pp. 250-254.

[TAY84] S. Taylor, "Symbolic layout," VLSI Design Journal, March 1984, pp. 34-42.

[TER85] H. Terai, M. Hayase and T. Kozawa, "A routing procedure for mixed array of custom macros and standard cells," 22nd Design Automation Conference, 1985, pp. 503-508.

[TSK87a] C. L. Tse and W. Kinsner, "A graph based heuristic channel router." Submitted for publication, August 1987.

[TSK87b] C. L. Tse and W. Kinsner, "A graph based heuristic channel router

with doglegs." Submitted for publication, August 1987.

[UED86] K. Ueda, R. Kasai and T. Sudo, "Layout strategy, standardization, and CAD tools," in Layout Design and Verification, Advances in CAD for VLSI, vol. 4, ed. T. Ohtsuki. New York: North-Holland, 1986. pp. 1-54.

[WEX85] A. Wexler, "Getting a handle on impedance, cross-talk, time delay, and ringing," Printed Circuit Design, December 1985, pp. 14-17.

[YOS82] T. Yoshimura and E. S. Kuh, "Efficient algorithms for channel routing," IEEE Transactions on CAD of Integrated Circuits and Systems, V. CAD-1, 1, 1982, pp. 25-35.

[YOS84] T. Yoshimura, "An efficient channel router," 21st Design Automation Conference, 1984, pp. 38-44.

[YOS86] K. Yoshida, "Layout verification" in Layout Design and Verification, Advances in CAD for VLSI, vol. 4, ed. T. Ohtsuki. New York: North-Holland, 1986. pp. 237-267.

APPENDIX A

DOGLEG ROUTER PROGRAM STRUCTURE

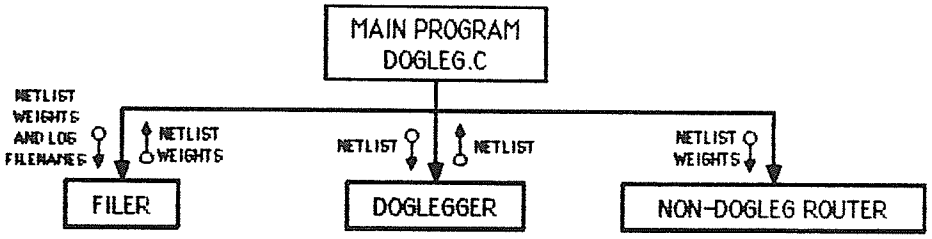


Fig. A1. Structure of the main program.

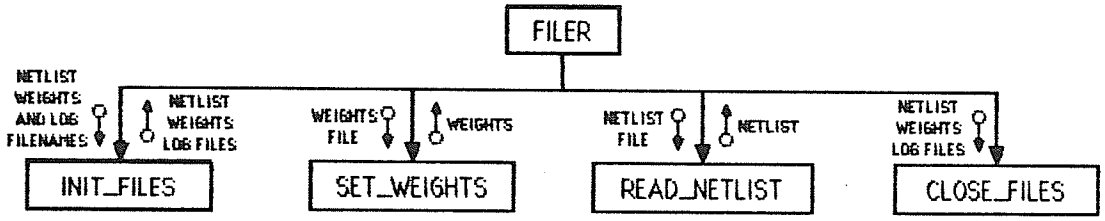


Fig. A2. Structure of the Filer section.

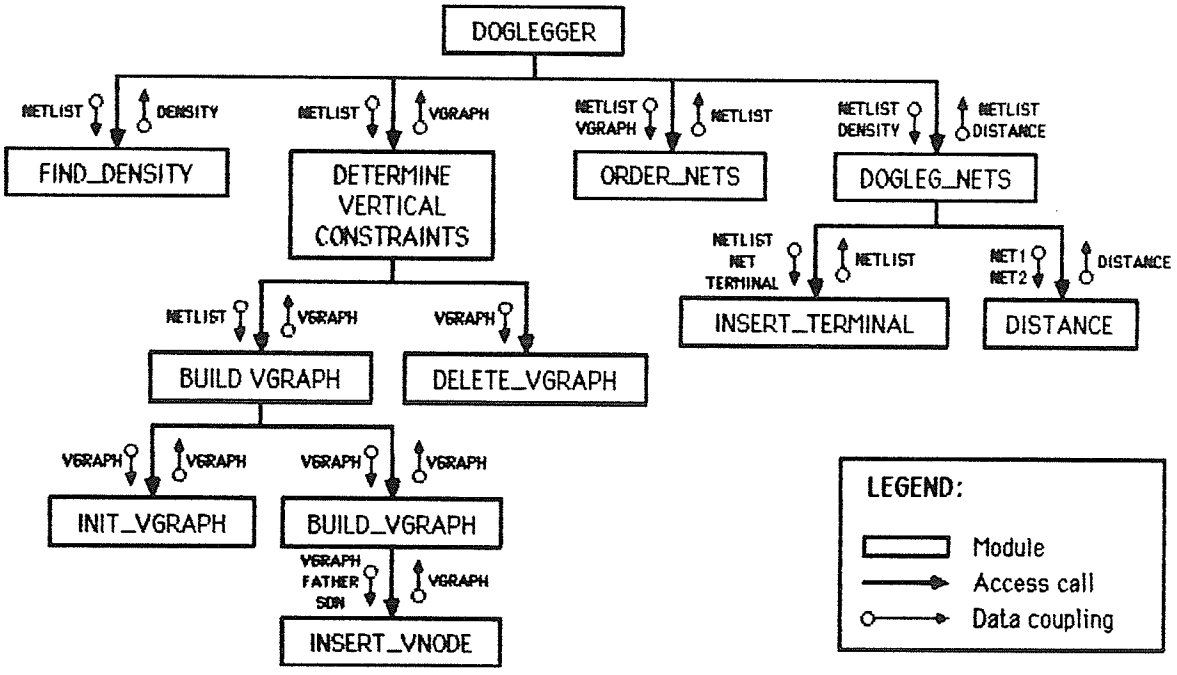


Fig. A3. Structure of the Doglegger section.

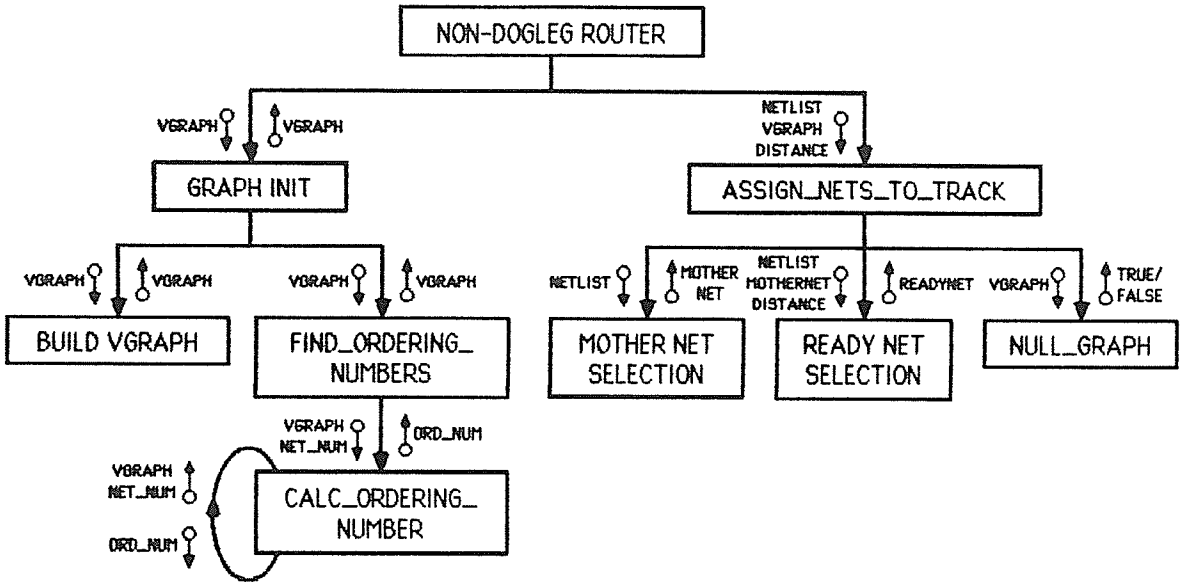


Fig. A4. Structure of the Non-Dogleg router section.

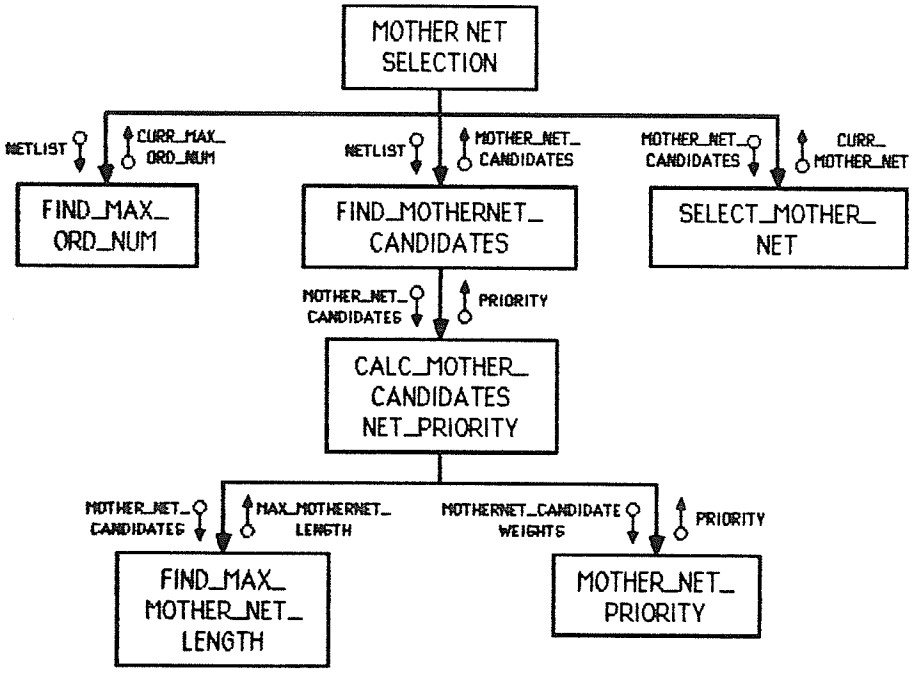


Fig. A4. Structure of the Mother net selection process.

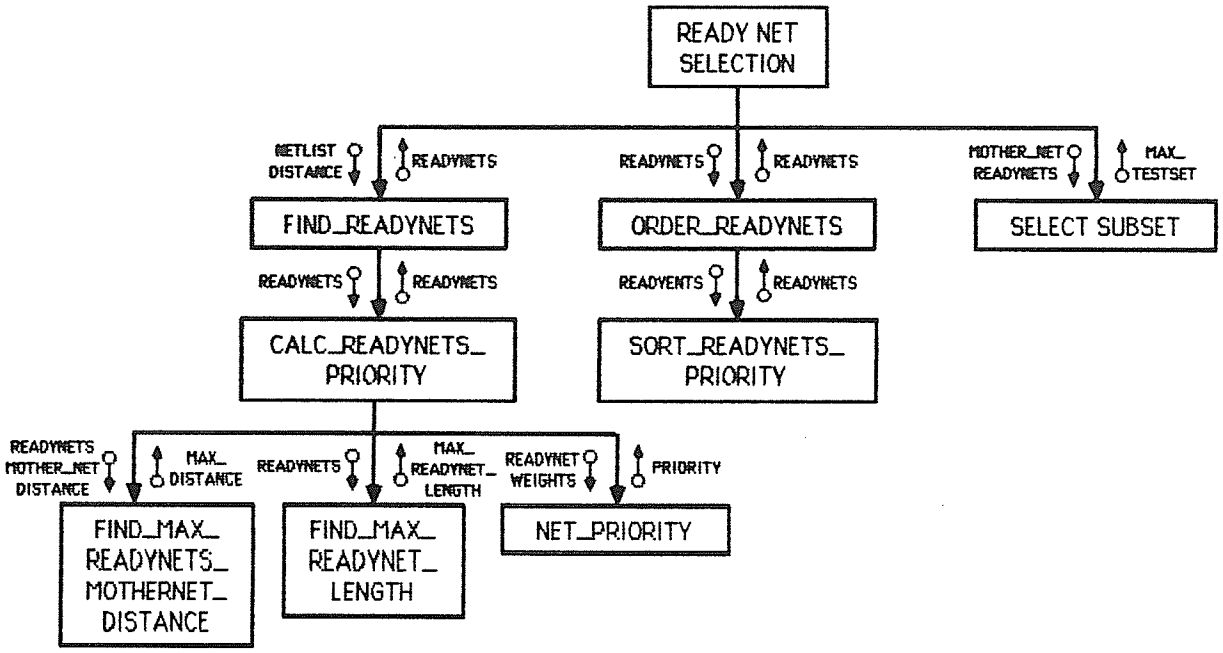


Fig. A6. Structure of the Ready net selection process.

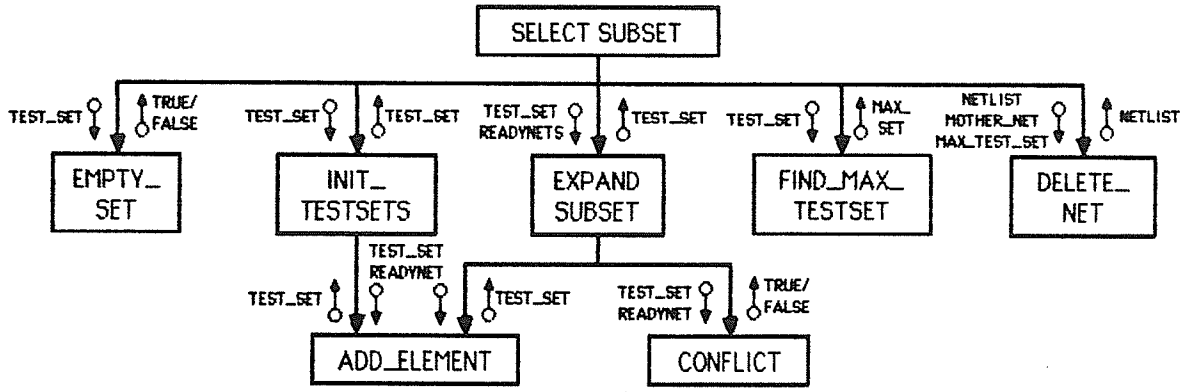


Fig. A7. Structure of the subset selection process.

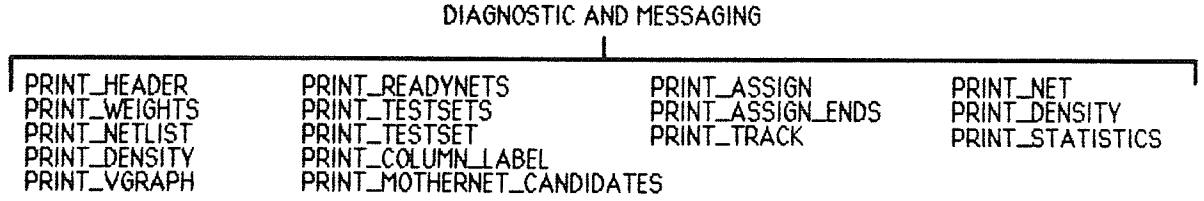


Fig. A8. List of diagnostic and messaging procedures.

APPENDIX B

DOGLEG CHANNEL ROUTER PROGRAM LISTING

```
/* Dogleg Detailed Channel Router Header File: dogleg.h */

# define   MAXNET              200
# define   MAXTERM             200
# define   MAXHEIGHT           35
# define   MAXVIA              10
# define   NUM_TEST_SET        5

# define   EONET               999
# define   NAMELENGTH          32
# define   MAXSETSIZE          50

# define   ABS(x)              (((x)<0)?-(x):(x))
# define   SIGN(x)             (((x)<0)?(-1):(1))
# define   MAX(a,b)            (((a)<(b))?(b):(a))
# define   MIN(a,b)            (((a)<(b))?(a):(b))

# define   LEFT(a)             (netlist[(a)].left)
# define   RIGHT(a)            (netlist[(a)].right)
# define   LENGTH(a)           (RIGHT(a)-LEFT(a))
# define   ORD(a)              (netlist[(a)].ord_num)
# define   PARENT(a)           (netlist[(a)].parent)


typedef struct terminal_struct
   {
   int                         number;
   struct terminal_struct      *next;
   } Terminal;

typedef struct vnode_struct
   {
   struct vnode_struct         *next;
   int                         net_num;
   } Vnode;

typedef struct net_struct
   {
   Terminal                    *first_term, *last_term;
   int                         left, right;
   Vnode                       *first_son;
   int                         ord_num;
   int                         parent;
   } Net;

typedef struct candidate_struct
   {
```

```
    int                         net;
    int                         priority;
    int                         degree;
    } Candidate;

typedef struct testset_struct
    {
    int                         element[ MAXSETSIZE ];
    int                         size;
    int                         priority;
    } Testset;



Net                             netlist[ MAXNET ];
int                             assignment[ MAXHEIGHT ];

Candidate                       mothernet_candidates[ MAXNET ];
Candidate                       readynets[ MAXNET ];
Testset                         testset[ NUM_TEST_SET ];

int                             density[ MAXTERM ];
int                             hgraph[ MAXNET ][MAXNET ];

int                             total_num_term = 0;
int                             initial_num_net;

int                             total_num_net = 0;
int                             total_num_dogleg = 0;
int                             total_num_mothernet_candidates;
int                             total_num_readynets;

int                             curr_max_ord_num;
int                             curr_mother_net;
int                             max_mothernet_length;
int                             max_readynet_length;
int                             ord_den_ratio;

/* channel statistics */
int                             max_density = 0;
int                             max_ord_num = 0;
int                             max_distance;
int                             max_term = 0;

/* weighting factors */
int                             mother_weight;
int                             ordering_weight;
int                             length_weight;
int                             distance_weight;
int                             subnet_distance;

/* global file names and file pointers */
char                            *netlist_filename;
char                            *log_filename;
char                            *weight_filename;
char                            temp_filename1[] = "_temp_file1";
char                            temp_filename2[] = "_temp_file2";
```

```
FILE            *netlist_file;
FILE            *log_file;
FILE            *weight_file;
FILE            *temp_file1;
FILE            *temp_file2;
```

```
/*****************************************************************/
/*                                                               */
/*   Graph Based Heuristic Dogleg Detailed Channel Router.       */
/*                                                               */
/*                        by Cheung-Lai Tse                      */
/*                                                               */
/*****************************************************************/



# include <stdio.h>
# include "dogleg.h"

main( argc,argv )
   int                          argc;
   char                         *argv[];
{
   filer( argc,argv );
   init_hgraph();
   build_hgraph();
   doglegger();          /* remove for non-dogleg routing */
   non_dogleg_router();
   close_files();
}



/*********/
/* FILER */
/*********/

filer( argc, argv )
   int                           argc;
   char                          *argv[];
{
   init_files( argc,argv );
   print_version();

   set_weights();
   print_weights();

   read_netlist();
   print_netlist( "Input " );
}



init_files( argc,argv )
   int                          argc;
   char                         *argv[];
{
   if ( argc < 4 )
      {
      printf( "Wrong number of arguments.\n" );
      printf( "Usage: %s netlist weights logfile\n",argv[0] );
      exit();
      }
```

```
    netlist_filename = argv[1];
    if ( (netlist_file = fopen( netlist_filename,"r" )) == NULL )
       {
       printf( "\nError: cannot open netlist file %s.\n",netlist_filename );
       exit();
       }

    weight_filename = argv[2];
    if ( (weight_file = fopen( weight_filename,"r" )) == NULL )
       {
       printf( "\nCannot open weighting factors file %s.\n",weight_filename );
       exit();
       }

    log_filename = argv[3];
    if ( (log_file = fopen( log_filename,"w" )) == NULL )
       {
       printf( "\nError: cannot open log file %s.\n",log_filename );
       exit();
       }

    temp_file1 = fopen( temp_filename1,"w" );
    temp_file2 = fopen( temp_filename2,"w" );
}



set_weights()
{
    fscanf( weight_file,"%*s%d %*s%d %*s%d %*s%d %*s%d",&mother_weight,
       &ordering_weight,&length_weight,&distance_weight,&subnet_distance );
}



read_netlist()
{
    int                         c, left, right, term, vtrack;
    Terminal                    *terminal;
    Net                         *net;

    /* skip over comments in the net list file delimited by $ */
    while( (c=getc(netlist_file)) != EOF && c!='$' )
       putc( c,log_file );

    /* read number of terminals */
    fscanf( netlist_file,"%d",&total_num_term );

    total_num_net = 0;
    net = netlist;
    while ( fscanf( netlist_file,"%d",&term ) != EOF )
       {
       left = term;
       total_num_net ++;
       net ++;
       net->first_term = net->last_term = NULL;
       net->prev = net->next = 0;
```

```
      do
         {
         terminal = (Terminal *)malloc( sizeof(Terminal) );
         terminal->number = term;
         insert_terminal( net,terminal );
         right = term;
         fscanf( netlist_file,"%d",&term );
         } while ( term != EONET );
      net->left = ABS( left );
      net->right = ABS( right );
      }
   initial_num_net = total_num_net;
}



close_files()
{
   int                              c;

   temp_file1 = freopen( temp_filename1,"r",temp_file1 );
   while( (c=getc(temp_file1)) != EOF )
      putc( c,log_file );
   fclose( temp_file1 );
   unlink( temp_filename1 );

   fprintf( log_file,"\n\n" );
   temp_file2 = freopen( temp_filename2,"r",temp_file2 );
   while( (c=getc(temp_file2)) != EOF )
      putc( c,log_file );
   fclose( temp_file2 );
   unlink( temp_filename2 );

   fprintf( log_file,"\n\n" );
   fclose( weight_file );
   fclose( netlist_file );
   fclose( log_file );
}



/*************/
/* DOGLEGGER */
/*************/

doglegger()
{
   find_density();

   init_vgraph();
   build_vgraph();
   print_vgraph( "Original " );
   order_nets();
   print_vgraph( "Ordered " );
   delete_vgraph();

   dogleg_nets();
```

```
   find_density();
   print_netlist( "Doglegged " );
   print_density();
}



dogleg_nets()
{
   Terminal                    *terminal,*new_terminal,
                               *prev_terminal, *first_terminal;
   Net                         *last_net, *net;
   int                         i, j, first_subnet, net_num;

   init_hgraph();
   last_net = netlist + total_num_net;
   for ( net=last_net,net_num=total_num_net; net!=netlist; net--,net_num-- )
      {
      if ( PARENT(net_num) == 0 || net->first_son == NULL )
         continue;
      first_subnet = total_num_net + 1;
      prev_terminal = net->first_term;
      terminal = prev_terminal->next;
      while ( terminal != net->last_term )
         {
         if ( density[ ABS(terminal->number) ] < max_density &&
              (prev_terminal->number ^ terminal->next->number) < 0 )
            {
            new_terminal = (Terminal *)malloc( sizeof(Terminal) );
            new_terminal->number = terminal->number;
            first_terminal = net->first_term;
            net->first_term = terminal;
            total_num_net ++;
            last_net ++;
            last_net->first_term = last_net->last_term = NULL;
            last_net->prev = last_net->next = 0;
            insert_terminal( last_net,first_terminal );
            insert_terminal( last_net,new_terminal );
            density[ ABS(terminal->number) ] ++;
            for ( i=1; i<=total_num_net; i++ )
               hgraph[total_num_net][i] = hgraph[i][total_num_net] = 0;
            }
         prev_terminal = terminal;
         terminal = terminal->next;
         }
      for ( i=first_subnet; i<total_num_net; i++ )
         hgraph[i][i+1] = hgraph[i+1][i] = subnet_distance;
      if ( first_subnet != total_num_net+1 )
         hgraph[total_num_net][net_num] = hgraph[net_num][total_num_net]
            = subnet_distance;
      }

   /* calculate left and right ends of the nets */
   for ( net=netlist+total_num_net; net!=netlist; net-- )
      {
      net->left = ABS( net->first_term->number );
      net->right = ABS( net->last_term->number );
      }
```

```
   /* calculate number of potential doglegs */
   total_num_dogleg = total_num_net - initial_num_net;

   build_hgraph();
}



/*********************************************/
/* HORIZONTAL CONSTRIANT GRAPH CONSTRUCTION */
/*********************************************/

init_hgraph()
{
   int                          i, j;

   for ( i=1; i<=MAXNET; i++ )
      for ( j=i; j<=MAXNET; j++ )
         hgraph[i][j] = hgraph[j][i] = 0;
}



build_hgraph()
{
   int                          i, j;

   for ( i=1; i<=total_num_net; i++ )
      for ( j=i; j<=total_num_net; j++ )
         if ( hgraph[i][j] == 0 )
            hgraph[i][j] = hgraph[j][i] = distance( i,j );
}



/*********************************************/
/* VERTICAL CONSTRAINT GRAPH CONSTRUCTION */
/*********************************************/

init_vgraph()
{
   int                          index;
   Net                          *net;

   for ( index=0,net=netlist; index<MAXNET; index++,net++ )
      {
      net->first_son = NULL;
      net->parent = net->ord_num = 0;
      }
}
```

```
build_vgraph()
{
   int                          terminals[4][MAXTERM];
   int                          net_num, term_num, sub, vtrack, *old_net;
   int                          i, j;
   Net                          *net;
   Terminal                     *terminal;

   for ( sub=0; sub<4; sub++ )
      for ( term_num=0; term_num<=total_num_term; term_num++ )
         terminals[sub][term_num] = 0;

   for ( net_num=1,net=netlist+1; net_num<=total_num_net; net_num++,net++ )
      {
      for ( terminal=net->first_term; terminal!=NULL; terminal=terminal->next )
         {
         term_num = ABS( terminal->number );
         if ( term_num >= 1 && term_num <= total_num_term )
            for ( sub=0; sub<4; sub++ )
               {
               old_net = &(terminals[sub][term_num]);
               if ( *old_net == 0 )
                  {
                  *old_net = (terminal->number<0)? -net_num:net_num;
                  break;
                  }
               else if ( *old_net > 0 && terminal->number < 0 )
                  insert_vnode( *old_net,net_num );
               else if ( *old_net < 0 && terminal->number > 0 )
                  insert_vnode( net_num,-(*old_net) );
               }
         }
      }
}



insert_vnode( father,son )
   int                          father, son;
{
   Vnode                        *new_node;
   Net                          *father_net;

   father_net = netlist + father;
   new_node = (Vnode *)malloc( sizeof(Vnode) );
   new_node->net_num = son;
   if ( father_net->first_son == NULL )
      {
      father_net->first_son = new_node;
      new_node->next = NULL;
      }
   else
      {
      new_node->next = father_net->first_son;
      father_net->first_son = new_node;
      }
   ((netlist+son)->parent) ++;
}
```

```
/**********************/
/* NON-DOGLEG ROUTER */
/**********************/

non_dogleg_router()
{
   init_vgraph();
   build_vgraph();
   find_ordering_numbers();
   print_vgraph( "Doglegged " );

   assign_nets_to_track();
   print_statistics();
}



find_ordering_numbers()
{
   int                              net_num;

   /* find the ordering numbers */
   max_ord_num = 0;
   for ( net_num=1; net_num<=total_num_net; net_num++ )
      {
      calc_ordering_number( net_num );
      if ( ORD(net_num) > max_ord_num )
         max_ord_num = ORD(net_num);
      }
}



calc_ordering_number( net_num )
   int                              net_num;
{
   Net                              *father, *son;
   Vnode                            *node;

   father =  netlist + net_num;
   if ( father->ord_num == 0 )
      {
      if ( father->first_son == NULL )
         father->ord_num = 1;
      else
         for ( node=father->first_son; node!=NULL; node=node->next )
            {
            calc_ordering_number( node->net_num );
            son = netlist + node->net_num;
            if ( son->ord_num >= father->ord_num )
               father->ord_num = son->ord_num + 1;
            }
      }
}
```

```
assign_nets_to_track()
{
   int                                  track_num;

   ord_den_ratio = (mother_weight * max_ord_num ) / max_density;

   print_column_label( temp_file1 );
   print_column_label( temp_file2 );

   track_num = 0;
   while ( !null_graph() )
      {
      track_num ++;
      fprintf( log_file,"\nTrack %d\n",track_num );

      find_curr_max_ord_num();

      find_mothernet_candidates();
      print_mothernet_candidates();
      select_mother_net();

      find_readynets();
      order_readynets();
      print_readynets();
      select_subset();
      }
   fprintf( log_file,"\nTotal number of track used: %d\n\n",track_num );
   printf( "    tracks: %2d\n",track_num );
}



/************************/
/* MOTHER NET SELECTION */
/************************/

find_mothernet_candidates()
{
   int                                  net;

   total_num_mothernet_candidates = 0;
   for ( net=1; net<=total_num_net; net++ )
      if ( ORD(net) != 0 && PARENT(net) == 0 )
         mothernet_candidates[ total_num_mothernet_candidates++ ].net = net;
   calc_mothernet_candidates_priority();
}



calc_mothernet_candidates_priority()
{
   int                                  i;

   find_max_mothernet_candidate_length();
   fprintf( log_file,"Max mothernet length:%d   Max ordering:%d   Ord/Den:%d\n",
      max_mothernet_length,curr_max_ord_num,ord_den_ratio );
   for ( i=0; i<total_num_mothernet_candidates; i++ )
      mothernet_candidates[i].priority
```

```
        = mother_net_priority( mothernet_candidates[i].net );
}



find_max_mothernet_candidate_length()
{
   int                              i;

   max_mothernet_length = 0;
   for ( i=0; i<total_num_mothernet_candidates; i++ )
      if ( LENGTH(mothernet_candidates[i].net) > max_mothernet_length )
         max_mothernet_length = LENGTH(mothernet_candidates[i].net);
}



int mother_net_priority( mothernet )
   int                              mothernet;
{
   int                              priority;

   priority = (10*LENGTH(mothernet))/max_mothernet_length +
(ord_den_ratio*ORD(mothernet))/curr_max_ord_num;
   fprintf( log_file,"Mother net %-3d: length:%-3d  ord:%-2d  priority:%-3d\n",
      mothernet,LENGTH(mothernet),ORD(mothernet),priority );
   return priority;
}



int select_mother_net()
{
   int                              i;
   int                              max_mothernet_priority;

   max_mothernet_priority = -99999;
   for ( i=0; i<total_num_mothernet_candidates; i++ )
      if ( mothernet_candidates[i].priority > max_mothernet_priority )
         {
         max_mothernet_priority = mothernet_candidates[i].priority;
         curr_mother_net = mothernet_candidates[i].net;
         }
   fprintf( log_file,"Mother net selected: %d\n\n",curr_mother_net );
}
```

```
/*********************************************************/
/* READYNETS SELECTION AND MAXIMUM SUBSET SELECTION */
/*********************************************************/

find_readynets()
{
   int                                  net, i;

   total_num_readynets = 0;
   for ( net=1; net<=total_num_net; net++ )
      if ( ORD(net)!=0 && PARENT(net)==0 && hgraph[net][curr_mother_net]!=0 )
         readynets[ total_num_readynets++ ].net = net;
   calc_readynets_priority();
}



calc_readynets_priority()
{
   int                                   i;

   find_max_readynets_mothernet_distance();
   find_max_readynet_length();
   fprintf( log_file,"Max ordering:%d  Max readynet length:%d  Max dist:%d\n",
      curr_max_ord_num,max_readynet_length,max_distance );
   for ( i=0; i<total_num_readynets; i++ )
      readynets[ i ].priority = net_priority( readynets[i].net );
}



find_curr_max_ord_num()
{
   int                                  net;

   curr_max_ord_num = 0;
   for ( net=1; net<=total_num_net; net++ )
      if ( ORD(net) > curr_max_ord_num )
         curr_max_ord_num = ORD( net );
}



find_max_readynets_mothernet_distance()
{
   int                                  i;

   max_distance = -99999;
   for ( i=0; i<total_num_readynets; i++ )
      if ( hgraph[curr_mother_net][readynets[i].net] > max_distance )
         max_distance = hgraph[curr_mother_net][readynets[i].net];
}



find_max_readynet_length()
{
```

```
   int                                  i;

   max_readynet_length = 0;
   for ( i=0; i<total_num_readynets; i++ )
      if ( LENGTH(readynets[i].net) > max_readynet_length )
         max_readynet_length = LENGTH(readynets[i].net);
}



select_subset()
{
   int                                  max_set, max_set_priority;
   int                                  i, j;

   /* clear all subsets to null */
   for ( i=0; i<NUM_TEST_SET; i++ )
      empty_set( testset+i );

   init_testsets();

   /* expand subsets */
   for ( i=0; i<total_num_readynets; i++ )
      for ( j=0; j<NUM_TEST_SET; j++ )
         if ( !conflict( testset+j,readynets[i].net ) )
            add_element( testset+j,i );
   fprintf( log_file,"Expanded subsets:\n" );
   print_testsets();

   /* find maximum test set */
   max_set = 0;
   max_set_priority = testset[0].priority;
   for ( j=1; j<NUM_TEST_SET; j++ )
      if ( testset[j].priority > max_set_priority )
         {
         max_set = j;
         max_set_priority = testset[j].priority;
         }
   fprintf( log_file,"Subset selected: %d\n",max_set );

   delete_net( curr_mother_net );
   for ( i=0; i<testset[max_set].size; i++ )
      delete_net( testset[max_set].element[i] );

   print_subset( max_set );
}



empty_set( set )
   Testset                              *set;
{
   set->size = set->priority = 0;
}
```

```
init_testsets()
{
   int                                  i;

   /* init subsets */
   for ( i=0; i<total_num_readynets && i<NUM_TEST_SET; i++ )
      add_element( testset+i,i );
}



order_readynets()
{
   int                          j, count;
   Candidate                    next;

   for ( j=1; j<total_num_readynets; j++ )
      {
      next = readynets[ j ];
      count = j - 1;
      while ( count >= 0 )
         if ( next.priority > readynets[count].priority )
            {
            readynets[count+1] = readynets[count];
            count --;
            }
         else
            break;
      readynets[count+1] = next;
      }
}



int net_priority( net )
   int                                  net;
{
   int                                  priority;

   priority = ( ordering_weight * ORD(net) ) / curr_max_ord_num +
              ( length_weight * LENGTH(net) ) / max_readynet_length +
              ( distance_weight * (max_distance-hgraph[curr_mother_net][net]) )
              / max_distance;
   fprintf( log_file,"Net %-3d:    ord: %-2d    length: %-3d    ",
            "dist: %-2d    priority: %-3d\n",net,ORD(net),LENGTH(net),
           hgraph[curr_mother_net][net],priority );
   return priority;
}



/*****************************/
/* MISC SUPPORTING ROUTINES */
/*****************************/

add_element( set,i )
   Testset                              *set;
   int                                  i;
```

```
{
   int                                  element_priority;

   set->element[ (set->size)++ ] = readynets[i].net;
   set->priority += readynets[i].priority;
}



int conflict( set,net )
   Testset                              *set;
   int                                  net;
{
   int                                  i;

   for ( i=0; i<set->size; i++ )
      if ( hgraph[set->element[i]][net] == 0 )
         return 1;
   return 0;
}



find_distance()
{
   int                          i, j;

   /* calculate distance matrix */
   for ( i=1; i<=total_num_net; i++ )
      for ( j=i; j<=total_num_net; j++ )
         hgraph[i][j] = hgraph[j][i] = distance( i,j );
}



find_density()
{
   int                          net, vtrack;

   for ( vtrack=0; vtrack<MAXTERM; vtrack++ )
      density[vtrack] = 0;

   max_density = 0;
   for ( net=1; net<=total_num_net; net++ )
      for (vtrack=LEFT(net); vtrack<=RIGHT(net); vtrack++ )
         if ( (++density[vtrack]) > max_density )
            max_density = density[vtrack];
}



insert_terminal( net,terminal )
   Net                          *net;
   Terminal                     *terminal;
{
   if ( net->first_term == NULL )
      net->first_term = net->last_term = terminal;
```

```
   else
      {
      net->last_term->next = terminal;
      net->last_term = terminal;
      }
   terminal->next = NULL;
}



delete_net( net )
   int                                  net;
{
   Vnode                                *prev, *node;

   ORD(net) = 0;
   node = netlist[net].first_son;
   while (  node != NULL )
       {
       netlist[node->net_num].parent --;
       prev = node;
       node = node->next;
       free( (char *)prev );
       }
}



delete_vgraph()
{
   int                          net_num;
   Net                          *net;
   Vnode                        *vnode, *prev;

   for ( net_num=1,net=netlist+1; net_num<=total_num_net; net_num++,net++ )
      {
      vnode = net->first_son;
      while ( vnode != NULL )
         {
         prev = vnode;
         vnode = vnode->next;
         free( (char *)prev );
         }
      }
}



order_nets()
{
   int                              j, count;
   Net                              next_net;

   for ( j=1; j<=total_num_net; j++ )
      {
      next_net = netlist[j];
      count = j - 1;
      while ( count >= 0 )
```

```
        if ( next_net.parent < PARENT(count) )
           {
           netlist[count+1] = netlist[count];
           count --;
           }
        else
           break;
     netlist[count+1] = next_net;
     }
}



int distance( net1,net2 )
   int                                net1, net2;
{
   int                                diff1, diff2;

   diff1 = RIGHT(net1) - LEFT(net2);
   diff2 = LEFT(net1) - RIGHT(net2);
   if ( diff1>0 && diff2<0 )
      return 0;
   else
      {
      diff1 = ABS(diff1);
      diff2 = ABS(diff2);
      return MIN( diff1,diff2 );
      }
}



null_graph()
{
   int                              net;

   for ( net=1; net<=total_num_net; net++ )
      if ( ORD(net) != 0 )
         return 0;
   return 1;
}



abort()
{
   printf( "\nProcess terminated.\n\n" );
   fprintf( log_file,"\nProcess terminated.\n\n" );
   exit();
}



sort_left( max_set )
   int                           max_set;
{
   int                           j, count;
```

```
   int                          next;

   for ( j=1; j<testset[max_set].size; j++ )
      {
      next = testset[max_set].element[j];
      count = j - 1;
      while ( count >= 0 )
         if ( LEFT(next) < LEFT(testset[max_set].element[count]) )
            {
            testset[max_set].element[count+1] = testset[max_set].element[count];
            count --;
            }
         else
            break;
      testset[max_set].element[count+1] = next;
      }
}



print_track( max_set )
   int                          max_set;
{
   int                          index, posn;

   fprintf( log_file,"     |" );
   posn = 1;
   for ( index=0; index<testset[max_set].size; index++ )
      {
      for ( ; posn<LEFT(testset[max_set].element[index]); posn++ )
         putc( ' ',log_file );
      putc( '+',log_file );
      posn ++;
      for ( ; posn<RIGHT(testset[max_set].element[index]); posn++ )
         putc( '-',log_file );
      putc( '+',log_file );
      posn ++;
      }
   putc( '\n',log_file );

   fprintf( log_file,"      " );
   posn = 1;
   for ( index=0; index<testset[max_set].size; index++ )
      {
      for ( ; posn<LEFT(testset[max_set].element[index]); posn++ )
         putc( ' ',log_file );
      fprintf( log_file,"%-3d",testset[max_set].element[index] );
      posn += 3;
      for ( ; posn<=RIGHT(testset[max_set].element[index]); posn++ )
         putc( ' ',log_file );
      }
   fprintf( log_file,"\n\n\n" );
}



print_column_label( file )
   FILE                         *file;
```

```
{
   int                              i;

   fprintf( file,"     " );
   for ( i=1; i<=total_num_term; i++ )
      fprintf( file,"%1d",i%10 );
   putc( '\n',file );
}



print_assign( max_set )
   int                              max_set;
{
   static int                       track = 0;
   Terminal                         *terminal;
   int                              index, connect, term, posn;

   fprintf( temp_file1,"%-5d",++track );
   for ( posn=1,index=0; index<testset[max_set].size; index++ )
      {
      terminal = netlist[testset[max_set].element[index]].first_term;
      for ( ; posn<ABS(terminal->number); posn++ )
         putc( ' ',temp_file1 );
      for ( ; terminal!=NULL; terminal=terminal->next )
         {
         connect = terminal->number;
         term = ABS( connect );
         for ( ; posn<term; posn++ )
            putc( '-',temp_file1 );
         if ( term == 0 )
            putc( '<',temp_file1 );
         else if ( term > total_num_term )
            putc( '>',temp_file1 );
         else if ( posn == term )
            putc( (connect<0)? 'v':'^',temp_file1 );
         else
            {
            total_num_dogleg --;
            continue;
            }
         posn ++;
         }
      }
   for ( ; posn<total_num_term+10; posn++ )
      putc( ' ',temp_file1 );
   for ( index=0; index<testset[max_set].size; index++ )
      fprintf( temp_file1,"%-4d",testset[max_set].element[index] );
   putc( '\n',temp_file1 );
}



print_assign_ends( max_set )
   int                              max_set;
{
   static int                       track = 0;
   int                              index, net_num, posn;
```

```
   posn = 1;
   fprintf( temp_file2,"%-5d",++track );
   for ( index=0; index<testset[max_set].size; index++ )
      {
      net_num = testset[max_set].element[index];
      for ( ; posn<LEFT(net_num); posn++ )
         putc( ' ',temp_file2 );
      if ( posn == LEFT(net_num) )
         {
         putc( '+',temp_file2 );
         posn ++;
         }
      for ( ; posn<RIGHT(net_num); posn++ )
         putc( '-',temp_file2 );
      if ( posn <= total_num_term )
         {
         putc( '+',temp_file2 );
         posn ++;
         }
      }
   for ( ; posn<total_num_term+10; posn++ )
      putc( ' ',temp_file2 );
   for ( index=0; index<testset[max_set].size; index++ )
      fprintf( temp_file2,"%-4d",testset[max_set].element[index] );
   putc( '\n',temp_file2 );
}



print_version()
{
   fprintf( log_file,"Dogleg Channel Router\n\n" );
   fprintf( log_file,"Netlist: %s\nWeighting factors: %s\nLog: %s\n\n",
      netlist_filename,weight_filename,log_filename );
}



print_weights()
{
   fprintf( log_file,"Weights: mother: %-2d   ordering: %-2d   length: %-2d   ",
      distance: %-2d   subnet: %-2d\n\n",mother_weight,ordering_weight,
      length_weight,distance_weight,subnet_distance );
   printf( "%-10s mother: %-2d   ordering: %-2d   length: %-2d   ",
      "distance: %-2d   subnet: %-2d",netlist_filename,mother_weight,
      ordering_weight,length_weight,distance_weight,subnet_distance );
}



print_statistics()
{
   fprintf( log_file,"Original number of nets: %d\n",initial_num_net );
   fprintf( log_file,"Total number of subnets: %d\n\n",total_num_net );
   fprintf( log_file,"Potential number of doglegs: %d\n",
      total_num_net-initial_num_net );
   fprintf( log_file,"Actual number of doglegs: %d\n\n",total_num_dogleg );
```

```
   fprintf( log_file,"Number of terminals: %d\n",total_num_term );
   fprintf( log_file,"Maximum ordering number: %d\n",max_ord_num );
   fprintf( log_file,"Maximum density number: %d\n\n",max_density );
}



print_vgraph( head )
   char                          *head;
{
   int                           net_num, term;
   Net                           *net;
   Vnode                         *node;

   fprintf( log_file,"%sVertical Constraint Graph:\n",head );
   for ( net_num=1,net=netlist+1; net_num<=total_num_net; net_num++,net++ )
      {
      fprintf( log_file,"Net %3d: ordering(%-2d) parent(%-2d)",
         net_num,net->ord_num,net->parent );
      for ( node=net->first_son; node!=NULL; node=node->next )
         fprintf( log_file,"\t%-3d",node->net_num );
      putc( '\n',log_file );
      }
   putc( '\n',log_file );
}



print_mothernet_candidates()
{
   int                           i;

   fprintf( log_file,"Mother Net Candidates:\n" );
   for ( i=0; i<total_num_mothernet_candidates; i++ )
      print_net( mothernet_candidates[i].net );
}



print_readynets()
{
   int                           i;

   fprintf( log_file,"Readynets:\n" );
   for ( i=0; i<total_num_readynets; i++ )
      print_net( readynets[i].net );
}



print_net( net )
   int                           net;
{
   int                           posn;

   fprintf( log_file,"%-3d  |",net );
   for ( posn=1; posn<LEFT(net); posn++ )
```

```
      putc( ' ',log_file );
   putc( '+',log_file );
   posn ++;
   for ( ; posn<RIGHT(net); posn++ )
      putc( '-',log_file );
   fprintf( log_file,"+\n" );
}



print_subset( max_set )
   int                              max_set;
{
   testset[max_set].element[ (testset[max_set].size)++ ] = curr_mother_net;
   sort_left( max_set );
   print_track( max_set );
   print_assign( max_set );
   print_assign_ends( max_set );
}



print_testsets()
{
   int                              i;

   for ( i=0; i<NUM_TEST_SET; i++ )
      print_testset( i );
}



print_testset( set )
   int                              set;
{
   int                              i;

   fprintf( log_file,"Set %-2d: (%-3d):",set,testset[set].priority );
   for ( i=0; i<testset[set].size; i++ )
      fprintf( log_file,"\t%-3d",testset[set].element[i] );
   putc( '\n',log_file );
}



print_netlist( head )
   char                             *head;
{
   int                              net_num;
   Terminal                         *terminal;

   fprintf( log_file,"%sNetlist:\n",head );
   for ( net_num=1; net_num<=total_num_net; net_num++ )
      {
      fprintf( log_file,"Net %3d:",net_num );
      for ( terminal=netlist[net_num].first_term; terminal != NULL;
         terminal= terminal->next )
```

```
            fprintf( log_file,"%5d",terminal->number );
         putc( '\n',log_file );
         }
      fprintf( log_file,"\n\n" );
}



print_density()
{
   int                                 term;

   fprintf( log_file,"Terminal density numbers:\n" );
   for ( term=1; term<=total_num_term; term++ )
      {
      fprintf( log_file,"%5d(%2d)",term,density[term] );
      if ( (term % 10) == 0 )
         putc( '\n',log_file );
      }
   fprintf( log_file,"\n\n\n" );
}
```