

Student Work

12-1-1999

A Genetic-Based Approach to Multi-layer Channel Routing In VLSI Design.

Mark P. Cloyed

Follow this and additional works at: <https://digitalcommons.unomaha.edu/studentwork>

Recommended Citation

Cloyed, Mark P., "A Genetic-Based Approach to Multi-layer Channel Routing In VLSI Design." (1999).
Student Work. 3586.

<https://digitalcommons.unomaha.edu/studentwork/3586>

This Thesis is brought to you for free and open access by DigitalCommons@UNO. It has been accepted for inclusion in Student Work by an authorized administrator of DigitalCommons@UNO. For more information, please contact unodigitalcommons@unomaha.edu.



A Genetic-Based Approach to Multi-layer Channel Routing
In VLSI Design

A Thesis

Presented to the

Department of Computer Science

And the

Faculty of the Graduate College

University of Nebraska at Omaha

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

University of Nebraska at Omaha

By

Mark P. Cloyed

December 1999

UMI Number: EP74785

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



UMI EP74785

Published by ProQuest LLC (2015). Copyright in the Dissertation held by the Author.

Microform Edition © ProQuest LLC.

All rights reserved. This work is protected against unauthorized copying under Title 17, United States Code



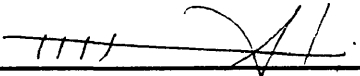
ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

Thesis Acceptance

Acceptance for the faculty of the Graduate College, University of Nebraska, in partial fulfillment of the requirements for the degree of Master of Science, University of Nebraska at Omaha.

Committee

Name	Department
Hesham Ali	Computer Science
Azad Azadmanesh Azadmanesh	Computer Science
Hamid Shafiq	Computer & Electronics Eng.



 Chairman

 Date 11/29/99

A Genetic-Based Approach
To Multi-layer Channel Routing in VLSI Design

Mark P. Cloyd

University of Nebraska at Omaha, 1999

Advisor: Dr. Hesham H. Ali

As our reliance on electric and electronic devices increases, the need to improve the design and manufacture of integrated circuits (ICs) grows. A microchip can be bettered if it can be made more powerful, smaller, cheaper, and/or more easily manufactured. The physical design phase of chip manufacture offers significant room for improvement.

This thesis intends to investigate the detailed channel routing phase of VLSI physical design. Channel routing has been seen to be intractable, in that an optimal solution may require too much time for calculation. Constraints severely limit many algorithms to an approximate solution. The heuristics developed thus far have fallen short of an optimal solution for similar reasons.

This thesis presents an evolutionary approach to channel routing. A genetic algorithm makes use of a parent solution to derive next generation solutions in an attempt to overcome local minimums. A three-layer approach is used to evaluate the use of various layering schemes, and also to reduce the number of constraints involved. When multi-

terminal nets or multi-nets are involved, a division parameter is used to determine if the best results are generated by treating the multi-net as a whole, or broken down into as small as two-terminal nets. A greedy approach is used to generate the original parents, and a compaction algorithm is used to further improve the output. The results are then represented three-dimensionally by a computer-aided design program, where it can be analyzed for accuracy.

Table of Contents

Thesis Acceptance	I
Abstract.....	II
Chapter 1 Introduction.....	1
1.1 The History of VLSI Development.....	1
1.2 Previous Approaches to Channel Routing	4
1.3 What Has Been Lacking	6
1.4 Project Identification.....	7
1.5 Expected Results.....	10
1.6 Thesis Organization	11
Chapter 2 Defining the Problem and Its Terminology	12
2.1 An Overview of Channel Routing	12
2.2 Definition of Terminology.....	13
2.3 Problem Formulation	15
2.4 Input Analysis	16
2.5 Summary	20
Chapter 3 An Overview of Prior Research in Channel Routing.....	21
3.1 Two Layer Approaches.....	22
3.1.1 The LEA Family of Algorithms.....	22
3.1.2 The Dogleg Router.....	23
3.1.3 The Net Merge Router	25
3.1.4 Yet Another Channel Router	26

3.1.5	The Greedy Channel Router	27
3.1.6	The Hierarchical Method	28
3.1.7	The Genetic Algorithm	28
3.2	Three-Layer Approaches	30
3.2.1	The Extended Net Merge and CWL Channel Routers.....	30
3.2.2	The Hybrid HVH-VHV Router	31
3.3	Summary.....	32
Chapter 4	The Genetic Algorithm.....	33
4.1	An Overview.....	33
4.2	How the Genetic Algorithm Works	34
4.3	The Mutation Process	35
4.4	The Selection Process	38
4.5	Global Control Parameters.....	39
Chapter 5	The Multi-layer Channel Router.....	41
5.1	The Basis for the Router Process.....	41
5.2	Data Structures Used.....	44
5.3	Process Elaboration.....	46
5.3.1	Creating Nets	47
5.3.2	Finding the Maximum Clique.....	48
5.3.3	Evaluating the Vertical Constraints, Cycles and the Longest Path.....	49
5.3.4	Creating Parent Solutions	52
5.3.5	Improving the Solution by Compaction.....	54

5.3.6	Expressing the Value of the Solution.....	56
5.3.7	The Mutation Phase	58
5.3.8	Choosing New Parents.....	60
5.3.9	The Reporting Mechanism.....	61
5.4	The Main Process.....	63
5.5	Generating a Graphic Output	64
Chapter 6	Experimental Results.....	67
6.1	The Problem Sets Used.....	67
6.2	Separation Value	70
6.3	The Control Parameter p_0	73
6.4	The Generation Size.....	75
6.5	The Control Parameter R	76
6.6	Using Elite Mode	77
6.7	Two-Layer Versus Three-Layer Analysis	79
6.8	Time Complexity	81
6.8.1	Time Complexity of the Analysis Phase.....	81
6.8.2	Time Complexity of the Parent Creation.....	82
6.8.3	Time Complexity of the Genetic Algorithm.....	83
6.9	Comparing the Parents to the Final Solution	84
6.10	General Observations.....	85
Chapter 7	Conclusions.....	87
7.1	A Recap of the Benefits	87

7.2 Positive and Negative Effects	88
7.3 Directions for Future Research	89
Bibliography	90
Appendix A: Source Code	92

List of Figures

Figure 1.1	The Design Cycle of VLSI	2
Figure 1.2	The Physical Design Process	3
Figure 2.1	Typical Channel Layout	13
Figure 2.2	Track and Vertical Layout	15
Figure 2.3	Determining the Maximum Clique from the HCG	16
Figure 2.4	Determining the Longest Path from the VCG	18
Figure 2.5	Identifying Cycles in the VCG	19
Figure 3.1	An Example of the Left-Edge Algorithm	23
Figure 3.2	The Dogleg Routing Algorithm	24
Figure 4.1	The Genetic Algorithm	35
Figure 4.2	Mutation by Switching	36
Figure 4.3	Mutation by Crossover	36
Figure 4.4	Mutation by Inversion	37
Figure 5.1	Structure for the Basic Net Storage	42
Figure 5.2	HVH Layering Model	42
Figure 5.3	VHV Layering Model	43
Figure 5.4	Structure for the Solution Storage	45
Figure 5.5	Net Creation	47
Figure 5.6	Maximum Clique	49
Figure 5.7	Creating the Vertical Constraint Graph	50
Figure 5.8	Cycle Check	51

Figure 5.9 Longest Path.....	52
Figure 5.10 Making the Parent Solutions	53
Figure 5.11 Checking for Free Space	53
Figure 5.12 The Compaction Algorithm.....	55
Figure 5.13 The Expansion Algorithm	55
Figure 5.14 The Costing Algorithm.....	57
Figure 5.15 Counting Vias.....	58
Figure 5.16 Performing Mutation	59
Figure 5.17 The Sorting Algorithm	61
Figure 5.18 The Output Algorithm	62
Figure 5.19 AutoCAD® Output Screen Showing Track and Layer Assignment.....	65
Figure 6.1 The Effect of Separation on Time	69
Figure 6.2 The Effect of Separation on Cost	70
Figure 6.3 Outputs Based on Separation.....	71
Figure 6.4 Layering Modes for 40 Terminal Problem Set.....	80

List of Tables

Table 6.1	Definition of Problem Sets	68
Table 6.2	Characteristics of 40 Terminal Problem Sets	69
Table 6.3	Comparison of Separations of 2 and 4.....	72
Table 6.4	Time and Cost on 40 Terminal Sets with Separation	72
Table 6.5	Best p_0 Values	73
Table 6.6	Effect of p_0 on the 40 Terminal Problem Sets.....	74
Table 6.7	The Effect of Generation Size on Cost and Time.....	75
Table 6.8	The Effect of R on the Cost	77
Table 6.9	Effect of Elite Mode on Cost and Time.....	78
Table 6.10	Analysis of Layer Mode on the 40 Terminal Problem Sets.....	79
Table 6.11	Comparison of Greedy and Genetic Algorithms	85

Chapter 1

Introduction

1.1 The History of VLSI Development

The development of the integrated circuit has changed dramatically over the last thirty-five years. Integrated circuits (ICs) have affected electronic and electrical device production in almost all facets of life. Circuit designs have changed from as small as 1000 circuits in a chip during 1965, to as many as five and one half million transistors during 1995. The potential exists to create a chip with over a billion transistors in the next 10 to 15 years. There can be seen several levels of integration from Small Scale Integration, using typically as many as 20 gates per chip, to Medium Scale Integration, incorporating as many as 200 gates per chip. Large Scale Integration normally involves as many as 5000 gates, while Very Large Scale Integration (VLSI) incorporates over 5000 gates per chip. VLSI design is now the largest part of production and research.

The physical design process for VLSI design involves multiple layers of metal oxide and silicon insulator, arranged on a silicon wafer in such a way to form gates, transistors, and circuits. An on-going limitation is the precision available in manufacturing, which limits the size of the smallest feature within the chip. An additional issue to be researched, is the increasing complexity of circuitry. As the feature size is reduced, either the chip size can be reduced, or there is the availability of more circuitry per chip. The latter requires

more time and effort to design and test. Changes in technology and types of materials used in the chip design can facilitate small changes in the precision size, which increases the complexity of the circuit design process. This design process deserves further investigation, because improvements to it can reduce the time from conception to market delivery.

The design process is shown in Figure 1.1. It begins with the system specification. In this phase, the function of the circuit is determined in general terms. From the system specification, a functional design of the circuitry is created. This is done in terms of a flow chart of the steps needed to perform the function. The functional design then leads to the logic design. In this phase, the flow chart is converted into an algebraic expression that represents the output from the process. The circuit design phase follows, in which the algebraic

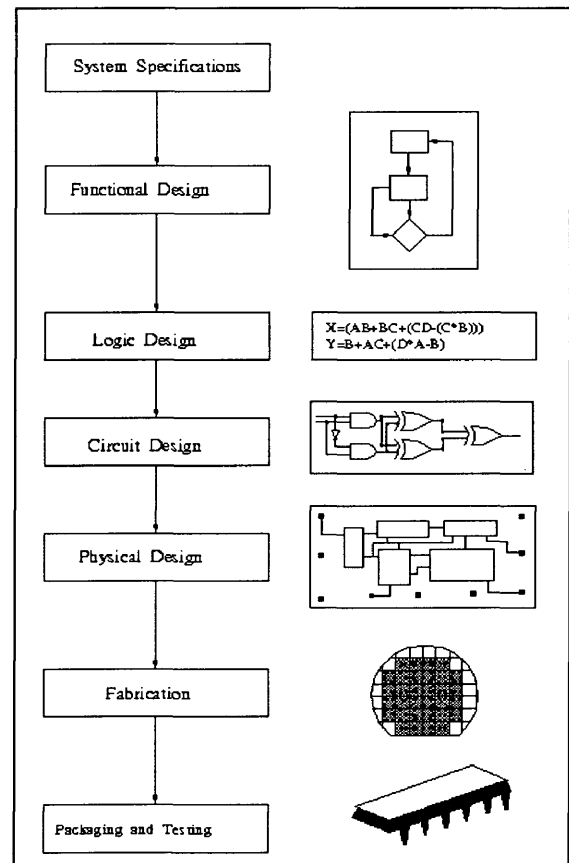


Figure 1.1 The Design Process

equation is analyzed and processed into a logic gate representation of the circuit. After the circuit has been laid out, the physical design phase takes the circuit design, and processes it into the physical layout of predefined circuit blocks and the connection leads

between those blocks. Finally, the fabrication cycle or phase implements this physical design into a set of chips that are then packaged and tested for accuracy before finally shipping to their ultimate destination.

The physical design process, as figure 1.2 shows, can be broken down into the following stages: First, the circuits are partitioned into blocks. Because of the nature of VLSI design and the large number of gates involved, it is impossible to work with all of them at one time. The gates are partitioned into small enough blocks to make them manageable. After the partitioning has been determined, the floorplanning and placement stage is performed. In this stage, the gates are arranged

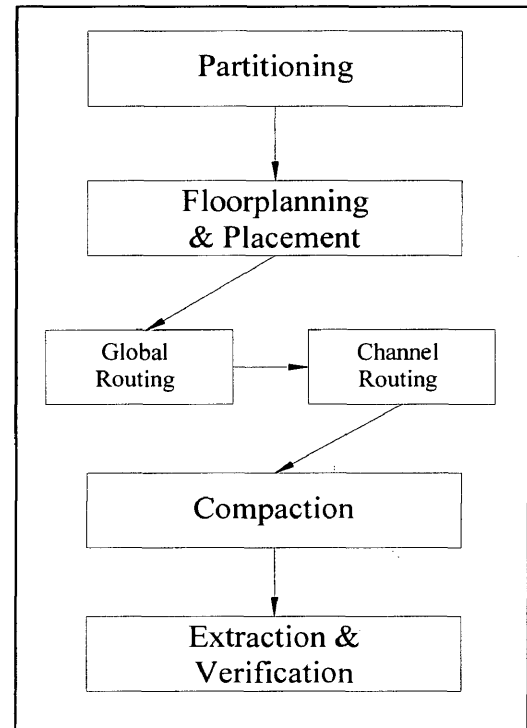


Figure 1.2 Physical Design

in such a way so that they are closest to the gates to which they will be interconnected. After the floorplanning and placement stage is accomplished, the routing stage begins. The routing stage occurs on two levels. The circuit is first globally routed to define the major circuits that will connect the different blocks and gates together. This is performed in general terms, following which the channel routing is performed. In the channel routing phase, the circuits, or nets, are given specific locations between chips or between blocks to connect the different blocks

together. Each connection requires a specific amount of space to make those connections, without interfering with other circuits in the area. After the channel routing, the compaction phase looks for unused areas of the layout, in an attempt to reduce the overall chip size. Finally, the chip enters an extraction and verification stage, where the chip is checked for design rule accuracy of the circuits. The circuit is checked to make sure it falls within fabrication limitations.

Each phase of the physical design process is computationally intense, and so improvements can yield significant benefits. The focus of this thesis is in the channel routing phase. Channel routing has been proven to be an NP-complete problem, meaning that no single algorithm has been found that handles all situations to provide an optimal solution. With these points in mind, we offer a multi-layer channel routing algorithm. It is intended to accept, as input, a wide range of problem sets, and return as output a near-optimal solution for the routed nets. We must first look at the previous approaches that have been taken in trying to solve this problem.

1.2 Previous Approaches to Channel Routing

Various approaches have been taken to resolve the channel routing problem. The problem is of an intractable nature, as an optimal solution is not always practical. In some instances, an optimal solution requires too much time and can be very computationally complex. Constraints can also cause optimal solutions to fail.

Heuristics have been developed to try to find a reasonable solution, but tend to have problems falling into local minimums. In 1971, Hashimoto and Stevens proposed the Left-Edge algorithm [HS71], which organized the nets in order of ascending left node, then used a type of greedy algorithm to assign nets to specific locations. Deutsch offered a dogleg router algorithm [DD76]. A dogleg is a vertical segment connecting two horizontal net segments. In this algorithm, analysis begins at either end of the channel, with doglegs allowed between multi-terminal nets, to reduce track width. A multi-terminal net, or multi-net as it will be referred to, is a set of three or more pins requiring a connection between all of them.

More recently, Yoshimura and Kuh offered a net merge channel router, called the YK algorithm, which partitioned the connection region, or channel, into zones, and then routed each zone [YK82]. The nets of adjacent zones are combined to create a composite net that could be placed in a track. Reed, Sangiovanni-Vincentalli, and Santamauro proposed the YACR2 (Yet Another Channel Router) algorithm that made use of vacant columns, and tracks to resolve vertical constraint violations [RS85]. The greedy channel router was developed by Rivest and Fiduccia to assign tracks to the nets column by column, from left to right [RF82]. Its intent was to reduce track width, allow doglegs in any column, and combine split horizontal nets where convenient. Burstein and Pelavin offered an algorithm based on a hierarchical method [BP83]. The nets are routed globally in the channel, and then the channel is repeatedly divided into smaller sections to be rerouted as necessary to better define the track assignments.

Last year, Jingsen Zheng offered an evolution-based, or genetic, algorithm for two layer channel routing [ZJ98]. In his graduate thesis, he presented a process that made use of interim solutions to evolve new solutions. The intent was to overcome local minimums by accepting less than optimal solutions as interim steps working toward an optimal solution.

1.3 What Has Been Lacking

Prior solutions to the channel routing problem have been inconclusive. Optimal algorithms can not handle constrained situations, or can take too much time to process when evaluating large data sets. An evaluation of the heuristic algorithms reveals their shortcomings. Several do not support doglegs or cyclic vertical constraints. This leads to a channel width that is wider than necessary. Some algorithms give solutions with excessive doglegs and vias, while others can be rather complicated to implement.

The genetic algorithm is of special interest, as it is the basis for this thesis. We are extending the work of Zheng in his thesis, on an evolution-based two-channel router. Zheng indicated the problems he encountered with the genetic algorithm [ZJ98]. Horizontal and vertical constraints posed the greatest challenge to overcome. The longest path, as defined in the vertical constraint graph (VCG) can also affect the time required to find a solution. Breaking down multi-nets into two-terminal nets was beneficial in some cases, but detrimental in others. He also indicated that the technique used to create the

initial populations also played a part in the time required to derive a solution, and in the quality of that solution.

All of these shortcomings help to point out that the channel routing issue is not a simple one. The heuristics can have difficulty escaping from local optimums. This sets the stage for less than optimal results. If we can reduce the constraints surrounding a routing situation, we can more easily and more accurately route the channel with a minimum track width. If we can escape from local optimums, we may be able to derive a more optimal solution.

1.4 Project Description

It is the intent of this thesis to evaluate alternative methods of channel routing. This phase of VLSI design is known to be NP-complete, in that there is not a general solution available that is optimal and can be performed in a reasonable amount of time for all situations. There have been many heuristics proposed, but they have been seen to be too narrow in scope or take too much time to process the input. Heuristics also tend to involve too many constraints, which can affect the output dramatically.

We intend to extend upon the work that Jingsen Zheng presented in his thesis on an evolution-based approach to channel routing. In this thesis, we will investigate the effect of several factors on the channel routing question to allow for a more optimal solution.

- A three-layer approach
- A separation parameter, used to break multi-nets into smaller sub-nets
- The value of the parameter used in selecting valid solutions, p_0
- The value of the variable used to define the termination point, R
- Using an ‘elitist mode’ in selecting interim solutions
- The effect of the generation size, or number of children generated by a parent solution

An evolutionary approach is still employed. We will use a three-layer approach, as the latest developments in VLSI have allowed a third layer for circuitry. With this additional layer, we have the ability to evaluate an HVH (horizontal-vertical-horizontal) layering scheme and compare it to a VHV (vertical-horizontal-vertical) layering scheme. These three-layer approaches can also be compared to the two-layer approach previously used. We intend to use an array to maintain our list of net segments’ parameters, and another array to store the track and layer onto which each net segment is assigned. This would allow the use of doglegs in the channel routing.

A separation parameter will be employed to divide multi-nets into smaller network sections. Using this parameter, we can evaluate a multi-net in its complete form, and in smaller sections, with the two-terminal net as a lower bound. This separation parameter can be adjusted to break the multi-nets into smaller multi-nets to fine-tune the output. We would like to see how this parameter affects the overall solution.

The values of two control parameters, p_0 and R , determine which solutions are kept to be parents for succeeding generations, and what the point of termination should be, respectively. In evaluating which solutions to keep, two final selection methods can be incorporated. In an elitist mode, only the top solutions are retained to create future generations. In a non-elitist mode, a random sample of the solutions is kept to become parents. The number of children derived from each parent can also affect the quality of the solution, as well as the time required to process the problem set.

A greedy routine will be used to create the initial populations. In this approach, the nets are placed in the channel in a 'first come, first served' basis. The first net is randomly chosen, and then the nets are processed in order to place them in the layers and tracks, based on their terminal placement. Nets that have both terminals located on the top of the channel will naturally be placed toward the top of the channel and those with both terminals on bottom will be placed toward the bottom of the channel. This should give better initial populations, so that the evaluation will be more robust, and hopefully allow more time to overcome local optimums.

We also intend to incorporate a compaction phase in the circuit design. As the tracks undergo mutation, the compaction phase will help further reduce the track width. In evaluating the feasibility of the generation, a costing formula will be employed with weights for horizontal and vertical violations as well as the track width.

Finally, the results of the channel routing algorithm will be used as input to AutoCAD[®], a computer-aided design program. A three-dimensional representation of the channel routing will be generated, which can be viewed and rotated to check for accuracy and to analyze it to determine whether it approaches an optimal solution.

1.5 Expected Results

By employing an evolutionary or genetic approach to channel routing, we allow some less than optimal solutions as interim generations. The advantage to including these solutions is that it allows the algorithm to mutate that solution into a possibly better overall solution than is currently thought to be 'best'. Using this approach, we can attack the problem from different angles, so as to avoid falling into a local minimum that in itself tends to only offer a poorer solution when trying to improve upon it.

By including a third metal layer in the evaluation, we take advantage of several things. First of all, we have the ability to reduce the track width to as much as half of the two-channel version. This can lead to a smaller chip design and possibly less wire length. Secondly, we also have the ability to remove vertical constraints from the picture. By using a VHV layering model, we can route all net nodes efficiently where before a vertical constraint would force more track width to allow room to overcome the constraint.

By using a separation parameter, to adjust how we handle multi-nets, we should be able to better minimize track width by the inclusion of doglegs. We can also evaluate the option of handling a multi-net as one large net versus several smaller nets. We expect to find a middle ground that better optimizes the solution, but doesn't run into the extended time constraints required for a large number of two-node nets to be processed. Lastly, incorporating a compaction phase helps to keep the intermediate solutions from having excessively wide tracks with significant amounts of unused track space.

1.6 Thesis Organization

In this thesis, we will define the problem at hand, including definitions as they apply to channel routing. In chapter two, the basic terminology will be identified, as well as the use of graph models to evaluate the conditions for the routing necessary. The objectives of minimizing channel width and connections, and eliminating constraints, will be detailed, as well. Chapter three provides an overview of the previous work done in the study of channel routing from a two-layer and a three-layer aspect. An overview of the genetic algorithm will be presented in chapter four, followed by an explanation of the genetic algorithm for the multi-layer channel router in chapter five. Chapter six discusses an analysis of the experimental results, as well as the performance of this algorithm. Finally, in chapter seven, conclusions are offered, as well as possible options for future research. We must next define the problem and the terminology used in the discussion of channel routing.

Chapter 2

Defining the Problem and Its Terminology

2.1 An Overview of Channel Routing

As mentioned in the previous chapter, the problem of channel routing is difficult at best. To further analyze the problem, we must first look at the task at hand. The blocks of circuits have been physically arranged, relative to each other, and the interconnections between the various blocks have been identified. The floorplanning and placement stage has allotted areas between these blocks that are to be used to route the interconnections between circuits. These areas are divided into rectangular regions, and those regions are evaluated. The regions that have a circuit block on opposing sides are known as the ‘channels’. Two-dimensionally speaking, channels can exist both horizontally and vertically, but they are analyzed in the same fashion, without a loss of generality. A region having circuit blocks on all sides is called a switchbox, and there can be 2D switchboxes with circuitry on four sides, and 3D switchboxes, which have circuitry on all six sides. Switchbox routing requires special analysis, and is beyond the scope of this thesis, so we will limit our discussion to channel routing.

The channel is defined by the terminals along its opposing sides. Our evaluations will be made on channels arranged in a horizontal orientation, with the terminals located along the top and bottom of the channel. The various parts of the channel can be identified in

figure 2.1. The channel has its top and bottom row of terminals, with the terminals numbered to identify their interconnections. These interconnections, or nets, are the elements we must arrange in an orderly fashion. All pins marked with the number one are to be connected, as are the pins marked with the number two, and so forth. The pin location is defined by its count from the left most pin, counting to the right. The channel length is defined by the number of pins included in the problem set. Those pins with the number zero are empty terminals and are not to be connected. The terminal list contains the order of the net connections for the pins, organized from left to right.

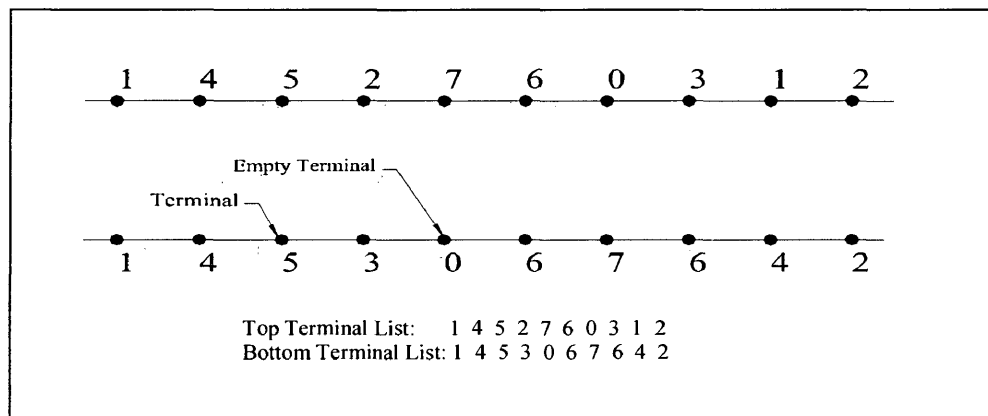


Figure 2.1 Typical Channel Layout

2.2 Definition of Terminology

Several terms will be used throughout this analysis of the channel routing problem. It is appropriate that we define them at this time.

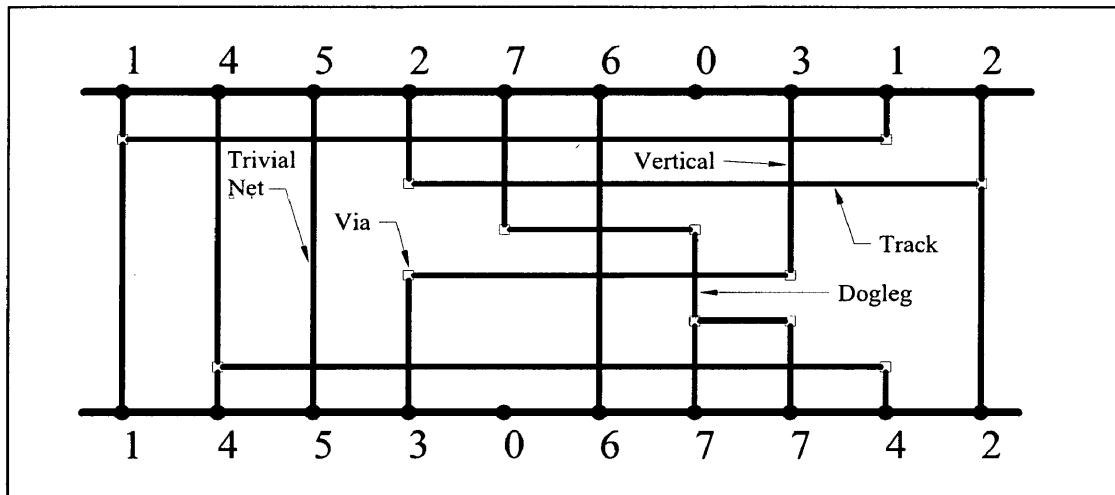


Figure 2.2 Track and Vertical Layout

- **net** – the connection between similarly numbered pins
- **multi-net** – a net that contains more than two connections to pins
- **sub-net** – a portion of a multi-net having two or more terminals
- **terminal** – the pin or connection point of a net to the circuit block
- **trivial net** – a net that connects a terminal from the top terminal list with a terminal directly below it in the bottom terminal list
- **track** – the horizontal path that a net may travel along the channel
- **vertical** – the vertical path that a net travels from its horizontal track to the terminal of the circuit block
- **via** – the connection between layers of circuitry that connects the track to the vertical of a net
- **layer** – refers to the physical layer within the circuit design on which the circuit lies
- **HVH** – a three-layer model with two horizontal layers, one on the top and bottom, with a vertical layer between

- **VHV** – a three-layer model with two vertical layers, one on the top and bottom, with a horizontal layer between

We are using a reserved layer model. This means that one layer is designated strictly for tracks, while another layer is strictly for verticals. In our treatment of the problem, we will be using a two-layer and a three-layer model, with the third layer designated for either tracks or verticals, as the problem set deems necessary. We complete the connection between layers by implementing vias. Vias are used at the ends of tracks to connect to the verticals, and may be used within multi-nets to connect two segments where a dogleg exists. A dogleg is created when a multi-net is assigned to more than one track. The dogleg is the vertical part of the multi-net between tracks, as shown in Figure 2.2.

2.3 Problem Formulation

Our objective is to create an arrangement of nets by assigning them to particular layers and tracks within the channel so that they accurately complete the electrical connections between blocks of circuits. The number of tracks required should be minimized, to offer a more compact layout. Obviously, the exclusion of conflicts between the assigned placement of nets is essential. Other goals can be to minimize the overall length of the routing connections, or the total wire length, and the minimizing of the number of vias required.

To better understand what kind of solution is expected, an analysis of the problem set is in order. Several parameters exist that can give us an idea of what to expect for an optimal solution. The use of graph models can be employed to aid us in this analysis.

2.4 Input Analysis

Analyzing the problem set offers us several important parameters that help us identify what an optimal solution should look like. We can first look at the number of tracks necessary to complete the connections. We can determine the minimum number of tracks required by comparing the nets' beginning and ending points, with respect to how they overlap. Two nets that require a common distance of the channel can not be assigned to the same track. By looking at all the nets' spatial relationship to each other, we can determine the minimum number of tracks required. Figure 2.3 shows the terminal layout

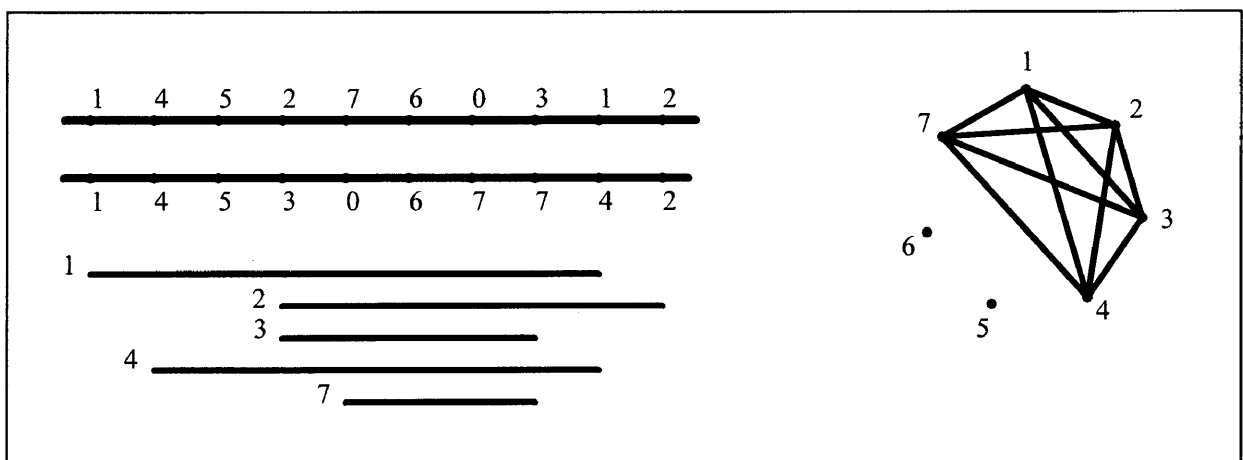


Figure 2.3 Determining the Maximum Clique from the HCG

of a channel. Below it, the nets are arranged beside each other based on their starting and ending points in the channel. Note that nets 5 and 6 are trivial nets and, as such, do not require a track for routing.

If we draw a vertical line through the arrangement of nets, it may cross one or more of them. If we find the placement that crosses the maximum number of nets, we can count the number of nets crossed. This represents the point where the channel would need to be the widest. We can incorporate graph theory to help us in determining this value. The graph for the problem set is created with the nets as nodes, and the overlaps as undirected edges between the nodes. This graph is known as the horizontal constraint graph, or HCG. We can extract the maximum number of nets overlapping each other by calculating the largest group of nodes in which each node contains an edge to every other node in the group. This is more commonly referred to as the maximum clique of the graph. In the HCG, the maximum clique gives us the maximum overlap of the nets, or the minimum track width for the channel. If we use a three-layer model that incorporates two horizontal layers, HVH layering, we have the ability to move half of the nets' tracks to that extra horizontal layer. This allows us to reevaluate the problem set so that now the minimum tracks required is one-half of the maximum clique.

We can also evaluate any possible conflicts between the top and bottom terminal lists. These conflicts are called vertical constraints, and can be easily evaluated using a graph model. Again the nets are used as nodes of the graph, but in this case, we compare the

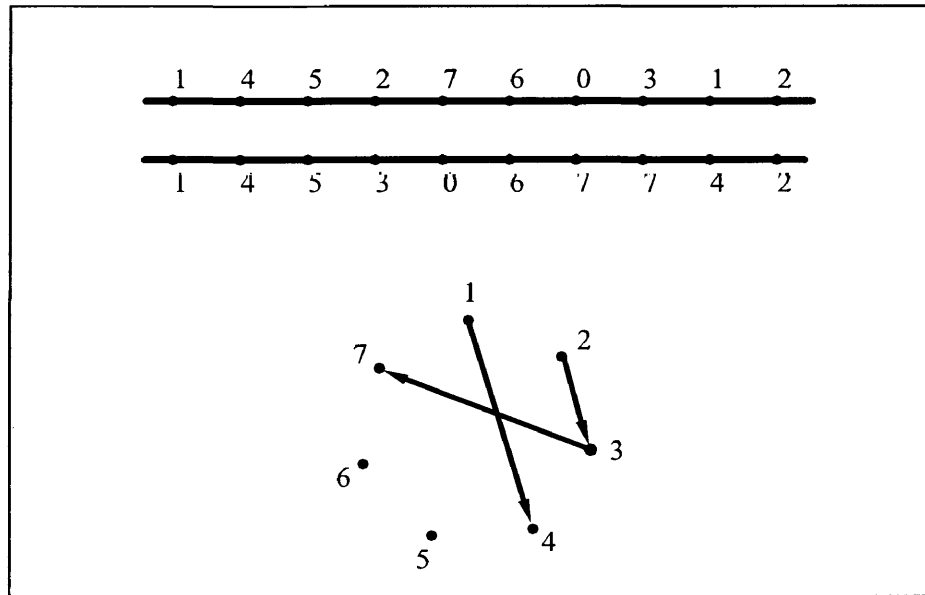


Figure 2.4 Determining the Longest Path from the VCG

top and bottom terminal lists, column by column, looking for a difference between the top and bottom terminals. This difference is considered a constraint in that the nets can not have their vertical components overlap. Figure 2.4 shows the terminal list with its associated vertical constraint graph, or VCG.

In the VCG, the edge represents the constraint between the upper and lower terminal, and therefore, is a directed graph. If we examine the VCG in figure 2.4, we see a directed edge between nets 2 and 3, 3 and 7, and 1 and 4. Each of these edges indicates the need for at least two tracks to allow room for the verticals of the nets to connect with their respective tracks. To evaluate this graph, we look for the longest directed path between nodes. The length of this path represents a lower bound on the number of tracks required

to connect the nodes of the problem set. In the example above, the longest path would be two for the path from 2 to 3 and 3 to 7.

Another important parameter, which can be extracted from the VCG, gives us a different perspective on constraints. We must also check the vertical constraint graph for the existence of cycles within the directed edges. If a cycle can be detected, we will be unable to define the longest path, as the cycle would provide an infinite path length. Secondly, a cycle indicates that the routing problem being evaluated has a more serious problem. Figure 2.5 shows the terminal layout containing a cyclic vertical constraint. Unless doglegs are used, there is no way to successfully route the example in a two-layer model. A three-layer model, however, will solve the problem by allowing one net's

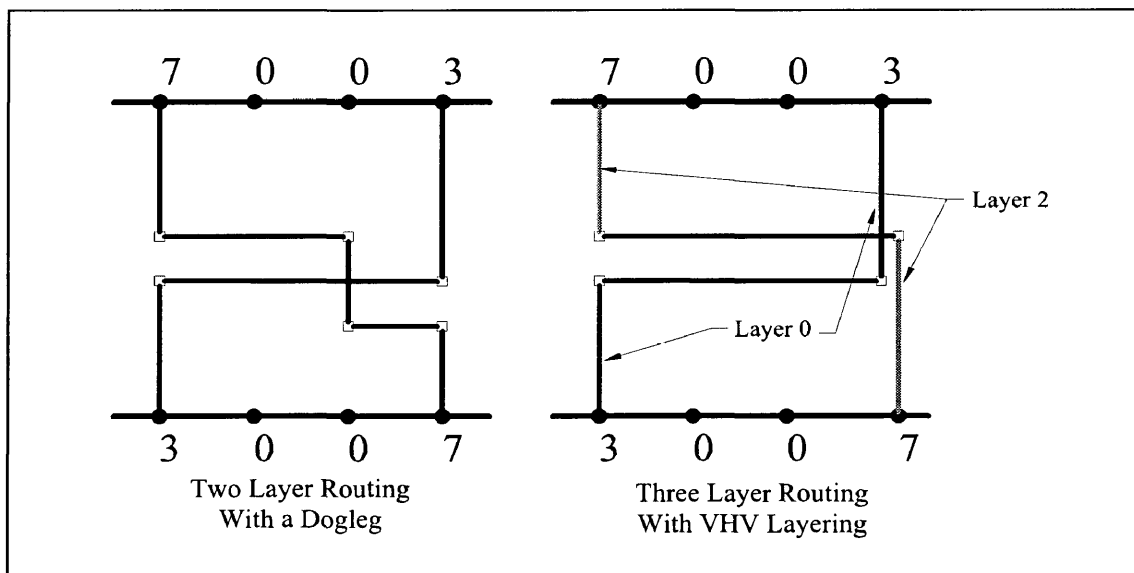


Figure 2.5 Identifying Cycles in the VCG

verticals on the first layer of a VHV layering mode, while the other net's verticals can overlap by placing them on the second vertical layer.

2.5 Summary

We now have an understanding of the channel routing problem, and how to evaluate the problem set so that the parameters of an optimal solution are known. We have defined the terminology used in the study of channel routing, and how we evaluate a problem set. If we can find the maximum clique of the HCG, the longest path of the VCG, and whether the problem set has cyclic vertical constraints, we know what we should expect our algorithm to produce for an optimal solution.

Many others have developed algorithms in an attempt to solve the channel routing problem. We will now provide an overview of several of those processes.

Chapter 3

An Overview of Prior Research on Channel Routing

The channel routing problem has existed for close to 35 years. There have been many approaches to the problem, from the single layer approaches, such as those reviewed by M. Marek-Sadowska and T. T. Trang [MT83]. While they appear to be a simpler procedure, there is still intensive computation involved. Single-layer routing problems are known to be NP-complete [RD84].

Two-layer approaches have been discussed in chapter one, but will be presented in greater depth in section 3.1. Three-layer approaches have been more recently proposed. They will be presented following the two-layer algorithms in section 3.2.

We would like to offer a new method of evaluating the channel routing problem. The problem has been approached from many angles, but none have been able to completely encompass all aspects of any given problem set. All seem to have their special circumstances in which they excel. If we look at what has been proposed before, we can then consider what other ways we can break down the problem to provide a more optimal answer for a wider range of input conditions.

3.1 Two-Layer Approaches

As mentioned in chapter 1, several algorithms have been developed to tackle channel routing from a two-layer point of view. As J. Zheng mentions in his thesis, most of the research in this category has been on heuristic methods, due to the NP-hard nature of the problem [ZJ98]. We will examine the LEA family of algorithms, the dogleg routing algorithm, the net merger router, the YACR2 router, greedy channel router, hierarchical method, and finally, the genetic algorithm.

3.1.1 The LEA family of Algorithms

The Left-Edge Algorithm, LEA, was first proposed by Hashimoto and Stevens [HS71]. This algorithm produces an optimal solution for the problem sets it can process, so it is not one of the heuristic algorithms. The algorithm has a limitation in that it will not work on problem sets with vertical constraints.

The Left-Edge algorithm begins by sorting the nets in ascending order of their left-most terminal positions. It then places the nets, processing them in their sorted order, into the first available track of the channel. It always begins with the top track and works toward the bottom. A reserved layer policy is used here, so that there is a layer for the verticals and a separate layer for the tracks. An example is shown in figure 3.1.

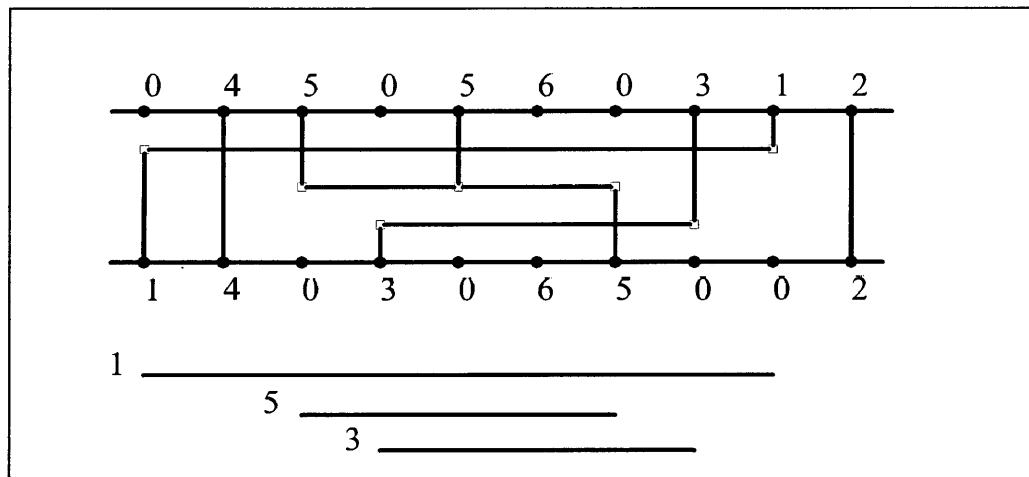


Figure 3.1 An Example of the Left-Edge Algorithm

This algorithm, while easy to prove its optimal nature, is not very practical. The restriction of not allowing vertical constraints is too limiting, as most channel routing problems contain some vertical constraints. Sherwani points out that it has been found to be a good initial router, which would lay out the basis for the routing scheme, after which, a clean-up procedure could be employed to handle constraints [SN95].

3.1.2 The Dogleg Router

The LEA algorithms were seen to have another shortcoming. Since they assigned the entire net to a single track, in some cases it led to the use of more tracks than was actually necessary. D. N. Deutch (1976) proposed an algorithm he called the Dogleg Router. It was based on the Left-Edge theory, but it would break multi-nets down into simple two-

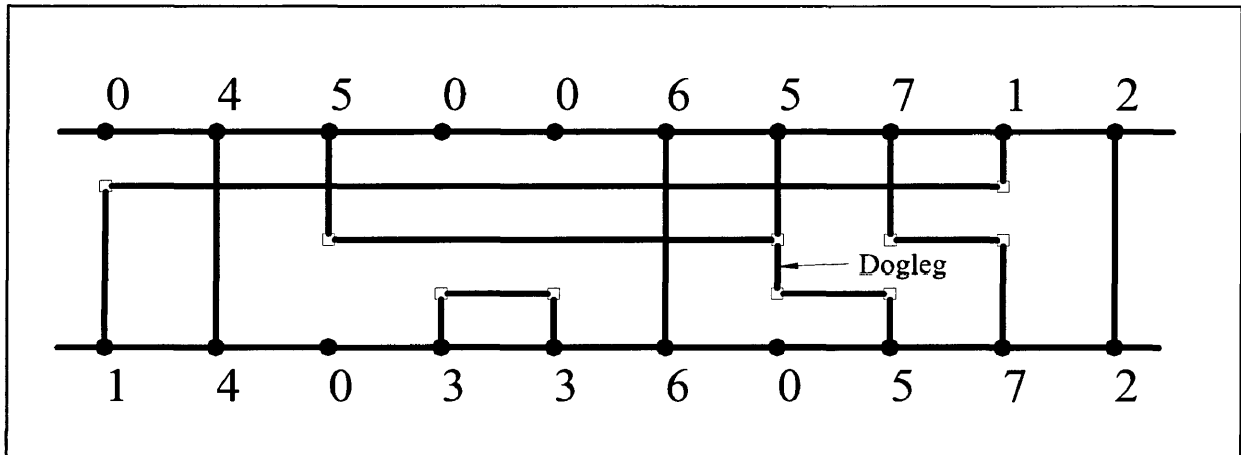


Figure 3.2 The Dogleg Routing Algorithm

terminal sub-nets before placing them. The vertical between two sub-nets was used to connect them together.

While the algorithm offered benefits over the basic Left-Edge Algorithm, it did add a wrinkle of its own. It could handle vertical constraints between the top and bottom terminal list, but still did not handle cycles in the vertical constraint graph. The process of finding the least number of doglegs to use has been shown to be NP-complete [SG85]. Badly placed doglegs can even increase the number tracks the solution requires. Another problem caused by the excessive use of doglegs is in the length of the path. As this algorithm is also a reserved layer algorithm, extra doglegs would require extra vias. This lengthens the path and can reduce the reliability of the circuit.

3.1.3 The Net Merge Router

Another approach to improving upon the LEA, by handling vertical constraints, was presented by Yoshimura and Kuh with their net merging algorithm, called the YK algorithm [YK82]. While previous research made use of the horizontal constraint graph, HCG, to determine the minimum number of tracks required, YK algorithm also makes use of the vertical constraint graph, VCG, to analyze relationships between the nets. The intention is to combine nets that do not have vertical constraints between them so that the number of nets and tracks required can be minimized.

Two conditions must be met to allow the nets to be combined:

1. There cannot be an edge between the nodes in the HCG.
2. There must not be a directed path from one node to the other in the VCG.

By evaluating two nets without common constraints as one net, the overall amount of calculation can be reduced. The process looks at the channel in zones, rather than looking at it on a column by column basis, so its efficiency is improved. If net i from one zone meets the conditions from net j in another zone, they can be merged together to simplify the analysis for track assignment. This process is repeated until nets can not be combined further, after which the merged nets are placed in tracks.

The process produced good results, although it was restricted by two conditions, it did not allow doglegs, and could not handle cycles in the VCG. Finding optimal pairs of nets for

the merging process was difficult to perform. When merging nets, the future effects were difficult to predict.

3.1.4 Yet Another Channel Router

Others evaluating the left-edge approach were finding other ways to handle vertical constraints. Reed, Sangiovanni-Vincentalli, and Santamauro proposed the YACR2 (Yet Another Channel Router) algorithm based on the following observations [RS85]. A vertical constraint is a localized problem. Further, there is usually unused space in the surrounding verticals and tracks that can be used to resolve this constraint, by employing a localized maze routing approach. If constraints cannot be resolved, extra tracks can be added to offer additional space.

A basic LEA is employed to initially assign track placement. Several maze routing techniques are employed to reroute constraints to one side or the other. The entire process is performed in a four-phase process. It begins by finding the column with the highest density in the channel. Phase 1 routes the high-density column, and then phase 2 routes the columns to the right. Phase 3 uses a modified LEA approach to route the left side columns by using a right-edge evaluation, following which phase 4 uses various maze re-routing techniques to resolve constraints.

While YACR2 was efficient, it was a much more elaborate implementation. It did handle vertical constraints well, while minimizing the number of tracks required.

3.1.5 The Greedy Channel Router

In 1982, Rivest and Fiduccia developed another twist on the LEA, which they called the greedy channel router [RF82]. They observed that assigning an entire net to just one track was very restrictive. They proposed a process that evaluated the channel from left to right, column by column. In the evaluation, each net within that column was placed in a track, and then the process moves to the next column. In this way, nets may change tracks part way through its path, i.e. a dogleg can occur. There are no restrictions to where a dogleg can occur or how often.

If two ends of a net cannot be connected immediately, the nets are placed on tracks to be pathed to the right. If space develops, the router will use a dogleg to reduce the number of tracks separating them, until an unused vertical is available to connect them. The nets when apart are referred to as split nets, and when the net is finally connected with a vertical, it is referred to as a collapsed net.

The greedy algorithm's biggest advantage is in handling cycles in the VCG. It deals favorably with vertical constraints, and minimizing the number of tracks needed, but has as its weakness the tendency to use too many doglegs and vias.

3.1.6 The Hierarchical Method

Others took very different approaches to the topic of channel routing. In 1983, M. Burstein, and R. Pelavin presented their hierarchical router for two-layer channel routing [BP83]. This method used a ‘divide and conquer’ point of view to net placement. In this method, the channel is reduced in height to a $2 \times n$ channel and this channel is used to route all of the nets. Special Steiner trees are created to route the connections globally. A cost is calculated for each step in the connection path, moving either horizontally or vertically. The net is initially pathed by the lowest cost and then the channel is recursively expanded out as the nets are divided out into individual rows.

The biggest drawback to the hierarchical method is that it cannot handle cycles in the VCG. Otherwise, it also uses a reserved layer model and does allow doglegs. It is reasonably efficient and produces a very good solution.

3.1.7 The Genetic Algorithm

As this thesis is an extension of Jingsen Zheng’s graduate thesis in 1998, it is important to distinguish his work, for it lays the groundwork for our current research. The genetic algorithm was first proposed by A. T. Rahmani and N. Ono [RO93]. Their approach was to look at the channel routing problem as an optimization problem. A solution to the problem was represented as a vector of positive numbers, with a cost associated that

measured how close the solution came to an optimal solution. The cost was calculated based on the number of tracks required and number of violations that occurred. The process created many solutions with the goal of finding the solution with the lowest cost.

Zheng used this as the basis for his genetic algorithm in the following manner. The problem set was initially analyzed to determine its relevant characteristics. The maximum clique was calculated, the VCG was evaluated for cycles, and if none, the longest path was determined. He also calculated the number of nets, terminal density, and average number of terminals per net. These parameters were used to select one of two routing techniques. The first, router-1, did not allow doglegs, and therefore would not handle a cyclic VCG. Router-2 broke down multi-nets into two-terminal sub-nets and would handle most cyclic VCG cases:

The nets are routed, and then a mutation phase modifies the assignments looking for vertical constraint violations. The mutation continues until no progress is made, at which time the output is created and the entire process terminates. The algorithm was most affected by constraints in the HCG and VCG, causing its solutions to vary from problem to problem. Another limitation was that the size of the problem set greatly affected the performance of the algorithm and the amount of memory needed for the computations.

This concludes our overview of the two-layer channel routing approaches. We will now look at the three-layer methods that have been presented so far.

3.2 Three-Layer Approaches

Three-layer channel routing algorithms have begun to surface, due to the increasing availability of three metal layers in chip design. Sherwani points out that the Motorola 2900ETL macrocell array, the Dec Alpha chip, and Intel's 486 and original Pentium chips were all designed using a three-layer design [SN95].

Most all of the three-layer approaches are extensions of two-layer methods. We will look at an extension of the net merge algorithm by Chen and Liu [CL84], another extension of the same by Cong, Wong, and Liu [CW87], and finally a hybrid HVH-VHV router by Ptchumani and Zhang [PZ87].

3.2.1 The Extended Net Merge and CWL Channel Routers

The first of the three-layer approaches to be based on the net merge algorithm was presented by Y. Chen and M. Liu [CL84]. Theirs was an extension of the YK algorithm of Yoshimura and Kuh [YK82]. Their process was to perform two types of merging. The first was serial merging, where two nets have neither horizontal nor vertical constraints, and so can be assigned to the same track and layer. The second was parallel merging, in which there exists a horizontal constraint between two nets, but not a vertical constraint, so the nets can be put on the same track in different layers. The process of creating zones is still incorporated.

Cong, Wong, and Liu [CW87] offered another extension of the YK algorithm, which can be referred to as the CWL algorithm. Their process was very similar to Yoshimura and Kuh [YK82] in that it merges nets into composite nets based on not having horizontal constraints between the nets based on their zones. They further merged composite nets into super-composite nets where their only constraint was horizontal. This allowed the super-composite nets to be assigned to the same track on different layers.

Evaluating the algorithms, they both suffered from less than optimal results if the net pairings were less than optimal. CWL gave solutions with extra tracks due to adjacent vias in nets being merged.

3.2.2 The Hybrid HVH-VHV Router

As a final three-layer example, we look at the channel router developed by V. Pitchumani and Q. Zhang [PZ87]. This process partitions the set of nets into two groups. One group is made up of the nets that lend themselves to an HVH layering scheme, and the other group is made up of those better suited in a VHV layering scheme. A transition track is used to complete the connections between partitions.

The hybrid algorithm does not allow doglegs, and Sherwani indicates that this algorithm is not for all situations [SN95]. He does say that it performs best when the problem set is entirely an HVH or VHV layering group.

3.3 Summary

With this analysis of many channel routing algorithms, we see that the problem has been addressed from many different points of view. There have been many two-layer solutions offered, each providing a solution to a particular facet of the problem. Several three-layer algorithms have been developed from two-layer versions to improve upon their results. It is clear that no single algorithm is best for all situations.

We will now present the basis for our multi-layer channel router. First an understanding of the genetic algorithm and how it applies to solving computationally intensive problems is in order. Afterward, we will present the algorithm for the multi-layer channel router.

Chapter 4

The Genetic Algorithm

4.1 An Overview

The Genetic Algorithm is patterned after the evolutionary process of ‘survival of the fittest’. It mimics the way nature will foster succeeding generations based on their ability to change to better fit in with and make the most of their environment. In nature, a bird with a more sharply pointed bill can better extract insects from a fallen log, and so flourishes. In a similar way, the genetic algorithm creates temporary solutions to a problem, and then uses those solutions to create new solutions, keeping the better solutions and throwing away those that fall further from the optimal.

We first heard of Genetic Algorithms when J. Holland introduced them in 1975 [HJ75]. They were developed to help evaluate the adaptive processes of natural systems. Today, we find them to be a valuable tool in evaluating processes that either have a large search space, or are large in terms of calculations required for an optimal answer. To better define the genetic algorithm’s properties, we will use a partitioning problem as an example.

The partitioning problem comes from an earlier phase in the VLSI microchip design. When initially laying out the locations for the circuits, each circuit is defined by a

minimum area it must occupy. These circuits are placed together, but must be partitioned into groups for layout purposes. In our example, we will look at how the genetic algorithm can be applied to evaluate the partitioning problem, so that two partitions are created that require nearly the same surface area. This problem can be evaluated optimally for rather small quantities of circuits, but quickly becomes difficult, if not impossible, to analyze by optimal methods as the problem set increases in size, therefore the genetic algorithm is well suited to provide a near optimal solution. Let us first understand the Genetic Algorithm process.

4.2 How the Genetic Algorithm Works

There are several steps involved in the Genetic Algorithm process. The problem must first be properly defined, as well as identifying the requirements the solution must meet. Once defined, a representation of the solution can be determined. The representation should take the form of a string, so that it can be easily manipulated, and is sometimes referred to as the chromosome. A cost must also be defined so that the correctness of a solution can be measured.

After the problem is well defined, initial solutions are derived. These first solutions, called the parent solutions, are then used as the basis to derive additional solutions. These are referred to as the child solutions or offspring. The child solutions are then evaluated by the costing criteria for their ability to solve the problem. The best child

```

input the problem set
analyze input to formulate solution content
create parent solutions, cost them, and store best
while not done do {
  for each parent {
    mutate to create a child solution
    if fit enough, keep it (cost)
  }
  for all children and parents {
    store the best so far
    choose new parents
  }
}until done
report results

```

Figure 4.1 The Genetic Algorithm

solutions are kept and then used as the bases to derive more child solutions. The process continues in this fashion, until it fails to show progress. A limiting variable controls the process by increasing with failures until it reaches a maximum value, after which the results are reported. Figure 4.1 shows the pseudocode for the genetic algorithm. We will now look at the key portions of algorithm to analyze its inner processes.

4.3 The Mutation Process

The derivation of a child solution is accomplished by a type of mutation. Mutation can be accomplished by several means. In one case, a parent solution can be modified by switching either one or two parts of the solution to create a new solution. As an example, figure 4.2 shows a parent string and the resulting child string after a mutation.

Example 1	
Parent:	1 0 0 1 1 1 0 0 1 0 1 1 1 0 1 1 0 0 1
Child:	1 0 0 0 1 1 0 0 1 0 1 1 1 0 1 1 0 0 1
Example 2	
Parent:	4 5 3 6 1 2 5 7 9 8 9 0 4 5 3 2 1 2 7
Child:	4 5 3 9 1 2 5 7 9 8 6 0 4 5 3 2 1 2 7

Figure 4.2 Mutation by Switching

The first example would be a possible mutation of the partitioning problem. In that example, the fourth entry has been switched, so that the fourth circuit was changed from one partition to the other. Generally this is performed one unit at a time and then the new solution is evaluated and another switch may be performed. In the second example, the fourth and eleventh positions have been switched. This could apply to a multi-partition problem, or our channel routing problem, in which two nets' placements are switched.

Example 1	
Parent1:	1 0 0 0 1 1 0 0 1 0 1 1 1 0 1 1 0 0 1
Parent2:	0 1 0 0 1 0 1 1 0 1 0 1 0 1 0 0 1 0 0
Child1:	1 0 0 0 1 1 0 1 0 1 0 1 0 1 0 0 1 0 0
Child2:	1 0 0 0 1 1 0 0 1 0 1 1 1 0 1 1 0 0 1
Example 2	
Parent1:	4 5 3 6 1 2 5 7 9 8 9 0 4 5 3 2 1 2 7
Parent2:	2 4 8 7 1 3 4 9 5 6 1 3 2 7 2 7 3 5 9
Child1:	4 5 3 6 1 2 5 7 9 8 9 0 2 7 2 7 3 5 9
Child2:	2 4 8 7 1 3 4 9 5 6 1 3 4 5 3 2 1 2 7

Figure 4.3 Mutation by Crossover

Another process, the crossover, involves using two parent solutions to create new child solutions, by taking a portion of one parent and appending to it the complimentary part from the other parent. In figure 4.3, we see two examples of this. For the partitioning problem in example 1, the parents are crossed over starting with the eighth circuit. In the second example, the crossover occurs with the twelfth position.

Still another form of mutation involves reversing the string or chromosome, end for end. This type is referred to as inversion, and can be performed on a part or on the full length of the string. In figure 4.4, example 1, for the partitioning problem, demonstrates the sixth through the thirteenth positions of the string being inverted, while in example 2, the entire string is inverted. Nearly any change can be useful as a means of deriving a new and different solution.

Example 1	
Parent:	1 0 0 1 1 1 0 0 1 0 1 1 1 0 1 1 0 0 1
Child:	1 0 0 1 1 1 1 1 1 0 1 0 0 1 0 1 1 0 0 1
Example 2	
Parent:	4 5 3 6 1 2 5 7 9 8 9 0 4 5 3 2 1 2 7
Child:	7 2 1 2 3 5 4 0 9 8 9 7 5 2 1 6 3 5 4

Figure 4.4 Mutation by Inversion

Mutation offers several benefits. It should be a randomly applied change so that its result is not necessarily predetermined. This gives us the opportunity to escape local

minimums, and try to reach an optimal solution for the problem. Using more than one type of mutation offers increased opportunity for solutions to approach an optimal solution. The value of that solution, to the common goal, is the next issue of investigation.

4.4 The Selection Process

After the child solutions are derived, they are evaluated for their fitness, or their ability to solve the problem. This evaluation is very specific to each problem, but usually involves a cost for the solution's suitability in solving that problem. As an example, in the partitioning problem, the total area of each partition can be calculated, and their difference can be used as a cost. As there would be connections between the partitions, another possible costing factor could be the number of connections that travel between the partitions. The overall partition size could have a maximum value, so that a third partition may be required. All of these options can be used together, with a weighting factor for each, to promote the more important criteria. Having been defined, we can now use this costing procedure to evaluate both the parents and children for their ability to solve the problem.

The costs for all children and parents are compared, and the lowest cost solution is stored. Several of the solutions are then chosen as the parents for succeeding generations. Multiple policies for retaining a solution as a parent can be used. The best can be saved,

or a random sampling can be kept. The important element here is to keep a varied sampling, so that the succeeding generations have some diversity. This offers the possibility to find better solutions and avoid falling into local minimums.

4.5 Global Control Parameters

There is also a global component to the evaluation process. As each new solution is created, the evaluation criteria are also changing. There are several control variables involved in the process that should be identified at this time. The first is the variable p , which controls the selection of child solutions. When a new solution is generated, its cost is calculated and is compared with its parent. The positive or negative gain is compared with this control variable p to see if it is worthy of being retained. The value of p can change depending on the success or failure of child solutions created. If the child solution shows a positive gain, p is kept at its origination value p_0 . If the succeeding solutions do not show a positive gain, the value of p can decrease to allow solutions to be retained that do not show a positive gain. This again allows for diversity so that local minimums can be overcome.

The other variable used to control the global process is defined as r . The value of r begins at zero, and increases as the newly created generations fail to improve in their suitability to solve the problem. If the new solution is an improvement, the value of r is reduced by a user defined upper bound value, R , for the next child creation. Thus, the

variable, r , increases in value if the process stagnates in its search for a better solution. When the value of r reaches the upper bound, R , the process stops generating new solutions. After the evolution process has terminated, the stored best solution is output.

Another factor that influences the process concerns the manner in which the parent and child solutions are created. The number of parents created and saved, as well as the number of children generated can also affect the quality of the overall outcome. This can also depend on the size of each generation, as a larger generation size will require more memory and tend to reduce the number of parent and child solutions that can be stored. Generally speaking, the greater the number of parents and children that are created, the greater the possibility of reaching an optimal solution quickly and efficiently.

We will now look at the multi-layer channel router problem and how the Genetic Algorithm can be applied to provide an efficient and effective solution. The basis for our using the algorithm is first identified, as well as the way the algorithm is applied. An analysis of the specific data structures used is followed by the details of the processes involved in the algorithm, as they apply to the channel router.

Chapter 5

The Multi-layer Channel Router

Because of the complexity of the channel routing problem, a genetic algorithm seems to be an excellent heuristic to find a near optimal solution. By its diversity, it can more easily overcome local minimums. With technology rapidly changing, improvements in the manufacturing process have allowed for an additional layer of circuitry within the chip, to reduce channel width, and help resolve constraints. We can make use of these two ideas as the basis for a new multi-layer channel routing process.

5.1 The Basis for the Router Process

The channel routing process begins by looking at the problem itself, to find the most efficient method of connecting similarly numbered pins through a channel, where the pins or terminals are on the top and bottom of the channel. The solution should offer a design with as narrow a channel as feasible, and without conflicts in the circuitry layout. The input for this process is a text file, with the number of terminals per side of the channel as its first entry, followed by a list of the top terminal connection numbers and then the bottom terminal connection numbers. This identifies the terminals that need to be connected and their order along the channel. We want the ability to run the process in a two-layer mode or a three-layer mode.

The process takes, as part of its command line, the input file name. The input file is evaluated to determine the number of nets required, along with their beginning and ending points in the channel. A user defined separation variable determines how the nets are assigned. An array of structures, as shown in figure 5.1, stores this information, as well as the net number, as defined by the terminal numbering sequence from the input file. The control parameters are defined by the user, following which the overall characteristics of the problem set are analyzed to determine the best way of approaching the problem. The analysis also involves setting the expectations of a reasonable solution. The greater of the maximum clique of the horizontal constraint graph and the longest path length of the vertical constraint graph can be used as the lower bound for the number of tracks in an optimal solution. The decision as to whether to use two or three layers is made, and if

```

struct net {
    int net_num;
    int strt_pin;
    int end_pin;
};

struct net netlist[ ];

```

Figure 5.1 Structure for Basic Net Storage

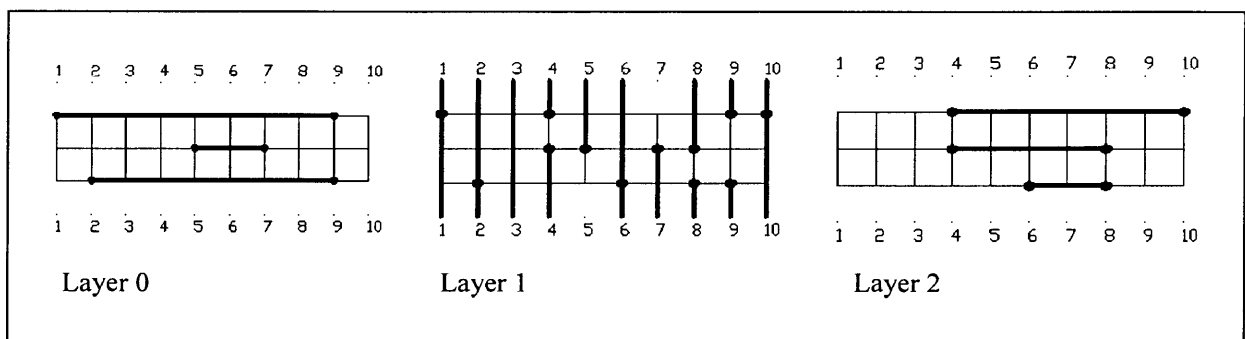


Figure 5.2 HVH Layering Model

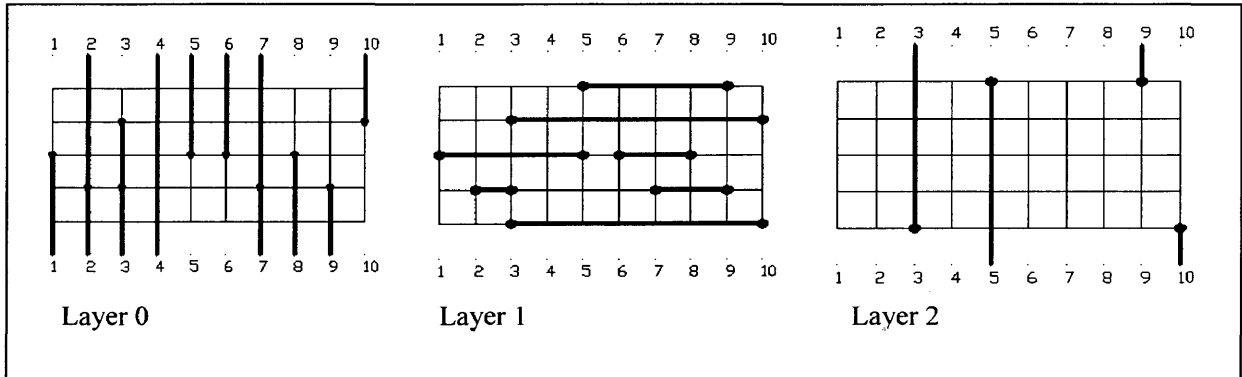


Figure 5.3 VHV Layering Model

three layers are chosen, it is determined whether to use the HVH or VHV layering scheme. Figures 5.2 and 5.3 give examples of channel routing using the HVH and VHV layering schemes, respectively.

Following this analysis phase, the initial parent solutions are derived. The parent solutions are sent through a compaction process that tries to minimize the number of tracks necessary, as well as expanding when conflicts arise between two nets vying for the same space. The parents are then evaluated using a costing routine to determine the best solution so far, and that solution is stored for future evaluation. Each parent is then used as the basis for generating several children via a mutation process. As each child is produced, it is run through the compaction process, and then evaluated for its fitness. Based on the evaluations of both the children and parents, a portion of the group is chosen to be the parents for the next generation. This process continues until an exit control variable reaches a user defined maximum value. All through the process, the

child solutions are compared against the best solution, and if an improvement is found, the new solution is saved.

After the genetic portion of the algorithm has concluded, the reporting phase outputs the best solution to a file that can be input into AutoCAD[®], a computer-aided design package, to visually display the channel layout. The display can then be analyzed for its accuracy and desirability in a three-dimensional manner on screen.

Now that we have the basic flow of the process, we will look more closely at the actual components of each process to get a better understanding of what takes place. An understanding of the data structures involved is presented first.

5.2 Data Structures Used

There are a number of arrays used in the process, several to store the net information and layout, while others are used during the analysis phase. Two arrays, *t_in* and *b_in* store the top and bottom terminal lists, respectively. Another two arrays, *left_pt_list* and *right_pt_list*, store the nets by net number based on their left and right points in the terminal list. *Netlist* is an array of structures, see Figure 5.1 earlier, that stores the nets' base information of net number, starting point and ending point. *Sol* is another array of structures, this one storing the various solutions created by the process. This array, as shown in Figure 5.4, is two-dimensional, the first index identifying the solution number,

and the second specifying which net in *netlist* it addresses. The structure within this array defines what layer and track number are to be used for a particular net. The small storage space required for these two arrays, allows us to handle larger problem sets without memory overflows. Finally, the array, *Solcost*, stores the costs involved with each solution generated.

```

struct layout {
    int track;
    int layer;
};

struct layout sol[ ];

```

Figure 5.4 Structure for Solution Storage

To handle the analysis of the problem set, *vcg* is a two-dimensional array used to calculate the vertical constraint graph. *Color* is an array used to perform depth first search analysis, while *saved* stores the solution numbers that will be used as parents for the next generation. Within several of the subroutines, the array *lyr_ck* is used to track net placement so that conflicts in assigned location can be identified.

There are also a number of variables to be identified, that handle special storage needs. The control variables, *sep*, *p*, *p₀*, *r*, and *bigr* are used to adjust the performance of the algorithm. *Sep* controls the number of terminals a net can have. A value of two creates two terminal nets, while a value of four would create nets with up to four terminals. The variable, *p*, controls the selection of child generations, with *p₀* storing the base value. The variable, *r*, controls the termination of the genetic process. The value of *r* increases

as the process doesn't progress, or is reduced by *bigr*, when it does generate beneficial solutions. As *r* increases, *bigr* also serves as a termination value for the value of *r*.

Additionally, there are several variables used to track some of the settings used within the algorithm. The value of the highest numbered net is stored in *top_net*, *total_nets* stores the total number of nets created, *total_seg* stores the number of non-trivial nets, and *total_term* stores the number of terminals contained in the problem set. The maximum clique of the horizontal constraint graph is calculated and stored in *max_clique*, *longestpath* stores the length of the longest path in the vertical constraint graph, *layers* stores the number of layers in use, and *vhv* is set to true if the vertical constraint graph signals that a cycle exists. This would require two vertical layers to resolve the conflict. Several other less significant variables handle loop counting, true / false values, and string values used for file access.

5.3 Process Elaboration

We will now look more closely at several of the modules involved in the genetic algorithm for multi-layer channel routing to get a better understanding of their functions. An examination of the sub-routines that handle creating nets, finding the maximum clique, vertical constraints, and longest path will be performed. We will also look at the routines that handle creating the parent solutions, costing and compacting them, mutating parent solutions into child solutions, the update routine that modifies *p*, which determines

whether to keep possible solutions, and finally the output function that creates an output file to create a graphic representation using AutoCAD® Release 2000.

5.3.1 Creating Nets

The process of creating nets, from the top and bottom terminal lists, makes use of the variable *sep*. The terminal lists are evaluated from left to right. The process searches for the terminal numbers corresponding to the first net, and then determines where net-1 begins and ends. Figure 5.5 shows an overview of the algorithm.

```

for i = 1 to the number of terminals do {
  set the count for this net to zero
  for each terminal do {
    if the net # for top or bottom matches i then {
      if count = 0 then store this as a starting point
      else { store the point as an end point
        add 1 to count
        if count = the separation value then
          use current point as start of new net
      }
    }
  }
}

```

Figure 5.5 Net Creation

The variable *sep* is used to specify how many terminals can be included in each net. As an example, if net X contains five terminals throughout the length of the channel, and if

sep has a value of two, then four nets would be created, between the first and second of its terminals, between the second and third, etcetera. Under the same conditions, if *sep* had a value of three, there would be two nets created between its terminals one and three, and between three and five. If, as a final example, *sep* had a value of four, then there would still be two nets created, but this instance would have a net from its first to fourth terminal and a second net from the fourth to fifth.

The process continues by looking for the terminals marked as having net number two attached, to create nets as defined by the value of *sep*, until the maximum number of terminals has been evaluated.

5.3.2 Finding the Maximum Clique

The process of finding the maximum clique is taken from the work of Jingsen Zheng [ZJ98]. On pages 39 and 40, he describes a process that calculates the maximum clique of the problem set. We first want to identify the intervals that each net will require. Each net is evaluated to retrieve its starting and ending points, and the net number is recorded in the *left_pt_list* and *right_pt_list* arrays in the locations of these points, respectively. We can then traverse the arrays from left to right, incrementing the value of the variable *clique* for each net that is encountered in the left point list and decrementing each time a net number is encountered in the right point list. The maximum value of *clique* is stored

```

for each net segment do {
  place its net # in the left point list at its starting point
  place its net # in the right point list at its ending point
}
for each entry in the list do {
  if the left point list has a net starting then
    add 1 to clique
  if the right point list has a net ending then
    subtract 1 from clique
}
return the maximum clique value

```

Figure 5.6 Maximum Clique

and is returned as the value of the maximum clique upon termination of the routine. Figure 5.6 shows the algorithm for finding the maximum clique.

5.3.3 Evaluating the Vertical Constraints, Cycles, and the Longest Path

An evaluation of vertical constraints in the problem set offers several valuable pieces of information. A graph of the vertical constraints, or VCG, can be created, which can then be used to search for cycles in the net design, and can also be used to determine the longest path of the VCG. This is particularly helpful in defining a lower bound on the number of tracks when a single layer is used for the horizontal tracks.

The vertical constraint graph is stored in an $n \times n$ array called *v_{cg}*, where n is the number of terminals in the problem set. In figure 5.7, we see the algorithm used to create the

```

for each terminal do {
  if the top net # is zero or the bottom net # is zero or
  the top and bottom net #s are the same then
    continue
  else vcg[top net #][bottom net #] = 1
}

```

Figure 5.7 Creating the Vertical Constraint Graph

VCG. The initial creation of the VCG is performed by evaluating the top and bottom terminal lists together. The terminal lists are compared, starting with their first pins respectively and moving toward the other end. The evaluation consists of checking to see if at least one of the pins is labeled with a net number of zero, indicating that there is no net on that pin, or that the pins have the same net number. If neither of these conditions is true, then *v*cg is marked with the number one (1) to indicate a vertical constraint in the position using the net numbers from the top list as the first index, and the net number from the bottom list as the second index.

After the VCG has been created, it can be checked for cycles, which would indicate that there are problems placing nets using a single vertical layer. The sub-routine, *cycle_ck*, is used to test for cycles, and its algorithm is shown in figure 5.8. It makes use of the array, *color*, to determine if it has visited a node of the VCG more than once. Each node is checked for its color, and if initially white (0), it is colored (1) and a search of other nodes adjacent to it is performed, using the recursive subroutine, *srch*. Adjacent nodes are likewise checked, thus creating a depth first search. If, during the search, a colored

```

for each node in the VCG do {
  if its node is white then search for adjacencies
}
if a cycle is found then return true

          Search

for each node in the VCG do {
  if its node is white then search for adjacencies
  if a cycle is found then return true
}

```

Figure 5.8 Cycle Check

node is found, the routine returns a positive response for a cycle. This sets the variable, *vhv* to a value of one (1), indicating that the vertical-horizontal-vertical (VHV) layering is favored to overcome the cycle.

The longest path can also be found by analyzing the VCG. The sub-routine, *longest_path* uses a modified depth first search to calculate the longest path between nodes. It processes each node, visiting all adjacent nodes recursively, and counting each visit. The longest path is stored as the greatest number of visits to adjacent nodes. Figure 5.9 lists the algorithm used.


```

for each node in the VCG do {
  perform a depth first search for adjacent nodes
}
return the longest path

Depth First Search

for each node in the VCG do {
  add 1 to the path
  perform a depth first search for adjacent nodes
  return the path length
}

```

Figure 5.9 Longest Path

5.3.4 Creating Parent Solutions

After the problem set has been analyzed to find its specific characteristics, the process proceeds by creating the initial parent solutions. The sub-routine, `mk_parent`, who's algorithm is shown in figure 5.10, performs the function of evaluating the list of nets to place them in tracks, based on the settings of `nhv`, the number of layers desired, and the locations of the terminals in the net list. The process initially works with only the non-trivial nets, and then places the trivial nets afterward. One of the non-trivial nets is randomly selected, and placed in the first available track. If the net has both of its terminals in the top list, it is placed in the track numbered for the maximum clique. This keeps it from interfering with nets that have both of their terminals on the bottom, or one in each terminal list.

```

Randomly choose a non-trivial net
for each non-trivial net do {
  if both terminals are from top then
    start at top track work down, use ck_space to find space
  else start at bottom track work up, use ck_space to find space
  store the track and layer position
  mark lyr_ck with the net # for the space used
}
for each trivial net do {
  search for a track & layer with the same net number
  store the track and layer position
}
store maximum track number

```

Figure 5.10 Making the Parent Solutions

The procedure, `mk_parent`, uses the sub-routine, `ck_space`, shown in figure 5.11, to verify that the length of a chosen track is available for the placement of the net currently being evaluated. The array, `lyr_ck`, is used to coordinate the placement of the nets within the

```

for the track and layer do {
  check from left point to right point for space
}
if the space is free then {
  mark the vertical components as used
  return successful find
} else return failure testing that space

```

Figure 5.11 Checking For Free Space

channel. The array is a three dimensional array to hold the locations of the nets placed into the channel based on their layer, track number, and location along the channel. The purpose of *ck_space* is to make sure that the net currently evaluated has room in *lyr_ck*, so that there isn't a conflict with another net in that vicinity. In a similar fashion, the other non-trivial nets are placed in a greedy fashion, by finding the first available space.

Finally, the trivial nets are placed by searching for a layer that already contains a non-trivial net with the same number. If no layer exists, the net is placed on the first layer. There is not a track required for the trivial nets, since their terminals are located at the same pin number in opposing terminal lists.

5.3.5 Improving the Solution by Compaction

The sub-routine, *compact*, is used to evaluate the placement of nets into the layer and track arrangement. The process strives to minimize the number of necessary tracks as well as expanding solutions with conflicts in the track layout to try to resolve those conflicts. The process also uses the array, *lyr_ck*, to analyze the net layout, looking for empty space and overlaps.

After clearing the array, the tracks and verticals are placed into the array. As the nets are read, the space is checked to make sure it is available. If it is not, the net is noted for relocation, and the next net is evaluated. The list of nets with overlaps is then sent to a

```

for each net do {
  if its placement falls on another, then mark for expansion
  else store its location
}
expand all nets without placements
for each non-trivial net do {
  check up to present location for a more compact location
  if found, then move it
}
check trivial nets for placement with same net # non-trivial nets
calculate maximum tracks used

```

Figure 5.12 The Compaction Algorithm

sub-routine, expand, which tries to place them in another track or layer where space is available. Once the nets have all been recorded in the array, the nets are evaluated again to see if space for any of them exists on a lower track and/or layer. If it can be moved, the new location is recorded in the solution, and the array is updated to free its old placement and fill the new one. Finally, the maximum number of tracks required by the solution is stored to be referenced later. Figures 5.12 and 5.13 list the algorithms used to do the compaction and expansion, respectively.

```

for each net not placed do {
  use ck_space to find a track to locate the net
  store the new track and layer
}

```

Figure 5.13 The Expansion Algorithm

5.3.6 Expressing the Value of the Solution

The solutions must be evaluated for their fitness. The sub-routine, *cost*, returns a value describing how the solution organizes the tracks and vias, and how it minimizes the number of each required, as well as indicating whether there are overlaps in tracks or verticals between nets. This process also uses *lyr_ck* to place the nets, so that their locations can be checked for overlaps. For each net, the track used is first evaluated. If the area between the left point and right point has been assigned to another net, a track error is logged.

Next the verticals are checked by comparing the net number with the left point terminal values on the top and bottom. If the top matches, the vertical from the track to the maximum track value is checked for free space to allow for the layout of the vertical. If the bottom matches, the vertical space is checked from the current track down to track zero. In a similar fashion, the right point of the net is tested for vertical space. All the vertical tests are performed on the vertical layer(s) so that they do not affect the track layer locations.

The vias are also counted in the sub-routine, *via_count*, to analyze the number of vias required to complete the solution. Since the number of vias affects the quality of the solution, *via_count* is included in the analysis of the cost of that solution. In *via_count*, *lyr_ck* is also used to mark their locations. As the nets are evaluated, the vias required

are marked in the array, and then a count is made of the number of vias required. This number is later used in the sub-routine, cost, to help determine the quality of the solution.

After logging all track and via overlaps, or violations, the sub-routine calculates a cost using the following function:

$$\text{sol_cost} = \text{max_track} + (4 * \text{trk_err}) + (6 * \text{via_err}) + (2 * \text{vias})$$

where sol_cost is the value returned as the cost, max_track is the maximum number of tracks required for this solution, trk_err is the number of track violations found, via_err is the number of vertical violations found, and vias is the number of vias required for the solution. Sol_cost is returned to the calling routine. Figure 5.14 shows the algorithm involved in costing the solutions, and figure 5.15 shows the algorithm for counting vias.

```
for each net do {  
  check for free space on the track assigned  
  if failure, then add one to trk_err  
  check for free space in each via location if necessary  
  if failure, then add one to via_err  
}  
cost = maximum tracks + 4 * track errors + 6 * via errors + 2 * vias  
return cost
```

Figure 5.14 The Costing Algorithm

```
for each net do {  
  if the net number matches a top or bottom terminal then {  
    mark lyr_ck with a via needed  
  }  
}  
count vias required  
return the value
```

Figure 5.15 Counting Vias

5.3.7 The Mutation Phase

The mutation of the parent solutions creates a new generation of child solutions. The sub-routine, mutate, performs this task. Earlier, we described several mutation processes. The crossover method was discarded due to the complexity of the problem. If too many nets were moved, the results would obviously be much worse than the parent offered. Inversion was also excluded since inverting one layer with another would not offer an improvement, and reversing the net locations would offer a very high cost because of numerous track and via violations. The switch method was seen as the most effective means of mutation.

The parent solution is first copied to a new solution location for manipulation. Two non-trivial nets are chosen at random to have their track and layer locations switched. The child solution's cost is calculated and then subtracted from the parent solution cost to find a positive or negative gain. This gain is compared with a random value between zero and the value of the control parameter, p . For the current problem, the value of p is negative to allow more solutions with less successful costs to be included in the parent solutions. If the gain comparison is successful, the new solution is retained, and if not, the old layer and track for the two nets are reset. The process repeats up to 20 times, looking for a viable solution.

The sub-routine, update, is also involved in the mutation process. Update compares the parent cost with the child cost, and if the child cost indicates a poorer solution, the value

```

repeat {
  pick two non-trivial nets at random
  switch the layer and track between them
  calculate gain as the cost of the parent – cost of the new child
  if gain > a random value between 0 and  $p$  then return
} up to 20 times

```

Update Function

```

if old cost < new cost then  $p = p - 1$ 
else  $p = p_0$ 

```

Figure 5.16 Performing Mutation

of p is reduced to allow less fit solutions the opportunity to be used as parents, to possibly create better child solutions in the future. If the solution is better, then p is reset to its initialization value of p_0 . Figure 5.16 shows the algorithms used for mutation and update.

5.3.8 Choosing New Parents

After the mutation phase has modified the parents to create children, the process must make a selection of the population to determine which to use as parents for succeeding generations. The process can be run in two modes, elite and non-elite. In the elite mode, the five solutions with the lowest cost are chosen to be parents. In the non-elite mode, a random selection of five solutions are chosen as the parents for the next generation.

The process is handled in two steps. The first step is performed within the main routine. When the user assigns the initial control parameters, the choice is made to run in elite mode or non-elite mode. After the mutations are complete, the cost of each solution is compared and the best five are chosen. Those five are then processed by the sub-routine, `sort_them`. In `sort_them`, the solutions are either sorted to be used as the next parents (elite mode), or the parents are chosen again on a random basis and then sorted to be used as future parents (non-elite mode). This random choice allows more diversity in the generations to overcome the local minimums encountered. The algorithm for `sort_them` is shown in figure 5.17.

```
the sub-routine is passed the top 5 solutions
randomly select between 0 and 2
if 0 then choose 5 solutions randomly
sort the five solutions and store them as parents
```

Figure 5.17 The Sorting Algorithm

5.3.9 The Reporting Mechanism

The final process to examine is the reporting mechanism. The sub-routine, `output_sol`, is used to create a script file that can be used by AutoCAD® to create a graphical representation of the solution. The routine takes as its parameters, the solution number and the name to be applied to the output file. The filename is created by concatenating the name and solution number, along with the extension '.scr', and the file is opened for write purposes.

The terminals for the solution are calculated by finding the maximum track number for the solution, and the length of the channel involved. The file is appended with the command structure to create vertical lines representing the terminal pins. These are placed below the track zero level and above the maximum track level, spaced one unit apart. These will be placed on a layer called 'terminals' within AutoCAD®. The information to print terminal numbers by each pin is then written to the file, followed by a grid along the bottom layer to help visualize the track layout.

The nets are then written to file via a loop that processes the nets by the layer onto which they have been designated. On the first pass, the nets for layer zero are recorded and on the second pass, if necessary, the layer 2 nets are recorded. As each net is evaluated, the command structure to place a line from the left point to the right point of the net is created. The information to draw the vertical connections to the pins is also stored, as well as lines to draw the vias connecting the tracks on their layer(s) to the verticals on their layer(s). Each part of the net has its own layer on which it will be created. There is a separate layer for the vias, and a separate layer for each of the track layers, if two are used, in an HVH model, or two for the vertical layers in a VHV model. A small circle is also drawn for the connections between the vias and the tracks or verticals. Figure 5.18 displays the algorithm for output_sol.

```
open output file
write commands to place terminals and number them
write commands to draw in the grid
for each net do {
  write commands to draw the track used
  write commands to draw the verticals to attach to the pins
  write commands to draw the vias and place circles at the ends
}
```

Figure 5.18 The Output Algorithm

5.4 The Main Process

The main process pulls all the individual parts together, processing the problem set to derive a solution that is near optimal. The main process takes as input the name of the file containing the input information. After verifying that an input file was defined, the user is asked to input the control parameters. The values of *sep*, *p₀*, and *bigr* are requested, after which *p* is set to the value of *p₀*, and *r* is initialized to zero. The decision to run in elite or non-elite mode is made. The sub-routine, reader, is executed to open the input file and populate the top and bottom terminal list arrays, *t_in* and *b_in*. These arrays are evaluated to find the maximum net number used.

The subroutine, create_nets is called to create the array of structures that holds the base information for each net. The analysis phase begins as the maximum clique is calculated, followed by the initialization of the vertical constraints graph. Cycle_ck is then executed to test for cyclic references in the terminal structure. If a cycle is discovered, the user is informed of this and the value of the variable, *vhv*, is set to one to identify the need for two vertical layers. The user is then asked whether the output should use two or three layers. Finally, the longest path is calculated. These parameters are used to determine the type of solutions created.

It is now appropriate to begin the genetic process, and so the initial parent solutions are created. The program has predefined, the number of parents to be used and the number

of children to be created from them. This causes `mk_parent` to create five parent solutions. Each has its cost calculated, and each is written out to a script file via the `output_sol` sub-routine. The parent solutions are compacted, and then the solution with the best cost is determined, and saved as the best solution so far.

Once the parents have been created and evaluated, four child solutions are created for each parent by the mutation sub-routine. As each is created, it is compacted, and its cost is calculated. Update is run to determine if progress is being made, and p is reduced if necessary. Also each solution's cost is compared with the cost of the best so far, and if better, the best solution is replaced by the current solution. Following the creation of all the child solutions, the child and parent solutions are all evaluated to find the five solutions with the best cost. Those five are placed in the roles of parent for another generation of solutions. This process repeats, with the value of r changing according to the process' success or failure to improve on the solution, until r reaches the value of *bigr*. The best solution is then printed to screen and `output_sol` is called again to create the script file for that solution. This script file can then be used as input into AutoCAD® Release 2000, to give us a better visualization of the track and layer output.

5.5 Generating a Graphic Output

Once the script file has been created, it can be used as input into AutoCAD® Release 2000, to allow a three-dimensional look at the track layout. We have chosen to use

AutoCAD® Release 2000, for its ability to rotate the design in three dimensions in real time, to better analyze the layer and track layout. Beforehand, a template was created as the basis for the drawing that will be created by the script. This template has all the pre-defined characteristics of the drawing. There is a layer defined for each of the tracks and verticals in the design, as well as a layer for the vias. The separation of information onto separate layers allows us to color the tracks and verticals separately to aid in the visualization.

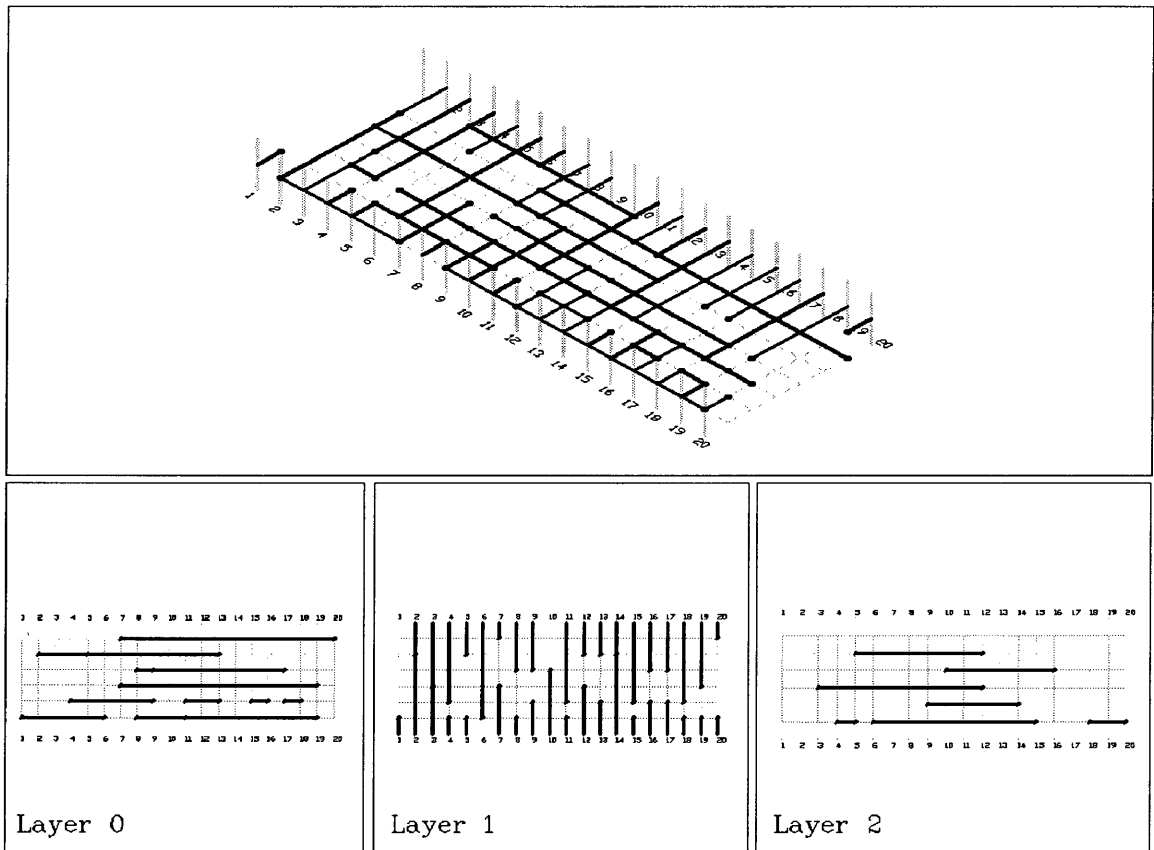


Figure 5.19 AutoCAD® Output Screen Showing Track and Layer Assignment

The script first creates the terminals on either side of the channel, and numbers them. A grid is then drawn in on the track and vertical lines. The nets are then drawn in. As each is drawn, the length of the net is checked to find terminals with that same net number, and when one is found, a vertical is drawn in to connect it. A via is added to connect the track and vertical layers to complete the connection. Figure 5.19 shows an example of the visualization as provided by AutoCAD[®] Release 2000.

Now that the process has been defined, we will turn our focus to the experimental results of the multi-layer channel router. We will analyze the performance of the algorithm based on the type and size of the problem set, and our choice of control parameters.

Chapter 6

Experimental Results

An evaluation of the experimental results of the multi-layer channel router reveals many of its characteristics, as well as its effectiveness. We will analyze the effect of each of the control parameters on the algorithm. The following parameters were analyzed for their effect:

- the separation value used to split multi-nets
- the value of the selection parameter, p_0
- the generation size
- the value of the termination variable, R
- the effect of executing the process in elite mode or non-elite mode.

We will also offer a comparison of running the algorithm in the two-layer and three-layer modes. The time complexity of the algorithm will also be evaluated, and general observations will be presented.

6.1 The Problem Sets Used

Two groups of problem sets were employed to give a variety of situations with which to evaluate the algorithm. Table 6.1 shows the characteristics of each problem set in the first group. Channel lengths of 30, 60, 80, and 100 were used to determine how the size

Problem Sets				
Input file	# Terminals	Max Clique	Cycles?	Longest Path
input30.txt	30	12	Yes	Undefined
input30a.txt	30	10	No	3
input60.txt	60	15	Yes	Undefined
input60a.txt	60	14	Yes	Undefined
input60b.txt	60	14	No	10
input80.txt	80	5	No	7
input100.txt	100	6	No	4

Table 6.1 Definition of Problem Sets

affects the solution generation. Approximately half of the problem sets contained a cycle in the vertical constraint graph. This forces the algorithm to address the layering differently when three layers are used. The VHV layering scheme is employed to eliminate vertical constraints. Those without a cycle have a value for their longest path identified. In the following sections, the problem sets are identified by the value of their channel length, and those that do not contain cycles are marked with an ‘N’ following their designation.

The second group consisted of six problem sets, each having a channel length of 40. They were tested to allow evaluation of a controlled channel length. One-half of the problem sets contained vertical constraint cycles. These sets were analyzed over the control variable, p_0 , and the separation variable, sep , to chart their solutions in three

Problem Sets					
Input file	# Terminals	# nets	Max Clique	Cycles?	Longest Path
input40.txt	40	36	6	Yes	Nil
input40a.txt	40	37	6	No	9
input40b.txt	40	35	7	No	9
input40c.txt	40	38	6	Yes	Nil
input40d.txt	40	39	8	Yes	Nil
input40e.txt	40	35	6	No	9

Table 6.2 Characteristics of 40 Terminal Problem Sets

different modes, 3-layer VHV, 3-layer HVH, and 2-layer. Their set characteristics are shown in Table 6.2.

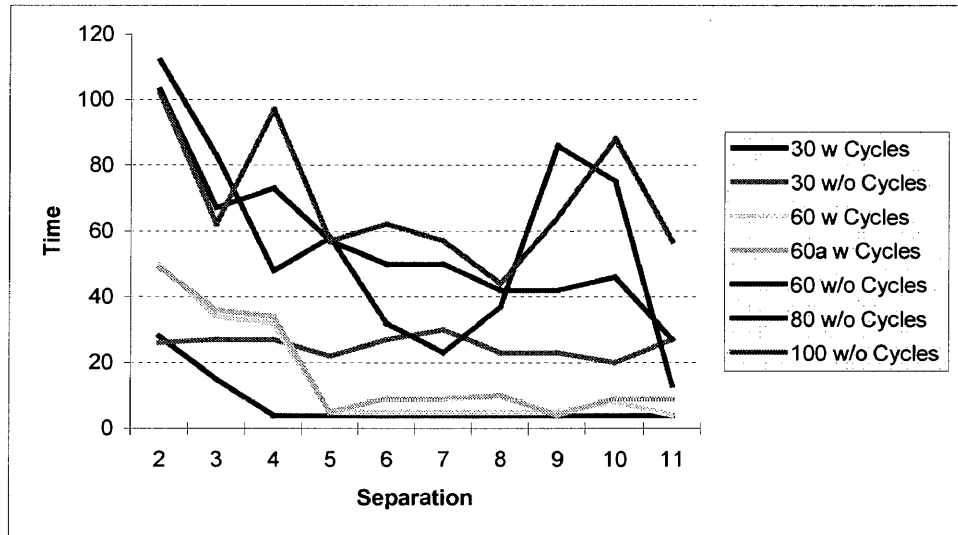


Figure 6.1 The Effect of Separation on Time

6.2 Separation Value

Initial testing was done to determine the affect of the separation variable, *sep*, on the solution. Figure 6.1 reflects the analysis time with respect to the value of *sep*, and figure 6.2 shows the change in cost, due to the change in the value of *sep*. The separation value was incremented from an initial value of 2 to a final value of 11. This upper value proved to be higher than necessary as each problem set had its own terminal separation value, based on the longest multi-net. Values above this point gave redundant results with most problem sets.

Analysis of the trials indicates that an increase in the separation variable causes a general decrease in both the time required for analysis and the cost. This is evidence of several

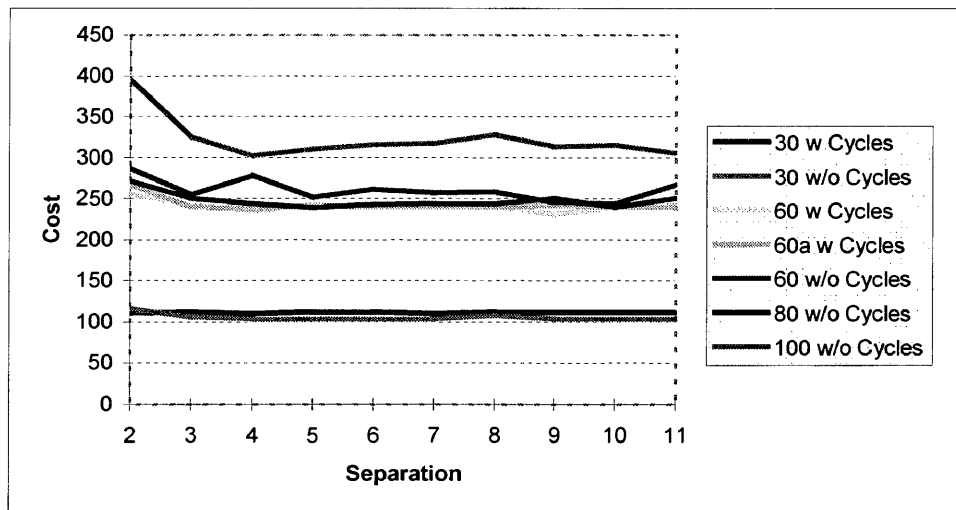
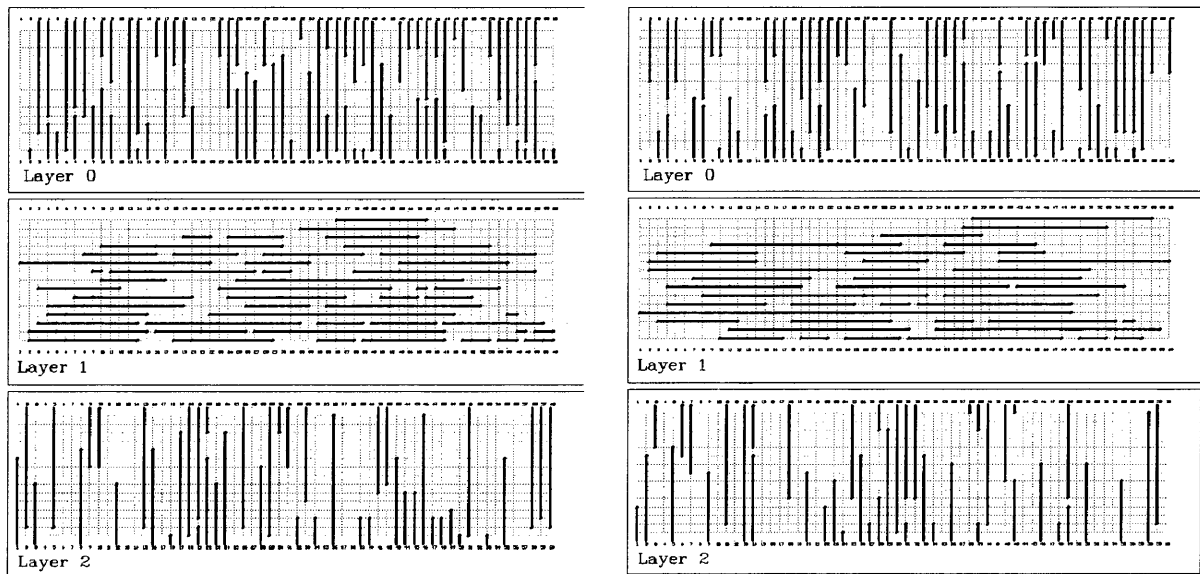


Figure 6.2 The Effect of Separation on Cost

factors changing. First, the number of nets is reduced by an increase in the separation value. This contributes to a lower time of analysis, due to shorter looping in the various processes. We can also consider the position that fewer nets allows less manipulation of those nets to find a better solution, so the termination value of R is reached more quickly. The cost of a solution is also reduced by an increase in the separation value. The reduction in nets also contributes to this, primarily because the number of vias is reduced.



Separation of 2
Figure 6.3 Outputs Based on Separation

Separation of 4

Fewer vias are required as a multi-net is forced to reside on fewer tracks, or on one track in the case of the terminal separation value. Therefore, we see fewer doglegs used when the multi-nets are treated as larger sub-nets or as a single net. Figure 6.3 shows a 60-terminal problem set, first modeled with a separation value of 2, and with a separation value of 4. Table 6.3 shows a comparison of the solutions. As the separation value

increases, the nets, vias, and cost are reduced.

The time required is also reduced from 49 to 36 seconds.

	Sep 2	Sep 4
Nets	77	46
Tracks	15	15
Vias	128	111
Cost	267	237

An evaluation of the problem sets having a channel depth of 40, reveals similar results.

Table 6.3 Comparison of Separations of 2 and 4

Again we compare the time and cost as the variable, *sep*, increases, but we also want to analyze the effect of the layering mode. Table 6.4 displays the results of the experimentation. As expected, the time required reduces as the separation value increases. The cost is not as obvious. We do see a similar reduction in cost as the

Time	Sep				HVH				2Lyr			
	VHV											
Set	2	3	4	5	2	3	4	5	2	3	4	5
input40.txt	18	14	4	3	84	46	16	26	55	47	42	19
input40a.txt	26	3	4	4	24	27	23	23	24	21	32	16
input40b.txt	19	11	4	4	36	27	27	31	23	32	16	24
input40c.txt	15	15	4	4	50	34	16	8	46	27	16	11
input40d.txt	30	11	4	4	50	51	16	29	59	39	23	32
input40e.txt	27	7	4	4	51	31	27	38	72	30	22	14
Cost	Sep				HVH				2Lyr			
	VHV											
Set	2	3	4	5	2	3	4	5	2	3	4	5
input40.txt	112	106	112	112	127	122	126	126	124	126	124	127
input40a.txt	116	112	112	112	140	123	121	120	134	119	117	118
input40b.txt	119	109	115	115	125	125	123	121	142	120	124	120
input40c.txt	112	106	110	110	123	119	120	129	124	128	121	129
input40d.txt	114	116	112	112	125	125	120	118	129	128	126	118
input40e.txt	116	112	110	110	143	128	127	123	140	122	124	130

Table 6.4 Time and Cost on 40 Terminal Sets with Separation

separation value changes, but not in all problem sets. Problem sets 40 and 40c do not exhibit this property, in regard to their costs. While both of these problem sets contains cycles, it does not indicate this as a contributory factor. Further research would be necessary to investigate this issue.

There was an exception seen in the times for some analysis periods after the terminal separation value was reached. This was evidenced as an unusually long or short time for a solution generation compared with other trials using similar parameters. This can be attributed to the nature of the genetic algorithm itself. A trial may reach a local minimum for a period, but before the terminal value of R is reached, a better solution is derived. This would reduce the value of r by R to continue analysis until the value of r reaches R , or another solution is found with a better cost. This loop can continue, which would account for the extended times for some trials. Other trials may have had a short evaluation time comparably, due to reaching a low cost solution quickly, and then not having the ability to improve upon it.

6.3 The Control Parameter p_0

The second parameter examined was the effect of changes to the value of the variable, p_0 . This parameter controls the choice of whether a solution is accepted as an interim solution. The

Set	Best p_0
30	-5
30N	10
60	10
60	0
60N	5
80N	-5
100N	-20

Table 6.5 Best p_0 Values

value was varied in increments of five, from an initial value of 10, down to a value of negative 65. In table 6.5, we see the values of p_0 that gave the best cost for each problem set of the first group. This variable is designed to be a tuning variable for the genetic algorithm to help optimize the performance. As such, we find there is no standard value that is right for every problem set. We see that there is no connection between the best values of p_0 and the problem size. All other variables were held static to maintain control of the examination. Analysis of the 40 terminal problems sets gave a similar result. In all problem sets, the p_0 value of 10 gave a minimal cost, that could not be improved upon by lower values. Table 6.6 shows the results of the 40 terminal problem sets.

	40			40a			40b		
	Tracks	Vias	Cost	Tracks	Vias	Cost	Tracks	Vias	Cost
VHV	6	53	112	6	56	118	7	57	121
HVH	5	58	127	3	58	131	5	57	137
2Lyr	8	54	126	6	59	134	10	56	138
	40c			40d			40e		
	Tracks	Vias	Cost	Tracks	Vias	Cost	Tracks	Vias	Cost
VHV	6	52	110	8	54	116	6	56	118
HVH	5	58	127	5	58	131	5	55	135
2Lyr	8	57	122	9	57	129	6	56	138

Table 6.6 Effect of p_0 on the 40 Terminal Problem Sets

Generation size	Cost						
	30	30N	60	60a	60N	80N	100N
1	120	113	241	256	287	270	309
1	118	114	241	257	284	264	315
2	115	120	231	258	303	268	308
2	124	115	241	266	286	268	315
3	112	115	241	268	290	264	318
3	115	115	240	251	296	264	324
4	111	120	231	268	295	262	324
4	113	112	241	254	296	266	319

Generation size	Time						
	30	30N	60	60a	60N	80N	100N
1	31	46	6	65	142	132	82
1	21	56	6	76	200	111	84
2	31	39	6	51	102	127	68
2	10	27	6	37	121	75	62
3	37	20	9	14	159	100	53
3	16	24	10	81	120	111	56
4	35	23	5	19	129	72	54
4	27	31	5	63	119	78	54

Table 6.7 The Effect of Generation Size on Cost and Time

6.4 The Generation Size

The generation size was evaluated by performing trials on the problem sets while varying the number of child solutions created during each generation loop. The child solutions were varied from an initial value of one child solution per parent, to a final value of four

child solutions per parent. Table 6.7 gives the experimental results for generation sizes of one to four, on each of the problem sets.

We find no conclusive data to indicate that the generation size affects the quality of the solutions, nor does it dramatically affect the generation time. There are a few instances where the generation time appears to be reduced by the increase in generation size, indicating that having more diversity favors finding a solution more quickly, but the data does not clearly point to this hypothesis. Another possible conclusion to be drawn by the time statistics is to infer that the time saved by fewer generations to process is balanced by the additional generations required to reach a best solution. The analysis of the costs for various generation values does not point to a gain by larger populations of interim solutions.

6.5 The Control Parameter R

The control parameter R is used to set the point of termination of the genetic algorithm. To analyze this parameter's effect on the algorithm, the problem sets were tested with the value of R set to 100, 200, 400, 800, and 1600. As expected, the run-time doubles as R doubles, so we see a linear relationship of growth in time as R grows.

R Value	30	30N	60a	60N	80N	100N
100	110	105	241	243	250	329
200	112	108	241	242	245	332
400	112	105	240	239	244	304
800	110	108	240	245	238	316
1600	112	104	240	238	245	319

Table 6.8 The Effect of R on the Cost

In analyzing the cost as R increases, we see in Table 6.8 that the cost does improve as R increases for most all problem sets. The first problem set of 30, containing a cycle, does not reflect an improvement with an increase in R . This is because the `mk_parent` subroutine, that creates the initial parents, is able to create an optimal solution, before the genetic algorithm is invoked.

6.6 Using Elite Mode

Another feature of the multi-layer channel router is its ability to be run in an elite mode, or in a non-elite mode. As mentioned earlier, when the elite mode is selected, the selection of solutions from a generation, for the next generation's parents, is handled by a strict 'lowest cost' calculation. In the non-elite mode, the selection is randomly chosen between the above mentioned lowest cost calculation, and a purely random selection of solutions.

Elite Mode	Separation	Problem Set							
		80N Cost	Time	60 Cost	Time	100N Cost	Time	60N Cost	Time
0	2	266	90	257	50	365	85	282	121
1	2	268	76	265	23	368	96	282	112
0	3	250	69	243	22	327	79	261	99
1	3	248	50	243	25	335	58	289	42
0	4	248	50	241	18	308	74	276	66
1	4	240	66	237	24	321	51	280	35
0	5	252	27	241	4	323	46	250	77
1	5	248	35	241	5	325	33	256	22
0	6	245	46	241	4	314	63	252	84
1	6	244	56	231	4	328	53	256	36

Table 6.9 Effect of Elite Mode on Cost and Time

An analysis of table 6.9, showing the problem sets with elite mode selected and not selected, offers two observations. We are comparing the cost for each problem set, and for a specific separation value, with the elite mode selected and not selected. The comparison reveals that the cost is generally lower when elite mode is not selected. This indicates that there is an advantage to selecting parents in a more random manner, so as to offer a wider variety of parents. This broadens the creation of child solutions to better reach an optimal solution.

The second observation is regarding the time component. It appears that most all the problem sets, with most all the separation values, will terminate in a shorter period of time when the elite mode is selected. This would indicate that we are using a more

narrow view of the solution set, and therefore, more likely to fall into a local minimum and terminate before reaching the best solution.

6.7 Two-Layer Versus Three-Layer Analysis

The problem set having 100 terminal channel length was analyzed for the quality of solutions in both a three-layer and two-layer mode. The relative costs of the solutions are very similar, covering the same range of values in both modes. Since the number of tracks used is a smaller portion of the cost, compared to the vias and conflicts, the costs do not show a big change between the layer modes.

Evaluating the number of tracks needed for each solution does provide some comparison. As a reference, the maximum clique for the 100 terminal problem set is six, and the longest path is four. In the three-layer mode, the best solution requires four tracks, one more than should be necessary. This solution also requires 134 vias. In the two-layer

Set	Max	Tracks			Cost					
	Clique	VHV	HVH	2Lyr	VHV	Optimal	HVH	Optimal	2Lyr	Optimal
input40.txt	6	6	5	8	112	112	127	121	126	116
input40a.txt	6	6	3	6	118	118	131	119	134	124
input40b.txt	7	7	5	10	121	121	137	119	138	122
input40c.txt	6	6	5	8	110	110	127	121	122	122
input40d.txt	8	8	5	9	116	116	131	121	129	123
input40e.txt	6	6	5	6	118	118	135	115	138	118

Table 6.10 Analysis of Layer Mode on the 40 Terminal Problem Sets

mode, the best solution requires 6 tracks and 143 vias, which is optimal in terms of tracks required. Both are using a terminal separation value, so that each multi-net is treated as a single net, rather than several smaller nets.

Analyzing the 40 terminal problem sets in the two-layer and three-layer modes, we see that the algorithm excels in the VHV mode. Table 6.10 shows the findings for each problem set in the three different layering modes, and figure 6.4 shows an example of the track layout in each layering mode. The maximum clique gives us the optimal number of tracks required in the VHV and two-layer modes, while the HVH mode would have an optimal number of tracks equal to one-half of the maximum clique. In all cases, the VHV mode gave a solution with a minimal number of tracks required. The table also shows the optimal cost based on the number of tracks required and the number of vias necessary. Again, the VHV mode offered an optimal cost, with no extra costs due to track or via conflicts.

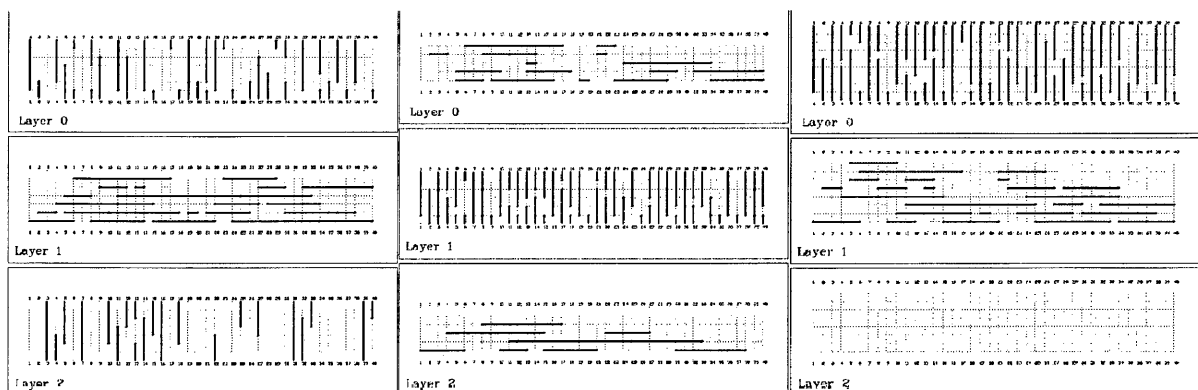


Figure 6.4 Layering Modes for 40 Terminal Problem Set

Now that we have an understanding of how the various parameters affect the performance of the algorithm, an understanding of the time complexity is in order.

6.8 Time Complexity

The efficiency of the multi-layer channel router is found by evaluating its main processes in terms of each of their individual sub-routines to find the accumulated overall complexity of each process. The main process performs its initial evaluation of the problem set in a linear fashion, so the evaluation of its sub-routines will give us the time complexity of this portion of the program. The second portion involves the creation of the parent solutions, followed by the genetic portion of the algorithm to generate child solutions. Each of these two latter portions requires a separate analysis.

6.8.1 Time Complexity of the Analysis Phase

The initial problem set analysis requires that several sub-routines execute. If we evaluate them in order, we can find the one with the highest time complexity, which will dominate the complexity of this portion of the process. In this phase, the sub-routines: reader, create nets, maximum clique, vertical constraints, and longest path are performed. An evaluation of these reveals that the time complexity of reader and maximum clique are of the order of the number of terminals in the problem set, or $O(\text{total_term})$.

We see that the other three sub-routines all require a time of $O(\text{total_term}^2)$ for their execution. These will dominate the time complexity of this phase, giving us a final complexity for the first portion of $O(\text{total_term}^2)$.

6.8.2 Time Complexity of the Parent Creation

The creation of the parent solutions requires the sub-routines, `mk_parent`, `cost`, `compact`, and `output_sol`. We have seen that maximum clique is of $O(\text{total_term})$ and `output_sol` requires the same amount of computation. The sub-routines, `cost` and `compact` require less time. Their time complexity is of $O(\text{total_seg})$, which is a subset of the total number of terminals, excluding the trivial nets.

A look at the `mk_parent` sub-routine shows a slightly more complex time requirement. We find a loop based on the value of the maximum clique within a loop based on the value of `total_seg`. This gives us a complexity of $O(\text{total_seg} * \text{max_clique})$. This is the dominant factor as compared with the other sub-routines. Even though the value of `total_seg` is smaller than `total_term`, its multiplication with `max_clique` causes it to dominate the process. This gives us a time complexity of $O(\text{total_seg} * \text{max_clique})$ for this phase of the algorithm.

6.8.3 Time Complexity of the Genetic Algorithm

The last phase to analyze is the genetic algorithm used to generate child solutions and in the process, find the best solution. The sub-routines, mutate, compact, and cost are involved, but special attention must be given to the Do-While loop that encloses all of this phase. Mutate requires a loop based on the value of the total number of nets. Within the mutate sub-routine, we call compact, which has a time complexity of $O(\text{total_seg})$. This can be evaluated to an upper bound of $O(\text{total_term})$, giving the sub-routine an overall complexity of $O(\text{total_term} * \text{nets})$.

We have previously evaluated the time complexity of cost to be of $O(\text{total_seg})$, and so we can disregard it for our analysis. Finally, we need to look at the effect the overall Do-While loop has on this phase. The Do-While loop contains a number of executions, including mutate, compact, and cost. Within this loop, we also find the control of the variable r is maintained until it reaches the termination value of R . This implies that the functions will be performed at least R times. If improvement is found, the value of r will be reduced by the value of R , but this would occur a minimal number of times, and so is considered constant. Multiplying the effect of R on the largest complexity, in this case mutate, we calculate an overall complexity for the third phase, of $O(R * \text{total_term} * \text{nets})$.

Adding the complexities of all three phases gives a time complexity of:

$$O((\text{total_term}^2) + (\text{total_seg} * \text{max_clique}) + (R * \text{total_term} * \text{nets}))$$

The middle term is dominated by the first and third terms, so we can see the time required is affected by the number of terminals, the number of nets, and the termination variable, R . This gives an understanding of the time required for the process. We see that as the problem set size increases and as the termination value increases, the time complexity grows quite quickly.

6.9 Comparing the Parents to the Final Solution

Our analysis of the genetic algorithm is not complete without a comparison of the parent solutions, as created by the greedy algorithm, to the final solution via the genetic algorithm. Table 6.11 compares the solutions created by the initial greedy algorithm with the final solutions after the genetic algorithm is applied. The table shows the maximum clique for each problem set, and the optimal cost for each solution.

We can see that in two examples, the greedy algorithm is able to create an optimal solution, so that the genetic algorithm is unable to improve upon it. Overall, the data indicates that the genetic algorithm is effective in reducing the cost of the final solution and, in most cases, is effective in reducing the maximum number of tracks required.

Problem Set	Max Clique	Parent Tracks	Vias	Cost	Optimal Cost	Final Tracks	Vias	Cost	Optimal Cost
Input30	12	11	50	117	111	12	50	112	112
Input30N	10	5	40	125	85	6	44	104	94
Input60	15	15	115	245	245	15	110	235	235
Input60a	14	15	113	241	241	15	113	241	241
Input60bN	14	10	130	390	270	7	124	287	255
Input80N	5	3	93	305	189	5	112	239	229
Input100N	6	3	113	363	229	6	147	310	300
Input40	6	6	53	112	112	6	53	112	112
Input40aN	6	4	53	162	110	4	53	140	110
Input40bN	7	4	36	146	76	5	47	125	99
Input40c	6	7	53	113	113	6	50	106	106
Input40d	8	8	56	120	120	8	54	116	116
Input40eN	6	3	43	155	89	4	47	128	98

Table 6.11 Comparison of Greedy and Genetic Algorithms

6.10 General Observations

There is an additional observation that merits discussion at this time. An overall analysis of the solutions generated by the various tests reveals an interesting finding. The algorithm favors a problem set with a cycle in its vertical constraint graph. In comparing the solutions, we see that those problem sets with cycles generated solutions that had an equal number of tracks to their maximum clique. They also returned solutions that were without track and via violations. The problem sets without cycles frequently returned solutions that were not optimal in the number of tracks required. In many cases, they were one or two tracks above the expected value of one half of the maximum clique.

They also occasionally contained some track and via conflicts, and so were not always viable solutions.

We can now offer our conclusions on the multi-layer channel router and identify possible directions for future research.

Chapter 7

Conclusions

7.1 A Recap of the Benefits

The multi-layer channel router offers an alternative method of resolving the detailed channel routing phase of VLSI design. Its use of a multi-layer design has proved its flexibility, and performance. We have shown how its three-layer approach can remove vertical constraints and overcome vertical constraint cycles that have been the limiting factor in many other algorithms. We have also shown that the three-layer approach can also narrow the channel width for those problem sets without vertical constraint cycles.

The separation variable has proven itself to be useful in determining the best way to handle multi-nets. We have seen that the value of the termination variable, R , has had little affect on the solution generation. The elite mode, of choosing the best solutions as future parents, has proven itself to be less effective in generating diverse and better child solutions, as compared with a more random approach to choosing parents. By the use of the genetic algorithm as its basis, we can overcome local minimums to try to reach a more optimal solution. We have also seen that the algorithm has very good performance, due to an efficient time complexity. Using the control variable, p_0 , and the separation variable, we can optimize the performance to generate solutions in a very reasonable period of time.

7.2 Positive and Negative Effects

There are two interesting features that make this algorithm attractive. The first is that it works very favorably on problem sets with cycles. Its ability to resolve vertical constraints and cycles, allows for an optimal solution in those situations. Secondly, by using the tuning parameter, p_0 , and the separation variable, the algorithm's performance is exceptional. The program took a minimal amount of time to process a problem set with as many as 100 terminals.

Using AutoCAD® Release 2000, as the graphic display of the results, we can easily see the quality of the resulting solutions. The program offers a simple approach to displaying the layers of the design to analyze any track or vertical conflicts. When viewing the solution, the 3D orbiting function can be used to rotate it in real-time, to better see how the tracks, vias, and verticals are arranged. Connectivity can be easily checked.

It is fair to also offer the disadvantages seen in this algorithm. There can be a problem with the initial parent creation. Since the algorithm uses a random approach to making changes, it does not know how to resolve a specific conflict between two nets. Experimentation shows that it has difficulty overcoming this. This problem was to be corrected by a sub-routine that tries to move conflicts to a clear location, but seems to have fallen short of its goal. This indicates a need for a more complex evaluation and replacement method, possibly by swapping the positions of offending nets.

Another issue of note is that the algorithm did not always generate an optimal solution for problem sets without vertical constraint cycles. These would have had an optimal solution with the number of required tracks equaling one-half of the maximum clique. The solutions approached this goal, but were usually one or more tracks over the optimal.

7.3 Directions for Future Research

One area of future research involves testing other algorithms as the basis for the parent generation. The greedy algorithm worked well, and did occasionally generate an optimal solution, but could also be responsible for the net conflicts that were previously mentioned. Other algorithms may generate different parent solutions, which may add the diversity needed to reach an optimal solution with every trial.

The compaction portion of the multi-layer channel router is also an interesting avenue of further investigation. It has shown itself to be helpful in reducing unnecessary tracks, but could be improved to be more robust in analyzing problem areas. Conflicts with track assignments and vertical assignments require special attention to resolve them. Alternative methods of reducing excess tracks, and placing nets with improper locations, can be investigated.

Bibliography

- [BP83] Burstein, M. and Pelavin, R. (1983), Hierarchical channel router, *Proceedings of the 20th ACM / IEEE Design Automation Conference*, pages 519-597, 1983.
- [CL84] Chen, Y. and Liu, M. (1984), Three-layer channel routing, *IEEE Transactions on Computer-Aided Design*, CAD-3(2): pages 156-163, April 1984.
- [CW87] Cong, J., Wong, D. F., and Liu, C. L. (1987), A new approach to the three-layer channel routing problem, *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 378-381, 1987.
- [DD76] Deutsch, D. N. (1976), A dogleg channel router, *Proceedings of the 13th ACM/IEEE Design Automation Conference*, pages 425-433.
- [HJ75] Holland, J. H. (1975), *Adaptation in Natural and Artificial Systems*, The University of Michigan Press.
- [HS71] Hashimoto, A. and Stevens, J. (1971), Wire routing by optimization channel assignment within large apertures, *Proceedings of the 8th Design Automation Workshop*, pages 155-163.
- [MT83] Marek-Sadowska, M. and Trang, T. T. (1983), Single-layer routing for vlsi: Analysis and algorithms, *IEEE Transactions on Computer-Aided Design*, pages 208-219, October 1983.
- [PZ87] Pitchumani, V. and Zhang, Q. (1987), A mixed hvh-vhv algorithm for three-layer channel routing, *IEEE Transactions on Computer-Aided Design*, CAD-6(4), 1987.

- [RD84] Richards, D. (1984), Complexity of single-layer routing, *IEEE Transactions on Computers*, C-33(3): pages 286-288, March 1984.
- [RF82] Rivest, R. and Fiduccia, C. (1982), A greedy channel router, *Proceedings of the 19th ACM/ IEEE Design Automation Conference*, pages 418-424, 1982.
- [RO93] Rahmani, A. T. and Ono, N. (1993), A genetic algorithm for the channel routing problem, *Proceedings of the ICGA*, pages 494-498, 1993.
- [RS85] Reed, J., Sangiovanni-Vincentalli, A., and Santamauro M. (1985), A new symbolic channel router: Yacr2, *IEEE Transactions on Computer-Aided Design*, CAD-4(3): pages 208-219, 1985.
- [SG85] Szymanski, T. G., Dogleg channel routing is np-complete, *IEEE Transactions on Computer-Aided Design*, CAD-4: pages 31-41, January 1985.
- [SN95] Sherwani, Naveed (1995), *Algorithms for VLSI Physical Design Automation*, Kluwer Academic Publishers.
- [YK82] Yoshimura, T. and Kuh, E. S. (1982), Efficient algorithms for channel routing, *IEEE Transactions on Computer-Aided Design*, CAD-1(1): pages 25-30, January 1982.
- [ZJ98] Zheng, Jingsen (1998), *An Evolution-Based Approach for the Channel Routing Problem in VLSI Physical Design*, Masters Thesis, University of Nebraska at Omaha.

Appendix A: Source Code

```

/*****
/*                               Mark P. Cloyd                               */
/* December 1999                 508-82-1623                 CSCI 8990 */
/*                               */
/* This program takes as input a text file containing to terminal */
/* lists and outputs a three layer net list containing net terminals,*/
/* layer and track used to implement the nets most efficiently. A */
/* genetic algorithm is used to find the best solution.           */
*****/

#include <stdio.h>
#include <time.h>
#include <stdlib.h>
#include <string.h>

#define TERMS 140
#define MAX_NETS 170
#define TRAX 40
#define num_prt 5
#define num_cld 4

                                /*declarations*/

int  t_in[TERMS], b_in[TERMS], /*top and bottom terminal list*/
    left_pt_list[TERMS*2+1],
    right_pt_list[TERMS*2+1], /*stores net # by left and right most points*/
    top_net,                    /*highest numbered net*/
    sep,                        /*variable used to separate multinets*/
    total_nets,                /*number of nets*/
    total_term,                /*total # of terminals*/
    cur_net,                    /*net currently evaluated*/
    total_seg,                 /*stores number of nets that require a track*/
    max_clique,                /*stores the maximum clique value for the nets*/
    longestpath,               /*stores the length of the longest path*/
    vcg[TERMS][TERMS],        /*vertical constraint graph to find cycles*/
    color[TERMS],              /*array used to color visited nodes for cycles*/
    oldcost,                    /*used to store old cost for comparison purposes*/
    solcost[num_prt*(num_cld+1)+1], /*stores the cost associated with each solution*/
    saved[num_prt],             /*used to organize the top 5 solutions*/
    layers,                     /*number of layers used for tracks*/
    vhv,                        /*determines whether to use 2 vert or 2 horz*/
    elite,                      /*determines whether to save best parents or random*/
    r,                          /*incrementing variable*/
    p,p0,bigr,                  /*control parameters*/
    j1,                          /*child generation counter*/
    i,j,k;                       /*loop counters*/

char s[12],
     out[13];                       /*outfile name storage*/

struct net {
    int net_num;                    /*stores the net's number start and end points*/
    int strt_pin;
    int end_pin;
};

struct layout {
    int track;                      /*stores the assigned track and layer for solutions*/
    int layer;
};

struct net netlist[MAX_NETS]; /*stores list of nets*/

```

```

struct layout sol[num_prt*(num_cld+1)+1][MAX_NETS]; /*stores the derived solutions*/

/*****
/***** main program *****/
/*****
main(int argc, char *argv[])
{
    int i,j,k;

    if(argc < 1) {
        printf("Usage: 3lyrchnl Input_file\n");
        exit(1);
    }
    for(i=0;i<(num_prt*(num_cld+1)+1);i++) {
        for(j=0;j<MAX_NETS;j++) {
            sol[i][j].layer=0;
            sol[i][j].track=0;
        }
    }
    randomize();
    k = 1;
    do {
        printf("What is the separation value for multinets (2-99): ");
        scanf("%d",&sep);
        if((sep > 1) && (sep < 100)) k = 0;
        else printf("Please input a number between 2 and 99.\n");
    } while(k);
    printf("Separation = %d\n",sep);
    printf("Input an integer for p0: "); //input control parameters
    scanf("%d",&p0);
    printf("p0 = %d\n",p0);
    printf("Input an integer for R: ");
    scanf("%d",&bigr);
    printf("R = %d\n",bigr);
    printf("Run in elite mode (0/1): ");
    scanf("%d",&elite);
    printf("Elite mode = %d.\n",elite);
    p = p0;
    r = 0;
    reader(argv[1]);
    top_net=0; /*calculate largest net # */
    for(i=1;i<=total_term;i++) {
        if(t_in[i] > top_net) top_net = t_in[i];
        if(b_in[i] > top_net) top_net = b_in[i];
    }
    create_nets();
    max_clique = m_clique();
    vert_cnstrt();
    if(cycle_ck()) {
        printf("Terminal list contains cycles. VHV layering favored.\n");
        vhv=1;
    }
    printf("VHV 0/1? ");
    scanf("%d",&vhv);
    printf("VHV = %d.\n",vhv);
    k=1;
    do {
        printf("Create 2 or 3 layer model?");
        scanf("%d",&layers);
        layers--;
        if((layers > 0) && (layers < 3)) k=0;
        else printf("Number of layers must be 2 or 3.\nPlease try again.\n");
    } while (k);
    printf("\nUsing %d layers.\n",layers+1);
    if(!vhv) {
        longestpath=longest_path();

```

```

    printf("Longest path = %d.\n",longestpath);
} else printf("Longest path can not be calculated due to cycles in the Terminal
List.\n");
printf("Creating Parent solutions.\n");
for(i=1;i<=num_prt;i++) {
    mk_parent(i); /*create parent solutions. */
    sol[i][0].layer = via_count(i);
    solcost[i] = cost(i);
    printf("Cost of %d.\n",solcost[i]);
}
sprintf(out,"%s","parent");
printf("Creating output files for %s.\n",out);
for(i=1;i<=num_prt;i++) {
    sprintf(out,"%s","parent");
    output_sol(i,out);
    compact(i);
    sol[i][0].layer = via_count(i);
    solcost[i] = cost(i);
}
oldcost = 19999;
for(i=1;i<=num_prt;i++) {
    if(solcost[i] < oldcost) {
        oldcost = solcost[i];
        saved[0] = i;
    }
}
printf("Saving best of parents\n");
for(i=0;i<=total_nets;i++) {
    sol[0][i].layer = sol[saved[0]][i].layer;
    sol[0][i].track = sol[saved[0]][i].track;
}
solcost[0] = solcost[saved[0]];
printf("Best solution so far:\n");
print_net(0);
do{ /*creates new generations
    for(i=1;i<=num_prt;i++){
        oldcost = solcost[i];
        for(j=1;j<=num_cld;j++){
            j1 = num_prt+(i-1)*num_cld+j;
            mutate(i,j1);
            compact(j1);
            sol[j1][0].layer = via_count(j1);
            solcost[j1] = cost(j1);
            update(oldcost,solcost[j1]);
            if(solcost[j1] < solcost[0]) { //Save to best if it is the best so far.
                solcost[0] = solcost[j1];
                for(k=0;k<=total_nets;k++) {
                    sol[0][k].layer = sol[j1][k].layer;
                    sol[0][k].track = sol[j1][k].track;
                }
                r=r-bigr;
            }
            else r++;
        }
    }
}
for(i=0;i<num_prt;i++){ //finds best to save
    oldcost = 19999;
    for(j=1;j<=num_prt*(num_cld+1);j++){
        if(solcost[j] < oldcost){
            saved[i] = j;
            oldcost = solcost[j];
        }
    }
    solcost[saved[i]] = 19999;
}
sort_them();
for(i=1;i<=num_prt;i++) {

```

```

        for(j=0;j<=total_nets;j++) {
            sol[i][j].layer = sol[saved[i-1]][j].layer;
            sol[i][j].track = sol[saved[i-1]][j].track;
        }
        sol[i][0].layer = via_count(i);
        solcost[i] = cost(i);
        sol[i][0].layer = via_count(i);
        solcost[saved[i-1]] = solcost[i];
    }
    printf(".");
} while(r < bigr);
sprintf(out,"%s","best");
printf("\n\nCreating output file for %s.\n",out);
sol[0][0].layer = via_count(0);
solcost[0] = cost(0);
print_net(0);
output_sol(0,out);
exit(0);
}
/*****
/***** End Main *****/
/*****/

/* reader reads in the input from the text file and translates it */
/* into two arrays of terminal lists */
reader(char *finp) {

    char rs[4];
    int net_cnt,
        i, t, r;
    FILE *fp;

    printf("Loading Input File: %s\n",finp);
    if((fp=fopen(finp,"r"))==NULL) {
        printf("Cannot open file.\n");
        exit(1);
    }
    fgets(rs,5,fp);
    total_term = atoi(rs);
    if(total_term > TERMS-1) {
        printf("Number of terminals is beyond program limits. Exiting now.\n");
        exit(1);
    }
    printf("Number of terminals=%d\n",total_term);
    for(i=1;i<=total_term;i++){
        fgets(rs,5,fp);
        t = atoi(rs);
        t_in[i] = t;
    }
    for(i=1;i<=total_term;i++){
        fgets(rs,5,fp);
        t = atoi(rs);
        b_in[i] = t;
    }
    fclose(fp);
    printf("\nPin #");
    for(i=1;i<=total_term;i++) printf("%4d",i);
    printf("\nT_In:");
    for(i=1;i<=total_term;i++) printf("%4d",t_in[i]);
    printf("\nB_In:");
    for(i=1;i<=total_term;i++) printf("%4d",b_in[i]);
    printf("\n\n");
    return 0;
}
/**** end reader ****/
/*****/
/* create_nets uses the terminal inputs to create a list of nets */
create_nets(){
    int i,j,

```

```

    count;

total_nets=0;
cur_net=1;
for(i=1;i<=total_term;i++) {
    count=0;
    for(j=1;j<=total_term;j++) {
        if(t_in[j] == i || b_in[j] == i) {
            if (! count){
                count++;
                netlist[cur_net].net_num=i;
                netlist[cur_net].strt_pin=j;
            }
            else{
                netlist[cur_net].end_pin=j;
                count++;
                if(count == sep) {
                    cur_net++;
                    if(cur_net==MAX_NETS) {
                        printf("\n\nThe input has exceeded the bounds of this program.");
                        printf("\nPlease reduce the input size and try again.\nExiting. . .");
                        exit(1);
                    }
                    count=1;
                    netlist[cur_net].net_num=i;
                    netlist[cur_net].strt_pin=j;
                }
            }
        }
    }
    if(count > 1) cur_net++;
}
total_seg = cur_net-1;
for(j=1;j<=total_term;j++) {
    if(t_in[j] == b_in[j]) {
        netlist[cur_net].net_num=t_in[j];
        netlist[cur_net].strt_pin=j;
        netlist[cur_net].end_pin=j;
        cur_net++;
    }
}
total_nets=cur_net-1;
printf("Number of Nets: %d\n\nNet List:\nNum  Strt  End\n",total_nets);
for(i=1;i<=total_nets;i++) printf("%3d %5d
%5d\n",netlist[i].net_num,netlist[i].strt_pin,netlist[i].end_pin);
return 0;
}
    /** end create_nets */
/*****
/* m_clique calculates the size of the maximum clique for the nets */
m_clique() {
    int i,j,
        num,st,nd,
        clique,mclique;

    for(i=0;i<=total_seg*2;i++) left_pt_list[i]=right_pt_list[i]=0;
    for(i=1;i<=total_seg;i++) {
        num=netlist[i].net_num;
        st=netlist[i].strt_pin;
        nd=netlist[i].end_pin;
        if(left_pt_list[2*(st-1)]==0) left_pt_list[2*(st-1)]=num;
        else left_pt_list[2*(st-1)+1]=num;
        if(right_pt_list[2*(nd-1)]==0) right_pt_list[2*(nd-1)]=num;
        else right_pt_list[2*(nd-1)+1]=num;
    }
    clique=0;
    mclique=0;
    for(i=0;i<2*total_term;i++) {

```

```

    if(left_pt_list[i] > 0) clique++;
    if(right_pt_list[i] > 0) clique--;
    if(clique > mclique) mclique = clique;
}
printf("\nMaximum Clique is %d.\n",mclique);
return mclique;
}
/** end max_clique */
/*****
/* vert_cnstrt calculates the vertical constraints present in the */
/* terminal layout */
vert_cnstrt() {

    for(i=1;i<TERMS;i++) {
        for(j=1;j<TERMS;j++) vcg[i][j] = 0;
    }
    for(i=1;i<=total_term;i++) {
        if((t_in[i]==0)|| (b_in[i]==0) || (t_in[i]==b_in[i])) ;
        else vcg[t_in[i]][b_in[i]]=1;
    }
    return 0;
}
/** end vert_cnstrt */
/*****
/* cycle_ck checks for cycles in the vertical constraints array */
cycle_ck() {
    int i,j,
        cycle;

    printf("Checking for cycles.\n");
    cycle=0;
    for(i=1;i<TERMS;i++) color[i] = 0;
    for(i=1;i<=total_term;i++) {
        if(color[i]==0) cycle = srch(i);
        if(cycle) {
            printf(" create a cycle.\n");          //added to check cycle #
            return 1;
        }
    }
    return 0;
}
/** end cycle_ck */
/*****
/* srch performs a depth first search for nodes that have already */
/* had a visit. This is used by cycle_ck to look for cycles. */
srch(int u) {
    int i,j,
        cycle;

    color[u] = 1;
    for(i=1;i<=total_term;i++) {
        if(vcg[u][i]==1) {
            if(color[i]==1) {
                printf("%d,",i);                //added to check cycle #
                return 1;
            }
            else if(color[i]==0) cycle = srch(i);
            if(cycle) {
                printf("%d,",i);                //added to check cycle #
                return 1;
            }
        }
    }
    color[u] = 2;
    return 0;
}
/** end srch */
/*****
/* longest_path calculates the longest path of the vertical */
/* constraint graph */
longest_path() {

```

```

int i,j,
    path_len,
    longpath;

longpath = 0;
for(i=1;i<=total_term;i++) {
    path_len = 0;
    path_len = dfs(i,path_len);
    if(longpath < path_len) longpath = path_len;
}
return longpath;
}                                     /** end longest_path ***/
/*****
/* dfs performs a depth first search of nodes to calculate the */
/* longest path from the currently evaluated node to any other */
/* directly and indirectly adjacent node in the vcg.          */
dfs(int node, int p_len) {
    int i;

    for(i=node+1;i<=total_term;i++)
        if(vcg[node][i] == 1) {
            p_len++;
            p_len = dfs(i,p_len);
        }
    return p_len;
}                                     /** end dfs ***/
/*****
/* mk_parent creates the initial parent solution upon which the */
/* children solutions will be derived.                          */
mk_parent(int sol_num) {
    int netnum,
        max_track,
        lyr_ck[3][TRAX][TERMS],
        l_pt,r_pt,
        l,t,
        ok,not_ok,
        i,j,k;

    max_track = 0;
    for(i=0;i<=2;i++)                //clear matrix to check placement.
        for(j=0;j<TRAX;j++)
            for(k=1;k<TERMS;k++) lyr_ck[i][j][k] = 0;
    cur_net = random(total_seg)+1;
    for(i=1;i<=total_seg;i++){       //place non-trivial segments first.
        netnum = netlist[cur_net].net_num;
        l_pt = netlist[cur_net].strt_pin;
        r_pt = netlist[cur_net].end_pin;
        if((netnum==t_in[l_pt])&&(netnum==t_in[r_pt])&&(l_pt!=r_pt))
            for(t=max_clique+1;t>=0;t--) { //if a top net, start at max_clique+1.
                for(l=2-layers;l<=layers;l=l+2) {
                    ok = ck_space(sol_num,netnum,lyr_ck,l,t,l_pt,r_pt);
                    if(ok) goto newnet;
                }
            }
        else for(t=0;t<=max_clique+4;t++){ /*find layer and track to use*/
            for(l=2-layers;l<=layers;l=l+2) {
                ok = ck_space(sol_num,netnum,lyr_ck,l,t,l_pt,r_pt);
                if(ok) goto newnet;
            }
        }
        newnet;                       /* Storing location */
        if((vhv) && (layers == 2)) {
            for(j=l_pt;j<=r_pt;j++) {
                lyr_ck[l][t][j] = netnum;
                if(b_in[j] == netnum) for(k=0;k<=t;k++) lyr_ck[l][k][j] = netnum;
                if(t_in[j] == netnum) for(k=t;k<max_track;k++) lyr_ck[l][k][j] = netnum;
            }
        }
    }
}

```

```

    } else {
        for(j=l_pt;j<=r_pt;j++) {
            lyr_ck[l][t][j] = netnum;
            if(b_in[j] == netnum) for(k=0;k<=t;k++) lyr_ck[layers-1][k][j] = netnum;
            if(t_in[j] == netnum) for(k=t;k<max_track;k++) lyr_ck[layers-1][k][j] = netnum;
        }
    }
    sol[sol_num][cur_net].track = t;
    sol[sol_num][cur_net].layer = l;
    if(cur_net == total_seg) cur_net = 1;
    else cur_net++;
    if(max_track < t) max_track = t;
}
max_track++;
for(i=total_seg+1;i<=total_nets;i++) { //place trivial nets.
    netnum = netlist[i].net_num;
    l_pt = netlist[i].strt_pin;
    not_ok = 1;
    for(t=0;t<=max_clique;t++) {
        for(l=2-layers;l<=layers;l=l+2) {
            if(netnum == lyr_ck[l][t][l_pt]) {
                sol[sol_num][i].track = t;
                sol[sol_num][i].layer = l;
                not_ok = 0;
            }
        }
    }
    if(not_ok) {
        sol[sol_num][i].track = 0;
        sol[sol_num][i].layer = 2-layers;
    }
}
sol[sol_num][0].track = max_track; /*store number of tracks used */
printf("\nSolution %d:\n",sol_num);
print_net(sol_num);
return 0;
}
/*** end mk_parent ***/
/*****
print_net(int sol_num) {
    int i;

    for(i=1;i<=total_nets;i++)
        printf("%2d. Net: %2d, l_pt: %2d, r_pt: %2d, Track: %2d, Layer:
%d\n",i,netlist[i].net_num,netlist[i].strt_pin,netlist[i].end_pin,sol[sol_num][i].track,s
ol[sol_num][i].layer);
    if(solcost[sol_num]!=0) printf("Cost of %d.\n",solcost[sol_num]);
    printf("Max track = %d\n",sol[sol_num][0].track);
    if(sol[sol_num][0].layer!=0) printf("Vias      = %d\n",sol[sol_num][0].layer);
    return 0;
}
/*** end print_net ***/
/*****
/* ck_space looks in layer l, track t for enough space to place
/* the current net being examined.
ck_space(int sol_num, int nett, int lyr[3][TRAX][TERMS], int l, int t, int l_pt, int
r_pt) {
    int max_track,
        i,j;

    max_track = sol[sol_num][0].track;
    if((vhv) && (layers == 2)) {
        for(i=l_pt;i<=r_pt;i++) {
            if((lyr[l][t][i] > 0) && (lyr[l][t][i] != nett)) return 0;
            if(b_in[i] == nett) for(j=0;j<=t;j++) if((lyr[l][j][i] > 0) && (lyr[l][j][i] !=
nett)) return 0;
            if(t_in[i] == nett) for(j=t;j<max_track;j++) if((lyr[l][j][i] > 0) && (lyr[l][j][i]
!= nett)) return 0;
        }
    }
}

```



```

    } else {
        for(i=l_pt;i<=r_pt;i++) {
            if((lyr[l][t][i] > 0) && (lyr[l][t][i] != nett)) return 0;
            if(b_in[i] == nett) for(j=0;j<=t;j++) if((lyr[layers-1][j][i] > 0) && (lyr[layers-1][j][i] != nett)) return 0;
            if(t_in[i] == nett) for(j=t;j<max_track;j++) if((lyr[layers-1][j][i] > 0) && (lyr[layers-1][j][i] != nett)) return 0;
        }
    }
    return 1;
}
    /** end ck_space */
/*****
/* output_sol creates an output file in the form of a script file */
/* for AutoCAD to use as input to graphically draw the net diagram.*/
output_sol(int sol_num, char out_name[13]) {
    int max_track,
        i,j,k,l,t,
        l_pt,r_pt;
    char n[3];
    FILE *fout;

    if(sol_num) {
        sprintf(n,"%d",sol_num);
        strcat(out_name,n);
    }
    strcat(out_name,".scr");
    if((fout=fopen(out_name,"w"))==NULL) {
        printf("Cannot open file.\n");
        return 1;
    }
    max_track=sol[sol_num][0].track;
    fprintf(fout,"%s\n","osmode 0 -layer s terminals "); //create terminals
    for(i=1;i<=total_term;i++){
        fprintf(fout,"%s%d%s%d%s\n","line ",i,"",-1,0 ",i,"",-1,2 ");
        fprintf(fout,"%s%d%s%d%s%d%s\n","line ",i,"",max_track,"0 ",i,"",max_track,"2
");
    }
    fprintf(fout,"%s\n","-layer s text "); //labels the terminal positions
    for(i=1;i<=total_term;i++){
        fprintf(fout,"%s%d%s%d\n","text ",i,"",-1.5,0 ",i);
        fprintf(fout,"%s%d%s%f%s%d\n","text ",i,"",max_track+0.2,"0 ",i);
    }
    fprintf(fout,"%s\n","-layer s grid "); //draw in grid
    for(i=1;i<=total_term;i++)
        fprintf(fout,"%s%d%s%d%s%d%s\n","line ",i,"",0,0 ",i,"",max_track-1,"0 ");
    for(i=0;i<max_track;i++)
        fprintf(fout,"%s%d%s%d%s%d%s\n","line 1",i,"",0 ",total_term,"",i,"0 ");
    if((vhv) && (layers==2)) {
        for(j=0;j<3;j=j+2){
            for(i=1;i<=total_nets;i++){
                l = sol[sol_num][i].layer;
                if(l==j) {
                    fprintf(fout,"%s\n","-layer s net1 "); //draw in nets
                    t = sol[sol_num][i].track;
                    l_pt = netlist[i].strt_pin;
                    r_pt = netlist[i].end_pin;
                    fprintf(fout,"%s%d%s%d%s%d%s%d%s\n","line ",l_pt,"",t,"",l,"
",r_pt,"",t,"",l," ");
                    if(i<total_seg) fprintf(fout,"%s%d%s%d%s%d%s%d%s%d%s\n","circle
",l_pt,"",t,"",l," 0.1 circle ",r_pt,"",t,"",l," 0.1");
                    for(k=l_pt;k<=r_pt;k++) {
                        //put verticals on appropriate layer and add vias
                        if(b_in[k]==netlist[i].net_num) {
                            fprintf(fout,"%s%d%s\n","-layer s net",j," ");
                            fprintf(fout,"%s%d%s%d%s%d%s%d%s\n","line ",k,"",t,"",l," ",k,"-
1,"",l," ");
                            if(i<total_seg) {

```

```

        fprintf(fout,"%s%d%s%d%s%d%s\n","circle ",k,",",t,",",l," 0.1");
        fprintf(fout,"%s\n","-layer s via ");
        fprintf(fout,"%s%d%s%d%s%d%s%d%s%d%s\n","line ",k,",",t,",",l,"
",k,",",t,",",l," ");
    }
    if(t_in[k]==netlist[i].net_num) {
        fprintf(fout,"%s%d%s\n","-layer s net",j," ");
        fprintf(fout,"%s%d%s%d%s%d%s%d%s%d%s\n","line ",k,",",t,",",l,"
",k,",",max_track,",",l," ");
        if(i<=total_seg) {
            fprintf(fout,"%s%d%s%d%s%d%s\n","circle ",k,",",t,",",l," 0.1");
            fprintf(fout,"%s\n","-layer s via ");
            fprintf(fout,"%s%d%s%d%s%d%s%d%s%d%s\n","line ",k,",",t,",",l,"
",k,",",t,",",l," ");
        }
    }
}
}
} else {
    for(j=2-layers;j<=layers;j=j+2){
        for(i=1;i<=total_nets;i++){
            l = sol[sol_num][i].layer;
            if(l==j) {
                fprintf(fout,"%s%d%s\n","-layer s net",j," ");           //draw in nets
                t = sol[sol_num][i].track;
                l_pt = netlist[i].strt_pin;
                r_pt = netlist[i].end_pin;
                fprintf(fout,"%s%d%s%d%s%d%s%d%s\n","line ",l_pt,",",t,",",l,"
",r_pt,",",t,",",l," ");
                if(i<=total_seg) fprintf(fout,"%s%d%s%d%s%d%s%d%s\n","circle
",l_pt,",",t,",",l," 0.1 circle ",r_pt,",",t,",",l," 0.1");
                for(k=l_pt;k<=r_pt;k++) {           //put verticals on
                    appropriate layer and add vias
                    if(b_in[k]==netlist[i].net_num) {
                        fprintf(fout,"%s%d%s\n","-layer s net",layers-1," ");
                        fprintf(fout,"%s%d%s%d%s%d%s%d%s\n","line ",k,",",t,",",layers-1,"
",k,",",-1,",layers-1," ");
                        if(i<=total_seg) {
                            fprintf(fout,"%s%d%s%d%s%d%s\n","circle ",k,",",t,",",layers-1," 0.1");
                            fprintf(fout,"%s\n","-layer s via ");
                            fprintf(fout,"%s%d%s%d%s%d%s%d%s\n","line ",k,",",t,",",l,"
",k,",",t,",",layers-1," ");
                        }
                    }
                    if(t_in[k]==netlist[i].net_num) {
                        fprintf(fout,"%s%d%s\n","-layer s net",layers-1," ");
                        fprintf(fout,"%s%d%s%d%s%d%s%d%s\n","line ",k,",",t,",",layers-1,"
",k,",",max_track,",",layers-1," ");
                        if(i<=total_seg) {
                            fprintf(fout,"%s%d%s%d%s%d%s\n","circle ",k,",",t,",",layers-1," 0.1");
                            fprintf(fout,"%s\n","-layer s via ");
                            fprintf(fout,"%s%d%s%d%s%d%s%d%s\n","line ",k,",",t,",",l,"
",k,",",t,",",layers-1," ");
                        }
                    }
                }
            }
        }
    }
}
fclose(fout);
sprintf(out_name,"");
return 0;
}
    /*** end output_sol ***/

```

```

/*****
/*compact is used to compact the space within the tracks to try */
/*to minimize the number of tracks used */
compact(int sol_num) {
    int netnum,
        max_track,
        lyr_ck[3][TRAX][TERMS],
        l_pt,r_pt,
        l,t,ll,tl,
        reset[MAX_NETS],
        ok,not_ok,
        i,j,k;

    for(i=0;i<=2;i++) //clear matrix to check placement.
        for(j=0;j<TRAX;j++)
            for(k=1;k<TERMS;k++) lyr_ck[i][j][k]=0;
    for(i=0;i<MAX_NETS;i++) reset[i]=0;
    for(i=1;i<=total_seg;i++){
        netnum = netlist[i].net_num;
        l_pt = netlist[i].strt_pin;
        r_pt = netlist[i].end_pin;
        l = sol[sol_num][i].layer;
        t = sol[sol_num][i].track;
        max_track = sol[sol_num][0].track;
        not_ok = 0;
        if((vhv) && (layers==2)) {
            for(j=l_pt;j<=r_pt;j++) {
                if((lyr_ck[l][t][j] > 0) && (lyr_ck[l][t][j] != netnum)) {
                    not_ok = 1;
                    goto next_one;
                }
                if(b_in[j] == netnum) for(k=0;k<=t;k++) if((lyr_ck[l][k][j] > 0) &&
(lyr_ck[l][k][j] != netnum)) {
                    not_ok = 1;
                    goto next_one;
                }
                if(t_in[j] == netnum) for(k=t;k<max_track;k++) if((lyr_ck[l][k][j] > 0) &&
(lyr_ck[l][k][j] != netnum)) {
                    not_ok = 1;
                    goto next_one;
                }
            }
            for(j=l_pt;j<=r_pt;j++) {
                lyr_ck[l][t][j]=netnum;
                if(b_in[j] == netnum) for(k=0;k<=t;k++) lyr_ck[l][k][j] = netnum;
                if(t_in[j] == netnum) for(k=t;k<max_track;k++) lyr_ck[l][k][j] = netnum;
            }
        } else {
            for(j=l_pt;j<=r_pt;j++) {
                if((lyr_ck[l][t][j] > 0) && (lyr_ck[l][t][j] != netnum)) {
                    not_ok = 1;
                    goto next_one;
                }
                if(b_in[j] == netnum) for(k=0;k<=t;k++) if((lyr_ck[layers-1][k][j] > 0) &&
(lyr_ck[layers-1][k][j] != netnum)) {
                    not_ok = 1;
                    goto next_one;
                }
                if(t_in[j] == netnum) for(k=t;k<max_track;k++) if((lyr_ck[layers-1][k][j] > 0) &&
(lyr_ck[layers-1][k][j] != netnum)) {
                    not_ok = 1;
                    goto next_one;
                }
            }
        }
        for(j=l_pt;j<=r_pt;j++) {
            lyr_ck[l][t][j] = netnum;
            if(b_in[j] == netnum) for(k=0;k<=t;k++) lyr_ck[layers-1][k][j] = netnum;
        }
    }
}

```

```

        if(t_in[j] == netnum) for(k=t;k<max_track;k++) lyr_ck[layers-1][k][j] = netnum;
    }
}
next_one:
if(not_ok) {
    reset[0]++;
    reset[reset[0]] = i;
}
}
if(reset[0] > 0) expand(reset,lyr_ck,sol_num); //find a new location for violations
for(i=1;i<=total_seg;i++) {
    netnum = netlist[i].net_num;
    l_pt = netlist[i].strt_pin;
    r_pt = netlist[i].end_pin;
    l1 = sol[sol_num][i].layer;
    t1 = sol[sol_num][i].track;
    // check up to existing location for a better place to put
it.
    for(t=0;t<t1;t++) { //find a layer and track to use*/
        for(l=2-layers;l<=layers;l=l+2) {
            ok = ck_space(sol_num,netnum,lyr_ck,l,t,l_pt,r_pt);
            if(ok) goto newnet;
        }
    }
    goto nobetterplace;
newnet:
if((vhv) && (layers==2)) {
    for(j=l_pt;j<=r_pt;j++) {
        lyr_ck[l][t1][j] = 0;
        lyr_ck[l][t][j] = netnum;
        if(b_in[j] == netnum) {
            for(k=0;k<=t1;k++) lyr_ck[l1][k][j] = 0;
            for(k=0;k<=t;k++) lyr_ck[l][k][j] = netnum;
        }
        if(t_in[j] == netnum) {
            for(k=t1;k<max_track;k++) lyr_ck[l1][k][j] = 0;
            for(k=t;k<max_track;k++) lyr_ck[l][k][j] = netnum;
        }
    }
} else {
    for(j=l_pt;j<=r_pt;j++) {
        lyr_ck[l1][t1][j] = 0;
        lyr_ck[l][t][j] = netnum;
        if(b_in[j] == netnum) {
            for(k=0;k<=t1;k++) lyr_ck[layers-1][k][j] = 0;
            for(k=0;k<=t;k++) lyr_ck[layers-1][k][j] = netnum;
        }
        if(t_in[j] == netnum) {
            for(k=t1;k<max_track;k++) lyr_ck[layers-1][k][j] = 0;
            for(k=t;k<max_track;k++) lyr_ck[layers-1][k][j] = netnum;
        }
    }
}
}
sol[sol_num][i].track=t;
sol[sol_num][i].layer=l;
nobetterplace:
}
for(i=total_seg+1;i<=total_nets;i++) { //check trivial nets.
    netnum = netlist[i].net_num;
    l_pt = netlist[i].strt_pin;
    l1 = sol[sol_num][i].layer;
    t1 = sol[sol_num][i].track;
    not_ok=1;
    for(t=0;t<=t1;t++) {
        for(l=2-layers;l<=layers;l=l+2) {
            if(netnum == lyr_ck[l][t][l_pt]) {
                sol[sol_num][i].track=t;
            }
        }
    }
}

```

```

        sol[sol_num][i].layer=1;
        not_ok=0;
    }
}
if(not_ok) {
    sol[sol_num][i].track=0;
    sol[sol_num][i].layer=2-layers;
}
}
max_track = 0; //calculate new # of tracks needed
if((vhv) && (layers==2)) {
    for(t=0;t<max_clique+4;t++) {
        for(j=1;j<=total_term;j++) {
            if(lyr_ck[1][t][j] > 0) {
                if(max_track < t) max_track = t;
                goto newtrack;
            }
        }
        newtrack:
    }
} else {
    for(t=0;t<max_clique+4;t++) {
        for(j=1;j<=total_term;j++) {
            if((lyr_ck[2-layers][t][j] > 0) || (lyr_ck[layers][t][j] > 0)) {
                if(max_track < t) max_track = t;
                goto newtrack1;
            }
        }
        newtrack1:
    }
}
max_track++;
sol[sol_num][0].track=max_track; /*store number of tracks used */
return 0;
}
/*** end compact ***/
/*****
/*expand tries to move track violations to valid locations */
expand(int reset[MAX_NETS], int lyr_ck[3][TRAX][TERMS], int sol_num) {
    int netnum,
        l_pt,r_pt,l,t,
        max_track,
        ok,v,
        i,j,k;

    for(i=1;i<=reset[0];i++){
        v=reset[i];
        netnum = netlist[v].net_num;
        l_pt = netlist[v].strt_pin;
        r_pt = netlist[v].end_pin;
        max_track = sol[sol_num][0].track;
        for(t=0;t<=max_clique+4;t++) { /*find a layer and track to use*/
            for(l=2-layers;l<=layers;l=l+2) {
                ok = ck_space(sol_num,netnum,lyr_ck,l,t,l_pt,r_pt);
                if(ok) goto newnet;
            }
        }
        newnet:
        if(ok) {
            if((vhv) && (layers == 2)) {
                for(j=l_pt;j<=r_pt;j++) {
                    lyr_ck[1][t][j] = netnum;
                    if(b_in[j] == netnum) for(k=0;k<=t;k++) lyr_ck[1][k][j] = netnum;
                    if(t_in[j] == netnum) for(k=t;k<max_track;k++) lyr_ck[1][k][j] = netnum;
                }
            } else {
                for(j=l_pt;j<=r_pt;j++) {

```

```

        lyr_ck[1][t][j] = netnum;
        if(b_in[j] == netnum) for(k=0;k<=t;k++) lyr_ck[layers-1][k][j] = netnum;
        if(t_in[j] == netnum) for(k=t;k<max_track;k++) lyr_ck[layers-1][k][j] = netnum;
    }
    sol[sol_num][v].track = t;
    sol[sol_num][v].layer = 1;
}
}
return 0;
}
/*cost calculates a cost associated with the possible solution */
cost(int sol_num) {
    int netnum,
        max_track,
        lyr_ck[3][TRAX][TERMS],
        l_pt,r_pt,
        l,t,
        sol_cost,
        trk_err,
        via_err,
        i,j,k,
        known_prob;

    for(i=0;i<=2;i++) //clear matrix to check placement.
        for(j=0;j<TRAX;j++)
            for(k=1;k<TERMS;k++) lyr_ck[i][j][k] = 0;
    sol_cost = trk_err = via_err = 0;
    max_track = sol[sol_num][0].track;
    for(i=1;i<=total_seg;i++){ //populate matrix and check for violations
        netnum = netlist[i].net_num;
        l_pt = netlist[i].strt_pin;
        r_pt = netlist[i].end_pin;
        l = sol[sol_num][i].layer;
        t = sol[sol_num][i].track;
        if((vhv) && (layers==2)) {
            known_prob = 0;
            for(j=l_pt;j<=r_pt;j++)
                if((lyr_ck[1][t][j] > 0) && (lyr_ck[1][t][j] != netnum)) {
                    if(! known_prob) {
                        trk_err++;
                        known_prob = 1;
                    }
                } else lyr_ck[1][t][j]=netnum;
            known_prob = 0;
            for(j=l_pt;j<=r_pt;j++) {
                if(b_in[j] == netnum) {
                    for(k=0;k<=t;k++)
                        if((lyr_ck[1][k][j] > 0) && (lyr_ck[1][k][j] != netnum)) {
                            if(! known_prob) {
                                via_err++;
                                known_prob = 1;
                            }
                        }
                } else lyr_ck[1][k][j]=netnum;
            }
            if(t_in[j] == netnum) {
                for(k=t;k<max_track;k++)
                    if((lyr_ck[1][k][j] > 0) && (lyr_ck[1][k][j] != netnum)) {
                        if(! known_prob) {
                            via_err++;
                            known_prob = 1;
                        }
                    }
                } else lyr_ck[1][k][j]=netnum;
            }
        }
    } else {

```

```

known_prob = 0;
for(j=l_pt;j<=r_pt;j++)
  if((lyr_ck[l][t][j] > 0) && (lyr_ck[l][t][j] != netnum)) {
    if(! known_prob) {
      trk_err++;
      known_prob = 1;
    }
  } else lyr_ck[l][t][j]=netnum;
known_prob = 0;
for(j=l_pt;j<=r_pt;j++) {
  if(b_in[j] == netnum) {
    for(k=0;k<=t;k++)
      if((lyr_ck[layers-1][k][j] > 0) && (lyr_ck[layers-1][k][j] != netnum)) {
        if(! known_prob) {
          via_err++;
          known_prob = 1;
        }
      } else lyr_ck[layers-1][k][j]=netnum;
    }
  if(t_in[j] == netnum) {
    for(k=t;k<max_track;k++)
      if((lyr_ck[layers-1][k][j] > 0) && (lyr_ck[layers-1][k][j] != netnum)) {
        if(! known_prob) {
          via_err++;
          known_prob = 1;
        }
      } else lyr_ck[layers-1][k][j]=netnum;
    }
  }
}
}
}
}
sol_cost = max_track + (4 * trk_err) + (6 * via_err) + (sol[sol_num][0].layer * 2);
return sol_cost;
}
/*** end cost ***/
/*****
/* via_count marks the matrix with via locations and then counts */
/* the quantity needed for the solution. */
via_count(int sol_num) {
  int lyr_ck[3][TRAX][TERMS],
  netnum,
  l_pt,r_pt,
  l,t,
  vias,
  i,j,k;

  for(i=0;i<=2;i++) /*clear matrix to check placement.*/
    for(j=0;j<TRAX;j++)
      for(k=1;k<TERMS;k++) lyr_ck[i][j][k] = 0;
  for(i=1;i<=total_seg;i++) {
    netnum = netlist[i].net_num;
    l_pt = netlist[i].strt_pin;
    r_pt = netlist[i].end_pin;
    l = sol[sol_num][i].layer;
    t = sol[sol_num][i].track; /*mark via locations*/
    for(j=l_pt;j<=r_pt;j++) if((b_in[j]==netnum) || (t_in[j]==netnum)) lyr_ck[l][t][j] =
1;
  }
  vias = 0;
  for(i=0;i<=2;i++) /* Count vias used*/
    for(j=0;j<TRAX;j++)
      for(k=1;k<TERMS;k++) if(lyr_ck[i][j][k] == 1) vias++;
  return vias;
}
/*** end via_count ***/
/*****
/*mutate switches nets' locations to overcome local minimums */
mutate(int prt_num,int sol_num) {
  int net_1,net_2,

```

```

    temp_l,temp_t,
    not_ok,check,
    gain,
    i;

check = 1;
do {
    for(i=0;i<=total_nets;i++) {
        sol[sol_num][i].layer=sol[prt_num][i].layer;
        sol[sol_num][i].track=sol[prt_num][i].track;
    }
    i = random(2);
    if(i--0) {
        net_1 = random(total_seg)+1;
        sol[sol_num][net_1].track = max_clique + 1;
    } else {
        not_ok = 1;
        do {
            net_1 = random(total_seg)+1;
            net_2 = random(total_seg)+1;
            if(net_1 != net_2) not_ok = 0;
        } while(not_ok);
        temp_l = sol[sol_num][net_1].layer;
        temp_t = sol[sol_num][net_1].track;
        sol[sol_num][net_1].layer = sol[sol_num][net_2].layer;
        sol[sol_num][net_1].track = sol[sol_num][net_2].track;
        sol[sol_num][net_2].layer = temp_l;
        sol[sol_num][net_2].track = temp_t;
    }
    compact(sol_num);
    sol[sol_num][0].layer = via_count(sol_num);
    gain = solcost[prt_num] - cost(sol_num);
    if(gain > (random(0-p) + p + 1)){
        check = 20;
        solcost[sol_num] = solcost[prt_num] - gain;
    }
    check++;
} while(check < 20);
return 0;
}
    /*** end mutate ***/
/*****
update(int c1, int c2) {

    if(c1 < c2) p--;
    else p = p0;
    return 0;
}
    /*** end update ***/
/*****
sort_them () {
    int i,j,p,                /*loop counters */
        temp,                /*temp storage  */
        done;                /*loop check    */

    if(! elite) {
        i = random(3);        //occasionally selects a random 5 as parents
        if(i==0) {
            for(i=0;i<num_prt;i++) saved[i] = 0;
            for(i=0;i<num_prt;i++) {
                temp = random(num_prt * (num_cld + 1)) + 1;
                saved[i] = temp;
            }
        }
    }
}
p = num_prt-1;
do {
    done = 0;
    for(i=0;i<p;i++) {

```