A Heuristic Algorithm for Automated Routing in Integrated Circuits

by

(Amir) Hossein Sepasi

Master of Arts (Economics), University of British Columbia, 1999
Bachelor of Science, Sharif University of Technology, Iran 1990

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENT FOR THE DEGREE OF
MASTER OF SCIENCE IN BUSINESS
ADMINISTRATION

in

THE FACULTY OF GRADUATE STUDIES

(Sauder School of Business)

We accept this thesis as conforming
to the required standard

THE UNIVERSITY OF BRITISH COLUMBIA

October 2003

Library Authorization

Name of Author *(please print)* HOSSEIN (AMIR) SEPASI	Date 01/10/2004

Title of Thesis: A HEURISTIC ALGORITHM FOR AUTOMATED ROUTING IN INTEGRATED CIRCUITS

Degree: MASTER OF SCIENCE Year: 2003

ABSTRACT

In recent years artificial intelligence techniques have been extensively applied to real world problems. Two common characteristics of the problems amenable to heuristics are: their large search space that makes exhaustive search practically impossible; and for most applications the need to find a near-optimal solution – instead of exact – in a reasonable and cost effective time frame.

Most of the problems throughout the design of Very Large Scale Integration (VLSI) circuits are combinatorial, which implies a huge search space. Furthermore the sheer number of components makes exhaustive search impossible. One of the common problems in this area is routing the terminals in the layout in a reasonable amount of time. Until very recently, manual routing was the most reliable way of doing routing. As the scale of integration grows, manual routing becomes practically impossible. There has been lots of research and studies on how automated routing can successfully replace manual routing. The purpose of this thesis is a to find an automated method that helps find a good solution to this problem in a reasonable amount of time.

CONTENTS

LIST OF FIGURES AND TABLES

Acknowledgment

This thesis is the outcome of a project from Intel Corporation that was assigned to the Center for Operations Excellence (COE) in the Sauder School of Business (the former Faculty of Commerce) at the University of British Columbia. Steve Duvall was the contact person from Intel Corporation who offered the project to the department and convinced that company to support it and to provide the financial grant. In this respect, I am highly grateful to him that made this thesis possible.

Many thanks to my supervisor, Tom McCormick, who provided invaluable support from the very beginning till the end of the project. It was a problematic project and to some extension risky, because the definition, finding a good solution, and even the duration of the project were all uncertain and unclear. I am grateful to his supervision, patience, and support.

Maurice Queyranne provided insights to the theoretical side of my thesis. It is usually not easy to understand the relationship between theory and practice; and he helped me to structure what I am developing in practice. In this respect, I am thankful to him.

I also like to thank Stephen Jones and Martin Puterman for their effort to make COE a pleasant and exciting place to research. Especially integrating projects with the materials in Consulting Practice course was a new way of looking at the school projects from business perspective.

OUTLINE

Section 1 and 2 provide an introduction of the subject, history of the project, motivation, and literature review. Section 3 is dedicated to defining the problem that we attempt to solve; and describes the data set that the developed algorithms were tested on. Section 4 briefly explains a previous work on the data set and its deficiencies in solving the problem. Section 5 and 6 discuss a heuristic idea and introduces two algorithms, which based on, the solution has been developed. Section 7 discusses why the developed algorithms are fast and memory usage efficient, and provides a few tips for more improvements. Section 8 concludes the discussion and provides hints for further research.

1. INTRODUCTION

Until very recently, most circuit designs were performed manually by design engineers, using schematic editors and circuit simulators. With increasing magnitude of circuit design problems, there has been strong tendency toward automation. However, this move has turned out to be not easy, especially from computational perspective. The size of the problems, the computational expense of analysis tools, and the non-linearity of the objectives and constraints pose significant challenges to the design engineers. Among circuit design problems, routing assignment seems quite challenging.

Routing assignment concerns all the communication paths that connect the logic cells within integrated circuits. A cell can be as small as a logic gate (e.g. NAND, AND, OR, NOT gates), intermediate logic cells (e.g. shift registers, multiplexers, memory registers) or as big as a complete functional block (e.g. memory units, arithmetic unit). In each level of hierarchy a routing problem is defined as connecting the associated terminals of the cells in that level of hierarchy. If two or more terminals are designed to interconnect then they are logically connected. A group of terminals that are logically connected is called a **net**. A router's assignment is to determine the route of these logical connections, i.e. determining the exact location of the wires that connects the associated terminals. Hence, if two or more terminals are logically connected then they must also be physically connected and vice versa; i.e. if they are physically connected then they have to be a net (logically connected too). A net that is routed is called a **routed net**. Some routers let that during the routing two or more nets touch or cross each other – called *dirty routing* - but, at the end of routing, a valid routing solution is that no two routed nets can touch or cross each other; because if they do then they become one net, which is wrong – and considered a short in electrical engineering term. We will explain the problem in more detail in section 3.

Efficient routing plays a key role in the design of high performance microprocessors. Optimization of wiring is essential for reducing the size of the chip, decreasing the heat dissipation, reducing signal propagation delay, and increasing the efficiency of design. An automated router that produces efficient routing in a reasonable amount of time is the main challenge of routing problems.

Most of the researches found in the literature employ customized versions of classic combinatorial optimization algorithms (e.g. linear programming, shortest path, etc.). These algorithms are still used by many software codes that semiconductor industries use in automated routing. However, with immense increase in density of integrated circuits, those prescribed solutions are becoming prohibitively time consuming for many of the newly faced problems. These algorithms mainly perform poorly with the new design specifications. The objective of this thesis is an effort to develop an efficient algorithm for automated routing.

The thesis has been written based on the result of a project – started in September 1999. The project was a contract between Intel™ Corporation and Center for Operations Excellence (COE) at the Faculty of Commerce and Business Administration in UBC. It is a continuation of an earlier work, which had been done at Cornell University (98-99) [1]. In Cornell, a customized version of Dijkstra's algorithm was developed that was applying that algorithm to routing of VLSI circuits. The outcome of that project was unsatisfactory from the points of view of both routing completion rate (number of nets successfully routed) and running time. This thesis pursues an alternative heuristic approach.

The project is in two phases. In the first phase we construct a heuristic framework, which is called Pattern Assignment Method (PAM). As the name of PAM suggests, it tries to find a feasible pattern to route each net. It looks for simple patterns so it can do a fast feasibility check. The idea behind PAM is that "the simplest route, if feasible, is the most efficient one", and so is the best choice for a net. PAM is a preliminary prototype of our heuristic approach. The results of our tests show that the routing speed significantly improves. Completion rate did not improve. Also PAM, as we will see in section 4, is mainly a *path* routing algorithm, meaning that it finds a route between two terminals only (a path). Although it can be extended to work on multi-terminal nets too, but in practice it is not capable of giving a near optimal route to those nets. From our project's point of view this was not an important concern because the majority of the nets in the given data were two terminal nets. Therefore, our primary focus was to do a routing assignment for paths only. A good router, however, has to have the flexibility to route nets with more than two terminals, so we addressed this problem in our next algorithm too.

In the second phase of the project we focused on improving the PAM algorithm to increase the completion rate, gain more speed, and address multi-terminal nets. PAM is used as the building block of the second algorithm, which is called Segment Utilization Method (SUM). SUM follows the same philosophy of PAM, i.e. it looks for simple feasible patterns that route a net but it uses a better data structure to improve the speed even more; and has a refined objective function that increases the completion rate. SUM algorithm turns out to be a reasonably good solution for the given data. It is also able to route the multi-terminal nets and provides near optimal solutions for them. Because SUM is a fast algorithm it lets us to sacrifice some time to reach an even better completion rate. An added feature to SUM, which is called revised-SUM (or r-SUM), gives us a better completion rate with some sacrifice in time. This thesis is mainly the description of these three algorithms, PAM, SUM, and r-SUM.

Table-A shows the results of running each algorithm on two given data sets[1] (provided by Intel™ Corporation – refer to Figure set 3.4). As the table shows, PAM is significantly faster than Dijkstra's Algorithm. The completion rate, however, is not acceptable (it is worse). SUM and r-SUM improved both the completion rate and running time.

	343 nets		5510 nets	
	Search Time	**% Assigned**	**Search Time**	**% Assigned**
Dijkstra-based[2]	5 hours	100 %	?> 132 hrs.	(≈ 74 %) ?
P.A.M.	4.3 minutes	99 %	2.47 hrs.	63 %
S.U.M.	9 Seconds	100 %	10 min.	77.6 %
S.U.M. (revised)			21 min.	80.3 %

Table – (A) outcome of running the four algorithms on Intel's data

[1] The first set is merely the first 343 trees of the whole BCD data, i.e. 5510 routing trees. The size of the layout and the number of layers are the same as the mother set.

[2] Dijkstra's Algorithm didn't finish on the larger test case because of memory crash. We didn't bother to resolve the problem because the result was not good anyway.

During development of the project, a major limitation that we faced was lack of a variety of data sets to do more testing of our algorithm. The more data we had the better we could test the efficiency and/or weakness of the algorithm. One solution to this could be testing the algorithm on random data. However, the result would not be an accurate measurement of the efficiency of the algorithm in the real world. The structure of VLSI is so that - in practice - the distributions of terminals in the layout follows some patterns. In fact, it is the existence of these patterns that heuristic algorithms can take advantage of, and which make them more successful than combinatorial algorithms, such as Dijkstra's. In the end we had two only data sets to test our algorithms on. The positive point, however, is that the bigger data set - which the project had mainly been involved with - has been a very challenging data set so that both our customer Intel and the previous works could not successfully fully route it. Our relatively successful results on the second data set supports the worth of our algorithms. These two data sets are not enough to definitively establish the overall efficiency of our algorithms, but they do show that our approach is promising. Because of time pressure, we had to close this thesis at some point, but hopefully the project will continue in future and more test and/or improvement will be made.

We brought a fresh approach to the routing problem by improving the previous ideas that led to development of a new algorithm. There are three phases in development of the solution: (i) inspection of the previously developed algorithm for its capabilities, limitations, and performance, (ii) development of a new algorithm, and (iii) validity test.

The development strategy for this project is mainly a heuristic search. In short, the approach tries to compensate for slowness of classic procedures by combining them with a heuristic search.

2. LITERATURE REVIEW

A microprocessor typically consists of a few functional blocks that communicate with each other to execute the instruction codes that are sequentially fetched from the computer memory. Integrated circuits are implemented by fabrication of layers of silicon wafers on top of each other. In the older technologies, there were spaces among the functional blocks, and the blocks were interconnected in those spaces. Those spaces - which are categorized in two groups and called *channel*s and *switch box*es - are provided for the routing purpose –Figure 2.1. *Channel* refers to a rectangular routing area with fixed pins (terminals) on the two parallel sides of the rectangle. The other two sides of the rectangle may have transient pins, pins that do not have a fixed location initially. Their exact location will be determined later when the whole channels are routed individually. *Switch box* refers to a rectangle routing area with fixed pins on all four sides of the rectangle. In both cases the inside area of the rectangle is clear and dedicated to routing wires.

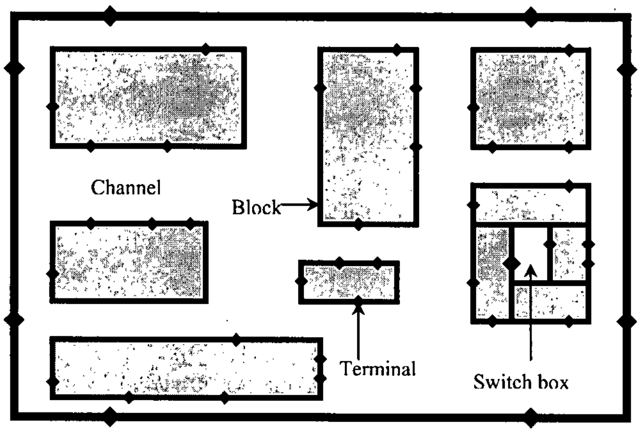


Figure 2.1 – The old layout of circuit blocks and terminals after placement. There are spaces between functional blocks (channels and switch boxes), which are used for routing

Channel routing and switch box routing cause a lot of waste in the fabrication of wafers. Generally, in fabrication, the routing area is considered as waste because they play no role in functionality of a chip, while the functional blocks do. With advancement in integration technology it is now possible to develop more layers on top of each other. So, it is feasible now to use the whole area of the chip for functional blocks (that means more integration, and hence, more provided functionality) and do the routing on the upper layers. Therefore, the routing area now consists of a few layers (for example three) , and wiring mainly occurs in those layers. They will be laid on top of a few other layers. The bottom layers are used for placement of the functional cells. Figure 3.2 shows a high-level schematic of a typical microprocessor. The arrows show hypothetical communication among the blocks.

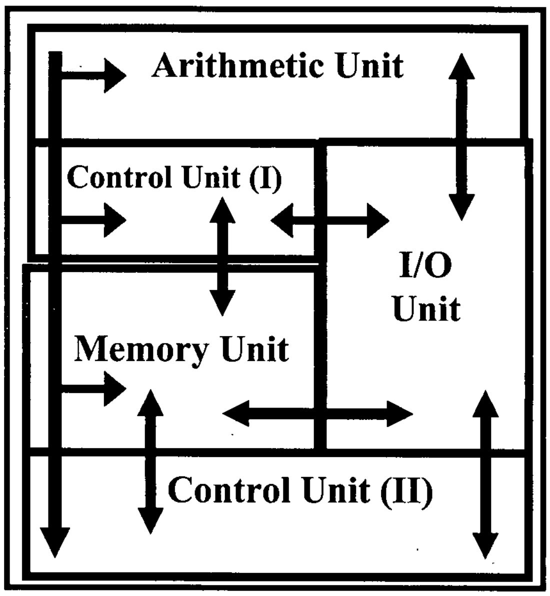
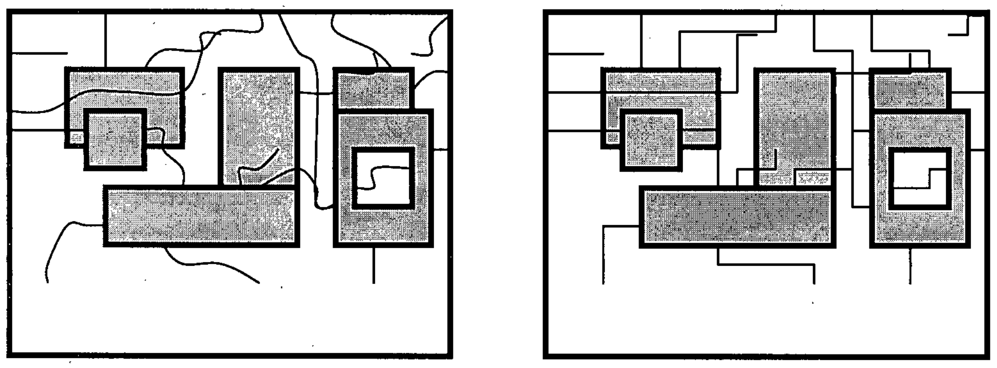


Figure 2.2 – A high level schematic of newer design of microprocessors. There is no channel left between the blocks. All the routings happen on the upper layers above the layers where the functional blocks are placed.

Routing of a given net can be decomposed into two phases: global routing and detailed routing [2] – Figure 2.3. The global routing phase is a higher-level process during which the routing topologies of the nets are defined in the available routing regions. Then, the detailed routing produces the actual geometry, which realizes those required connectivity in the fabricated chips [3]. This thesis dedicated to the detailed routing phase of a physical design.



(a) Global Routing **(b) Detailed Routing**

Figure 2.3 – The two phases of routing, i.e. global routing and detailed routing, represents a *hierarchical* routing strategy. First, the blocks are routed globally without any decision on the details. Next, a detailed routing fixes the exact position of the wires.

VLSI design rules dictate minimum spacing between wires, and therefore the area occupied by the routing on a chip is roughly proportional to the total wire-length of the routing. The increase in length of wire generally increases signal delay and power consumption because of increased resistance and capacitance. It also increases fabrication area. Other concerns are increase of fabrication cost, decrease of yield and reliability that increase with the fabrication area. Thus, a fundamental objective is to minimize the total wire-length [3]. In the literature, *minimum spanning tree* is introduced as a minimal tree that connects n nodes on a plane and has the shortest possible length. In the case of routing, it is sometimes possible to add one or more additional nodes to construct even a shorter tree. The extra nodes added to the plane are called the *Steiner nodes* and the shortest possible tree with these extra nodes is called the *Steiner Minimal Tree*. Since routing, in this thesis and in general, is based on a rectilinear (Manhattan) grid, we only consider rectilinear Steiner trees. The ideal solution to any routing problem is a Minimum Rectilinear Steiner Tree (MRST). The MRST problem and, in general, Steiner Minimal Tree problem is central to VLSI routing and wire length estimation and is well studied in combinatorial optimization and network design [5, 6, 7]. It has been proved that the general problem of constructing Minimal Rectilinear Steiner Trees (MRST) is a *NP-complete* problem [4]. Routing is an extensive studied problem and several hundred articles have been published on its different aspects [8]. Since almost all problems in routing are *NP-complete*, so computationally hard, the researchers have focused on the alternative heuristic algorithms (e.g. chapter two of [3] has a good study of some heuristics for MRST. Also, Joobbani [6] provides some sources for heuristic search of MRST). Due to the nature of the routing algorithms, complete routing of all the connections cannot be guaranteed in many cases. As a result, a technique called *rip-up and reroute* is used, which basically removes troublesome connections and reroutes them in a different order [8]. We have not used this technique in our algorithm. One reason is that it is beyond the scope of this thesis. The purpose of this thesis is not development of a complete CAD[3] tool for complete routing, but only the design of a heuristic algorithm that does part of the job - i.e. detailed routing. It is plausible though, that the *rip-up and reroute* technique and our algorithm (and maybe a few other sub-tools) can be mixed together to provide a complete routing system. Moreover, using the *rip-up and re-routing* technique needs some

[3] Computer Aided Design

information feedback from the manufacturer about the nature of the terminals; their functionality, and their relative importance, which we didn't have access to.

3. PROBLEM STATEMENT

Before describing the statement of the assignment we have to define and explain a few terms.

Net: A group of terminals that are logically inter-connected is called a **net**. A *net* that is routed is called a **routed net**. A net can be partially routed if not all of its terminals are connected. A *net* that contains no cycle (meaning that two terminals are not connected in more than one way) is called a **tree**. A tree with only two terminals is also called a **path**. For ease of use *path* and *tree* might also be used to refer to their un-routed nets. We assume that no routed net will end up having a cycle so the two terms, *tree* and *routed net*, are interchangeable.

Layers: Terminals are routed in a multiple number of layers. In our given data there are three layers available for routing purpose. Two different nets on the same layer cannot cross each other but different nets on different layers can run over each other without causing a short. Two of the layers (upper and lower) are dedicated to the vertical direction and the middle one is dedicated to the horizontal direction. The reason that each layer carries only parallel lines is a rule of thumb that heuristically minimizes blocking. It is also easier for algorithm development. The reason that two layers are dedicated to the vertical direction, in our special case, is that the given data is more vertically distributed than horizontally (distances between the terminal pairs are more vertical that horizontal), so more vertical lines are needed to connect them.

Segment: is a piece of line, either horizontal or vertical, which represents a piece of straight wire. A routed net consists of one or more pieces of connected segments.

Via: Two segments on different layers can be connected using via. Via are placed in the overlapped grid point of two segments that belong to the same net on two different layers. If a via connects two different nets it is a short.

Grid: In *Manhattan* geometry only rectilinear lines are allowed [9]. In the our terminology, the distance between two terminals refers to a *Manhattan* distance. The routing area is divided into an equally distanced grid (i.e. matrix of points). The horizontal lines are referred to as *rows* and the vertical lines are referred to as *columns*. A line that connects two adjacent grid points is called an **edge**.

Length: **length** of a net (or similarly length of a path or a tree) is the total length of segments (or equivalently total number of edges) that is used to connect a net.

Cost: The *length* of a path plus the number of vias that is used to route a net is the **cost** of that route. The cost of via is set to be equal to the cost of one edge. Note that vias are costly to fabricate and are often points of failure, so avoiding too many vias is important. This might suggest that we should set a higher cost to via insertion. In our algorithm this is not necessary because via is inherently minimized in the algorithm by avoiding too many turning points in the routing pattern. As we will see we need the same cost as the edge for via to be able to decide on the best path in advance.

Routing: In abstract terms, the routing problem is to find an algorithm that routes a set of source points to a set of destination points, through a series of connected segments - see Figure 3.1.

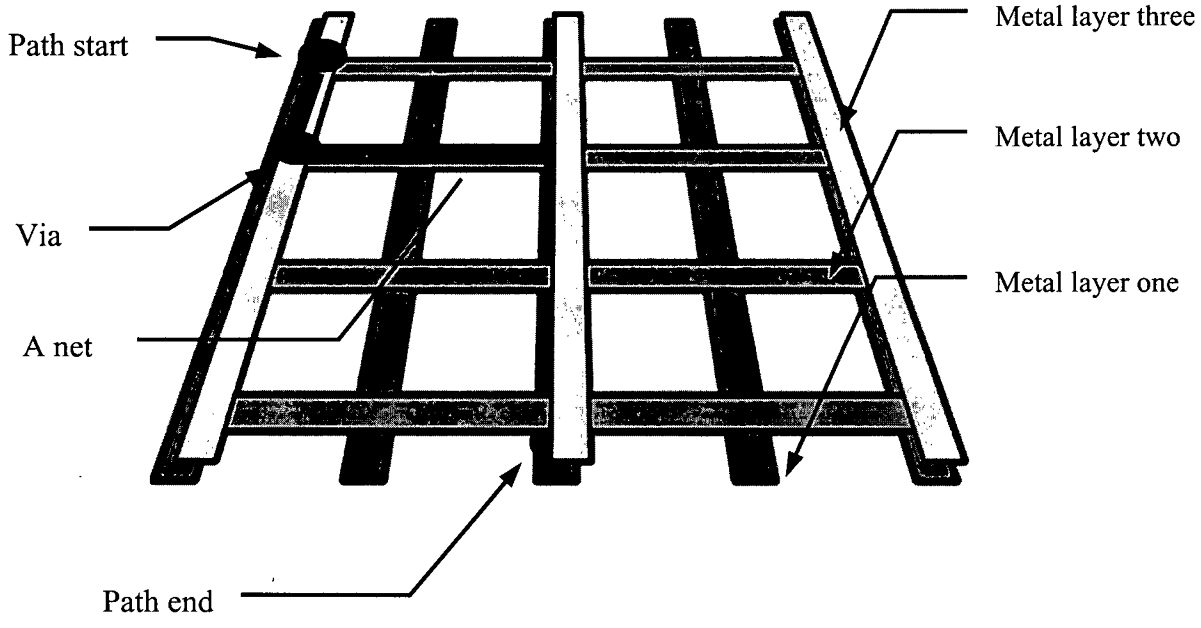


Figure 3.1

In the context of integrated circuits, i.e. VLSI design, *routing* is defined as following: Given a rectangular area with a number of terminals inside or in the border of the area, connect all the terminals that belong to the same net subject to the following constraints or rules [9].

1) A net is supposed to be routed in a multi-layer rectilinear grid network so that its segments lie only on grid points. In our given data, the grid matrix consists of three levels: the top and bottom are used for vertically aligned wires, while the middle level is used for horizontally aligned wires – Figure 3.1. The three levels can be connected by either a 1-level via (used for 90° turning angles) or a 2-level via (used to move from top layer to bottom layer or vice versa).

2) Different nets cannot occupy the same grid point (considered as electrical short).

3) If a 2-level via is used then the middle grid point cannot be used by another net.

An ideal router should have the following qualities:

1. **Do 100% routing**. The manual routing is time consuming and often involves rerouting of most if not all of the routed nets. So, it is important for a router to complete, meaning that to route all of the nets in the cell. However, in many cases it is very hard to reach 100% routing [8]. Therefore the completion rate is a major factor in design of routing algorithms.
2. **Minimum area.** It is desirable to use the least space for routing. The less area is used the more net can be included to the routing area, meaning that higher utilization and more integration is possible.
3. **Least wire length.** The shorter the length of the route the smaller the propagation delay, the less heat dissipation, the better quality, the better chips.
4. **Least number of vias.** The introduction of via between the two layers means longer propagation delays, and lower fabrication yield. The fewer the number of vias the better the quality of routing.

The above four factors are generally used to measure the quality of routing.

Figure set [3.2] shows an example that depicts the segments and via of two routed nets in three layers.

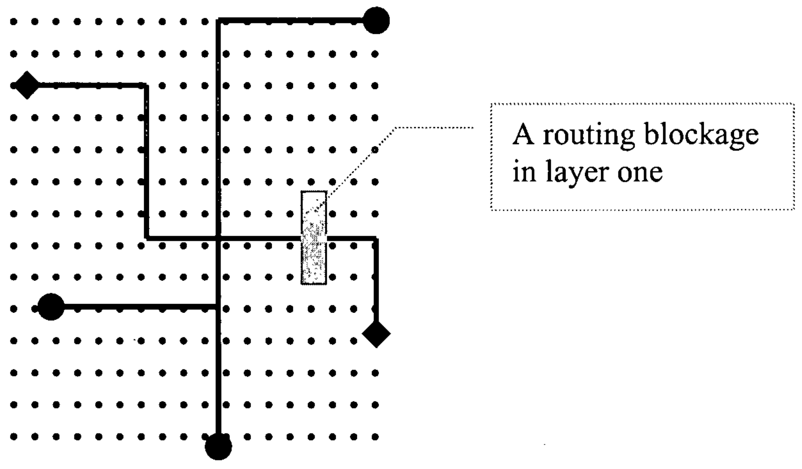


Figure 3.2.1 - A two-dimensional presentation of two sets of associated terminals (one path and one tree). In this example a 'W' pattern[4] connects the square terminal set and a tree connects the circle terminal pair. A blockage[5] also exists in layer one.

[4] We define the meaning of patterns later in description of Pattern Assignment Method. However for sake of argument, a pattern is a path that its name visualizes its shape. For example a 'L' pattern is two perpendicular, connected segments, 'Z' pattern is three connected segments, 'W' pattern four connected segments, etc.

[5] A *blockage* can be any physical block that must be avoided, either another path or a functional block.

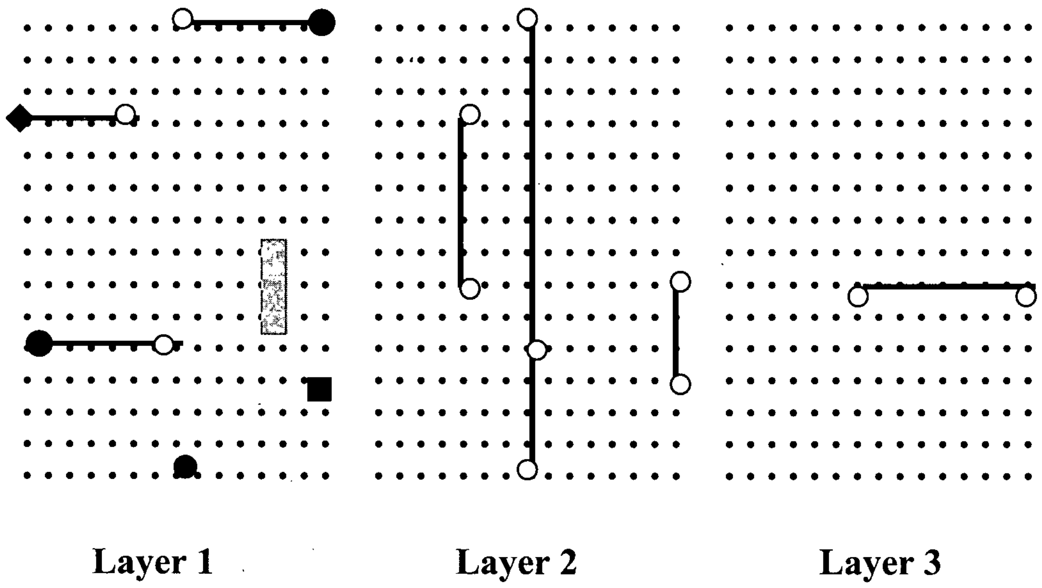


Figure 3.2.2 - Three-dimensional presentation of Figure [3.2.1]. Parallel segments are located in the same layers (Layer 1 and 3 are dedicated to horizontal segments and layer 2 to vertical segments). The hollow circles represent the connecting via. The blockage in layer 1 does not let any crossing segment passes through it, so it has to route through layer 3.

As mentioned before, we consider that the cost of using one more via is equal to the cost of using one more edge. Assuming so, the layout network will be a symmetric three-dimensional network, as shown in figure 3.3.

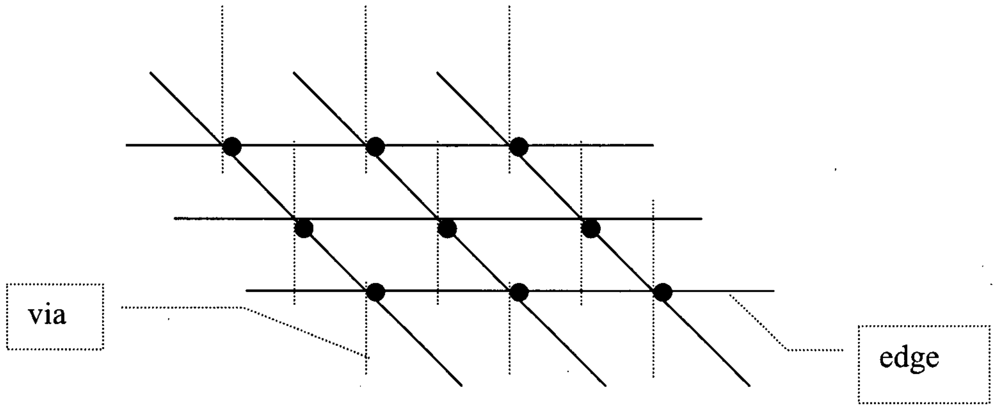


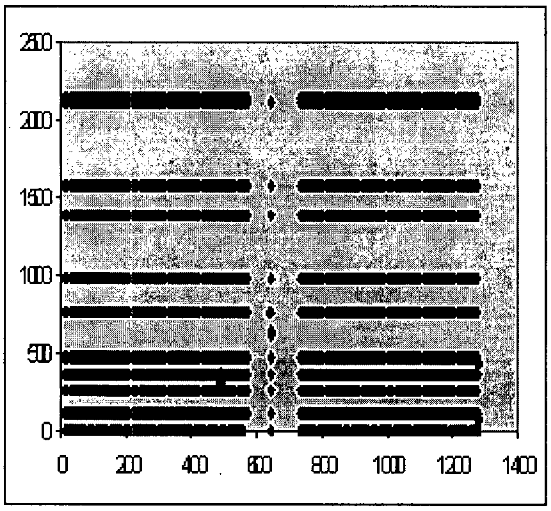
Figure - 3.3

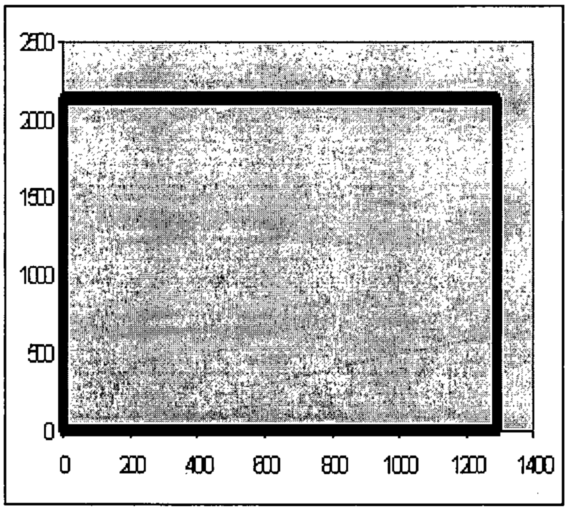
The algorithms in the project are tested on two real data sets that are provided by Intel™. The first - and the most challenging one - is called [illegible] data. Figure 3.4 shows the distribution of the terminals. It contains 11,692 terminals. The data consists of 5510 nets: a set of 4954 two-node nets (paths)[6], 498 three-node nets, 57 four-node nets, and one five-node net. There are 5510 source terminals (shown in Figure 3.4a) and 6182 destination terminals (shown in Figure 3.4b). All of the terminals are located in layer three (the lower layer - the nearest layer to the functional blocks).

As the numbers show, the majority of the nets have two terminals, i.e. the net is a path. This suggests that during the development of the algorithm we could focus on routing the paths only and still get a high completion rate. Based on this observation, we developed PAM., which is a poor algorithm for routing multi-node nets. However, SUM provides a rather good solution to the multi-node nets. The result of running SUM on our data showed that on the average it routed

[6] Throughout the text we use 2-node tree and *path* interchangeably, unless otherwise mentioned.

the multi-node nets near to optimality,. We will return to this point later in section 6.1.b when we demonstrate the algorithm.





(a) Source terminal distribution **(b) Destination terminal distribution**

Figure 3.4 - Figures [a] and [b] show the distribution of the BCD data set. As shown, the source terminals are distributed inside the layout area (Figure [a]) and destination terminals are distributed in the boundary (Figure [b]). Each source terminal is associated with one or more destination terminals in the boundary.

As mentioned, all nets have terminals in the boundary. It shows that the BCD data is only one block and the terminals inside the area (the source terminals) are supposed to communicate with the outside of the block. Figure 3.5 shows an example.

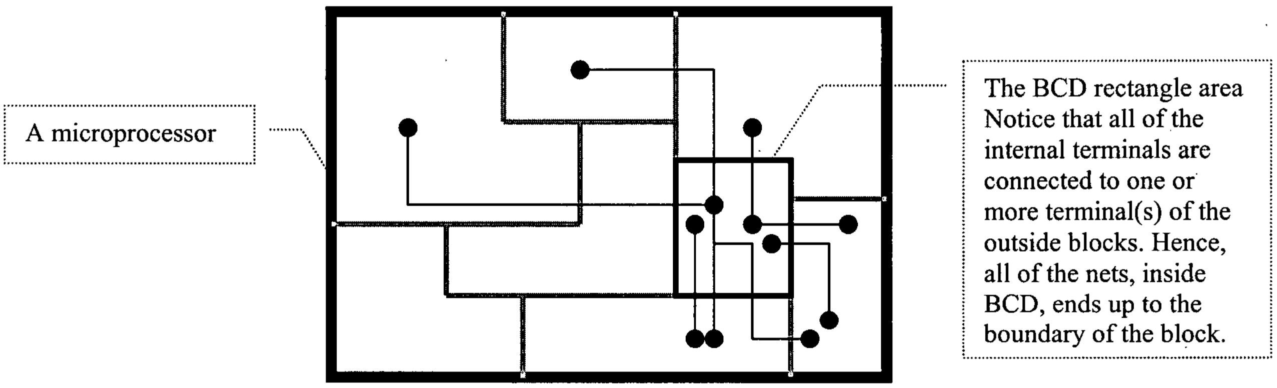

A microprocessor
The BCD rectangle area Notice that all of the internal terminals are connected to one or more terminal(s) of the outside blocks. Hence, all of the nets, inside BCD, ends up to the boundary of the block.

Figure 3.5 – An example that explains why all nets of the BCD data terminate at the boundary. Boxes represent functional blocks within a chip.

The second data set, also provided by Intel, is shown in Figure 3.6. As seen, the terminal density is less than the BCD set. There are only 44 trees (186 terminals) to be routed in an area widened by (2737 x 870 =) 2381190 grids in three layers. Comparing it with the BCD data shows that we should have fewer problems with routing it. In fact, our algorithm has been able to route this data set completely in a few seconds.

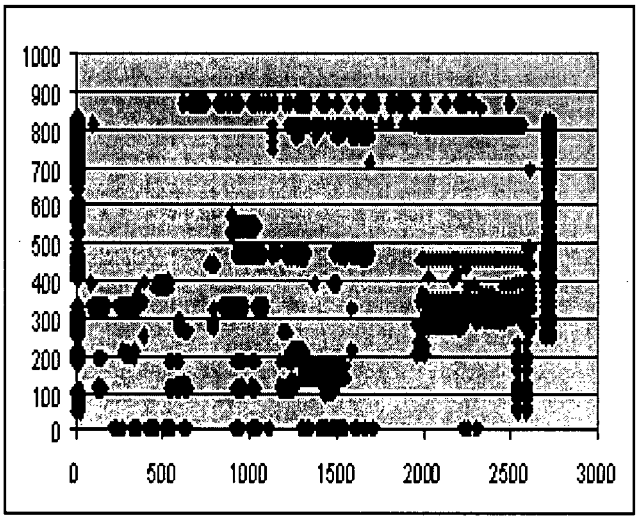


Figure 3.6 - The map of the terminal distribution of the second data. It contains 44 nets to be routed. 11 two-node nets, 8 three-node nets, 8 four-node nets, 14 six-node nets, and 3 eight-node nets.

In the second data set the terminals are more than one grid[7]. Therefore, there is more room for routing the net. Figure 3.7 explains the situation for a 2-node tree.

[7] The study of terminal grouping as provided by the second data set, and its more flexible property, is beyond the subject of this thesis. For one research work on this subject refer to [11].

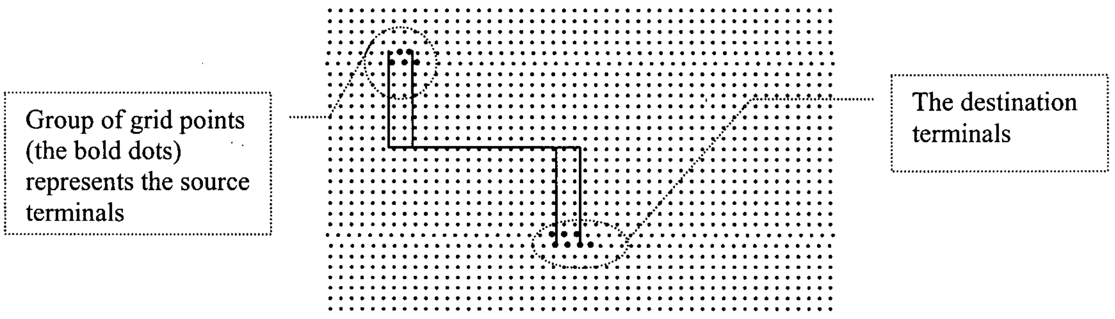

Group of grid points (the bold dots) represents the source terminals
The destination terminals

Figure 3.7 - An example of terminal group in the second data set. Routing is more flexible than the case of one-grid-terminals.

4. EXPERIMENTING WITH DIJKSTRA'S ALGORITHM IN A ROUTING PROBLEM

In this section we briefly explain the customized version of Dijkstra's Algorithm that had primarily been used to route our data. Dijkstra's Algorithm is a famous combinatorial algorithm that has many applications in networking and theoretical routing problems. However, the results in table (A) imply that it has not been a practical tool for solving large scale problems.

We don't intend to explain Dijkstra's Algorithm in detail, but only the way that it has been used in our case. For more detail of the algorithm one may refer to numerous books in network optimization [10]. In the following, first we explain the customized algorithm, and how it attempts to solve our routing assignment and then we explain why it is not a good approach to attack this assignment and what are the pros and cons of using an alternative heuristic approach.

A Dijkstra-based algorithm

Before this project starts in UBC, in a similar project in Cornell University, an algorithm that uses the Dijkstra's Algorithm was implemented and tested on our data [1]. For this specific application Dijkstra's Algorithm had to be modified. Therefore, what is implemented does not reflect the exact form of Dijkstra's Algorithm.

The implemented Dijkstra-based algorithm is a two-stage procedure. In the first stage, the procedure sets fixed costs to each net for the use of the horizontal metal layer, and two top and bottom vertical layers. There are also costs assigned for changing directions (equivalent to a one-level drop either from vertical wiring to horizontal or vice versa), and for changing layers while maintaining a constant vertical direction (2-level via). Lastly, for every wire that has already been routed through a given vertex (grid point), a congestion cost (multiplied by the level of congestion) is assigned to that vertex when a new wire is routed through that vertex. With this cost structure in place, a shortest path can be found between each terminal pair. The program runs through each pair sequentially; and for that pair performs the Dijkstra's Algorithm [10] on the reduced grid, formed by the bounding box of the source and destination (plus a buffer region). As it runs, it keeps the record of the cost of each path (and the total cost thus far) plus the total number of conflicts that have been occurred. In the second stage, it throws away all paths found in stage one and keeps the computed congestion costs of each vertex. Then it redoes the first stage using the new cost. Because the algorithm does not mark a vertex permanently labeled with a net until it has checked all possible routes to that node, it is guaranteed to find the cheapest path between any pair by avoiding already congested regions.

The above algorithm is an example of a combinatorial shortest path solution (integer programming model). It optimizes an objective function subject to a few constraints. The objective function is to find a set of non-crossing paths from the source to the destination, and the goal is to minimize the sum of the *cost* of individual paths (so the accumulated *costs*). The constraints are represented by the structure of the network, meaning that the paths are confined to route through a three-dimensional grid network.

The first row of the table (A) in page 3 shows the results of our different tests on the two given data sets. In all the four algorithms the running time of the algorithm increase with size of the data, but the rate of growth in the Dijkstra-based algorithm is dramatically faster. The rate of growth in Dijkstra for a network with N nodes and A arcs is known to be O(A log N). In this case, if we define n as the number of grid points on a side of the routing region, then A and N are $O(n^2)$ and the running time of Dijkstra is $O(n^2\log n)$. This grows very fast with n, which means that with the increasing number of nets, the process becomes prohibitively time consuming very quickly. Therefore, it cannot be used as a practical routing tool, although it is a good benchmarking tool.

The poor result of our experiment with Dijkstra's algorithm suggests that we have to look for alternative solutions and that a heuristic approach is needed to intelligently bypass some part of the search space, and search for a solution in a smaller portion of the feasible space where there is more possibility of finding a good solution in a more reasonable amount of time.

In the rest of this document we try to explain the kind of heuristics that we used to solve this specific routing problem.

5. A HEURISTIC ALTERNATIVE TO THE SHORTEST PATH ALGORITHMS

The reason that the conventional shortest path algorithms, like Dijkstra's, take so long to find a solution is that they search for an exact optimal solution (i.e. shortest path) in a huge search space. The size of the assignment and its complexity make an exact solution unreachable – or very time consuming. An alternative philosophy is to find a good-enough solution that can be achieved in a reasonable time.

To overcome the speed deficiency of combinatorial algorithms to solve our assignment, we developed a heuristic alternative. The general idea is to reduce the size of the search space by ignoring those parts of the set that have slim chance of containing a good solution. Through this pruning we could reduce the search time for finding the feasible routes to all the nets. As we will see, the symmetric property of the grid network provides us with a good solution technique to attack the problem.

The main critique against heuristics algorithms is that, regardless of speed, whether they are generally able to solve every routing problem that the combinatorial algorithms can. This is an important concern and we have to be careful about the heuristic approaches that we choose. It is quite a valid point that no heuristic algorithm can be as generalized as combinatorial algorithms. In this thesis our heuristic algorithm, like any other heuristic, tries to take advantage of specific properties of the given data in order to by-pass part of the search space, and speed up the search. Combinatorial algorithms are not allowed to use these specific properties because first, they have to be comprehensive; and second their solution has to be exactly optimal so they cannot prune the part of the search space that might have the exact optimal solution. Therefore, in general, <u>there is a trade-off between generality of an algorithm and its speed</u>. In implementation of a heuristic algorithm, understanding of which specification should be paid attention to and which one should not is very important because there is a potential risk of developing an algorithm that either is not general enough to cover wide range of data sets or is not able to provide a good solution.

To overcome these risks, first we look for a heuristic strategy that is general enough; then we customize it based on specific characteristics of our data.

5.1 Pattern Assignment Method (PAM)

The idea of PAM is simple. On a sequential basis, the algorithm attempts to assign a path to each pair of terminals that is as close to an optimal path as possible. It works on terminal pairs sequentially. It starts with patterns known to be optimal (i.e. paths with least number of bends - the least length and cost) and tries to connect the current terminal pair with that pattern. If that assignment is feasible, i.e. if the path does not conflict with any other already routed paths or any existing block, then the assignment is final and the algorithm picks the next pair. Otherwise the algorithm depreciates the solution to a more costly path - by considering another pattern that is less desired - and checks for feasibility until all of the terminal pairs are examined. It might not be possible that all of the nets are routed. In fact, in the case of BCD it happens that we do not reach 100% routing. If some nets remain un-routed then they should be routed manually or by doing an exhaustive search.

The following three sections describe the algorithm. The first section is dedicated to explaining the steps of PAM in detail. Next, a pseudo code is provided that demonstrates the structure of the algorithm; and finally the third section is an example that shows how PAM works.

5.1.1 The structure of PAM

The following four parts are the building blocks of the PAM structure.

1) Building the routing patterns. Because of the homogeneous structure of the grid matrix the next best choice of each pattern is always known in advance. The least *costly* route that connects a terminal pair is either an 'L' or 'Γ' shape – i.e. two perpendicular connected segments. The next choices are a 'Z' or 'N' shape - i.e. three connected segments. The next is a 'W' shape. Figure 5.1 shows the first seven basic patterns that we use in our algorithm. Notice that in counting the *cost* of a path, not only the number of edges counts but also the number of vias. 'Z' and 'N' paths are different from 'L' and 'Γ' paths only in one extra via, and from 'W' in one less via. Therefore, 'L', 'Γ', 'Z', 'N', and 'W' patterns all have the same *length* but 'W' is more costly than 'Z' (and 'N') more than 'L' (and 'Γ').

The next pattern U has larger *length* than the previous ones. It is because a U pattern goes beyond the bounding box of a terminal pair (all of the previously mentioned patterns fall inside the bounding box of their terminal pair). It is obvious that the deeper the concavity of U is (i.e. the more it goes outside the bounding box) the more costly it becomes. Hence, when the algorithm searches for a feasible U pattern, it starts with the one that its edge is nearest to the bounding box and then looks for deeper U until it finds a fit or reaches a limit. The limit is set inside the algorithm and can be the border of the cell or a different limit. Our experience shows that the best setting is to let the U extend outside the bounding box up to half the distance from the border. There is no precise justification for this setting but the reason that we don't search near the border, i.e. the other half, is that it might block a lot of terminals that are located along the border. For this reason we like to push the routed segments to the inside of the cell as much as possible. Notice here that this is a setting that fits for this specific data and might be different for another, if for example the terminals are mostly distributed inside a cell. This is an example of tuning our algorithm to work at its best performance for a specific data. Also notice that a 'U' pattern can extend in all four directions and '⊃', '⊂', '∪', '∩' all are considered to be a U pattern.

A '_U' (or simply LU) pattern, which is a segment (call it a dash '-') connected to a U, is more costly than U because it has one more bend. In our algorithm the search for this pattern starts with the shortest dash plus the widest U, and continues to the largest dash plus the narrowest U. It should be easy to figure out how the length and cost of more complex patterns – e.g. an 'LU' pattern - compare with the others.

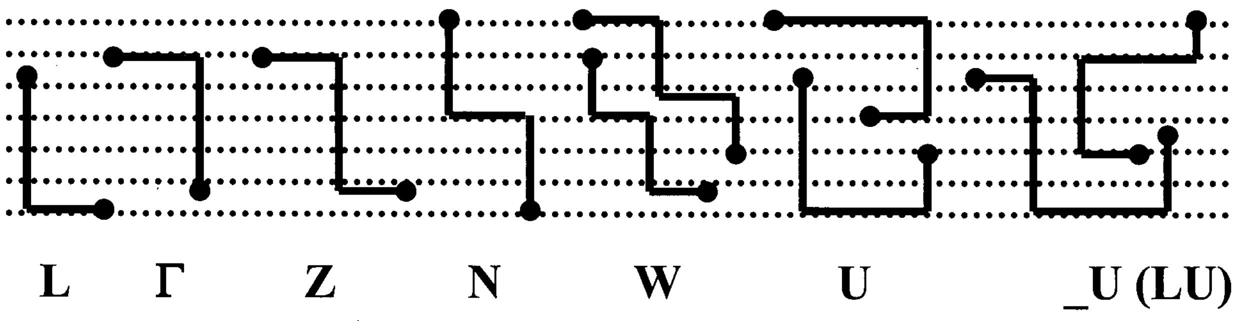


Figure 5.1 - Examples of patterns used in Pattern Assignment Method. For ease of references each one is associated with an alphabet. The number of implemented patterns depends on how complicated the routing patterns intend to be. Moreover, as we will see, more complicated patterns take more time to be feasibility checked. Hence, there is a trade-off between processing time and percentage of successful routing.

2) PAM is a path router (two-terminal net). At the beginning all of the terminal pairs are pushed into a routing list, which is called *priority queue* (next stage explains the priority and ordering technique that is used). If a net has more than two nodes then all of the terminal pair combinations of that net are pushed into the routing queue, i.e. [N x (N- 1) / 2] where N is the number of terminals in that net. Because the algorithm orders the routing pairs based on their minimum length (see part 3) it does not guarantee that all the terminals in one net are routed right after another. When two terminals are routed they are marked as *connected.* When a terminal pair is picked the router checks if at least one of the two terminals is not labeled as *connected.* In case of multi-node each terminal might be already connected to another terminal of that net. If both are labeled then the router skips this pair. This checking avoids creating any cycles. At the end of the routing, if all terminals of a net labeled as *connected* then that net is fully routed.

3) The ordering technique of the terminal pairs. The algorithm orders the pairs, and pushes them into the priority queue, based on their geometrical distances from shortest to longest - measured by ($\Delta x * \Delta y$). The larger ($\Delta x * \Delta y$) is, the more distant the two terminal are from each other. The pairs with shorter distances place on top of the list and are routed first (it was also possible to order them based on their *length*s, i.e. $\Delta x + \Delta y$). The reason behind this type of sorting is that it is less probable that the terminal pairs with shorter distance block the future routings[8]. In practice, the algorithm pushes all the terminal pairs in a list. Then it sorts them by a binary sort algorithm.

Note – One alternative sorting technique is to order the pairs in a reverse order, i.e. decreasing distances. We tested this reverse ordering technique on the BCD data to understand if there is any major difference in percentage of routing. They are almost the same. The reason might be because of the following argument. There is a correlation between the distance of two terminals and their potential route-ability. The further the two nodes are from each other, the more simple

[8] The similar idea was used in the Dijkstra's Algorithm.

patterns exist in their bounding box that might be able to connect them. Figure 5.2 explains this idea. Therefore, if the algorithm starts with nearer terminal pairs then there is more chance of failure of finding a simple routing path between them than if it starts examining more distant ones. So, at the end, the percentage of routing remains more or less the same. More experiments would be necessary for a firm judgment.

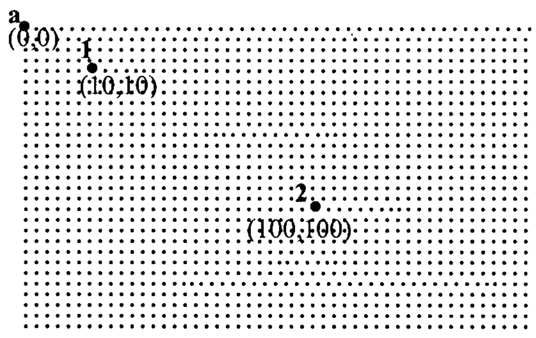


Figure 5.2 - If the terminal pair (**a** and 1) are to be connected then the number of patterns that can be routed between them is less than of terminal pair (**a** and **2**). For example there exists 99 'Z' patterns that potentially can connect **a** and **2**, but there exist only 9 of this type of pattern between **a** and **1**.

Generally, if there are paths that for some electrical reasons - like being electrically hazardous - are considered to be critical, and should be routed as short as possible, then they should be labeled as high priority paths and hence be routed first. This consideration will minimizes the blockages on their way (i.e. the already routed paths) so they most probably achieve the shortest routes. In our cases we did not have such kind of nets, but the program has the capability of considering them.

4) Ordering the patterns and routing each pair. The patterns are chosen sequentially and ordered increasingly based on their *costs*. The ordering is computed this way: for all the patterns with the same length (i.e. 'L' and 'Γ', 'Z' and 'N', 'W') the one with the fewest bends is considered first. After 'W', the 'U' is considered and being tested in all four directions, with the least depth first till the deepest that the algorithm sets. After 'U', the 'LU' is considered with the shorted dash first till the longest dash last. Later, we describe an alternative ordering of the patterns. For each pair of terminals a pattern is chosen and all variations of that pattern are tested until it finds a feasible route between the two – see figure set 5.3. The first successful pattern (meaning that none of the grid points of that path is used by any previously routed tree) is implemented in the layout (we call this technique *First-Feasible-Choice*) and the layout grid matrix is updated. If the routing with that pattern fails then PAM picks the next pattern from the pattern list and tests it the same way until either a feasible path is found or no further pattern is left to be tested. If no success, then the algorithm marks this pair as un-routed (i.e. doesn't mark its terminals as connected) and moves to the next pair until all of terminal pairs are tested.

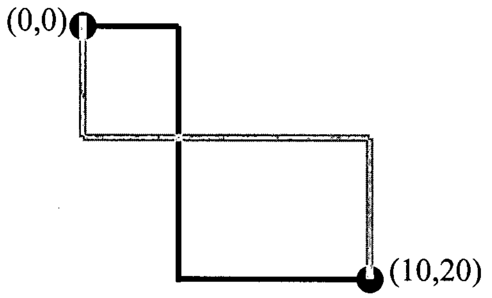


Figure 5.3a - Two alternatives patterns, i.e. 'Z' and 'N', with the same *cost*. There are many variations of the same pattern between the two terminals. Assume that the source terminal is located in (0,0) and the destination terminal is in (10,20). Then in a two-layer

layout, between the two terminals, nine 'Z' paths and nineteen 'N' paths exist – see Figure [b]. The number of variations for other patterns can be computed the same way.

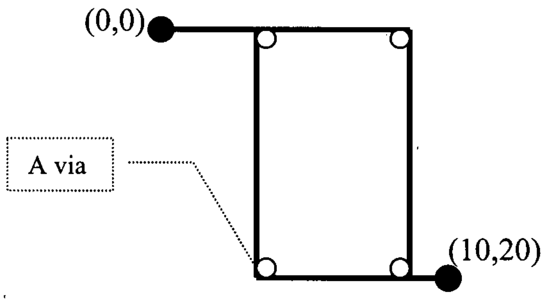
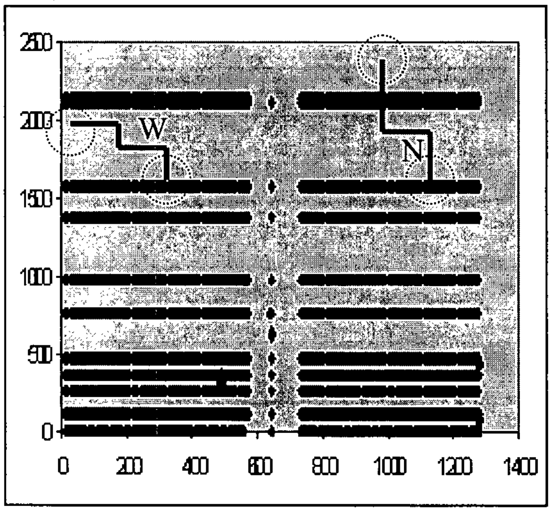
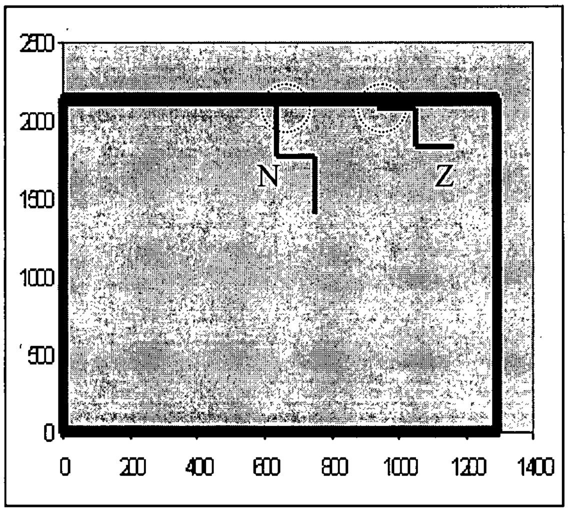


Figure 5.3b – Between the two terminals there are nine potential alternative 'Z' patterns that connect them. The picture shows the two alternative routes. Two horizontal lines connected to one of the vertical lines (by via), constructs a Z route between the two terminals.

Study of patterns and distribution of terminals in the data set shows that, for each pair, some of the patterns have more chance to be successful candidates than the other ones – see Figure 5.4. For example in the BCD data set if one of the terminals is located on the horizontal boundary then a 'Z' pattern has a very slim chance to connect the two terminals. The reason is that in the horizontal boundary the terminal distribution in that data set is horizontally distributed. Therefore, any attempt to fit a pattern with a horizontal upper segment, e.g. 'Z', most probably fails. On the other hand, those terminals that are located on the vertical boundary have less chance to be successfully routed by a 'N' pattern (based on the same argument). Notice that the same argument is valid about the relationship between successful routing patterns and distribution of the source terminals (see Figure 5.4.a). For most terminals a trial of 'Z' patterns fails because the neighbor terminals in the same y coordinate most probably block the path. Considering the two distributions together then one can find out that the best pattern that connects the destination terminals in the horizontal boundary to their counterpart is 'N'; and the best for those that are located in vertical boundary is 'W'. These considerations are what we call *customizing* the heuristics for a special data set. A procedure is implemented in the program that considers the location of the terminal pairs and chooses the patterns based on the positions of each terminal in the layout (the procedure **Continue()** in the program). Notice that the algorithm would still produce the same result if we have not included this optimization. It would try out those patterns and if it fails (that most probably does) then would pick the next pattern. It is only a manual adjustment to the algorithm to optimize its best performance on this data. A different data set might need a different setting to run faster.





(a) Source terminal distribution **(b) Destination terminal distribution**

Figure 5.4 - Figures [a] and [b] show the distribution of BCD data set again. As shown, some patterns have a better chance of routing in certain conditions. The circles indicate the critical areas where the orientation of the pattern should match the distribution of the data in the neighborhood.

5.1.2 <u>The Pseudo code of P.A.M.</u>

The following is the description of the PAM algorithm in pseudo code. A few formats are used for ease of understanding:

A variable in **Bold** is an exact variable that is used in the algorithm. e.g. **node1**

The suffixes (_left, _right, _lower, _upper) to the variables are chosen intuitively. x_left means the x of the left-most node between **node1** and **node2**; node_right means the right-most node; etc.

A variable in *Italic* is a variable that carries the concept but might be implemented differently, e.g. *horizontal_layer*.

For the procedure name, each concatenated word starts with its Capital letter, e.g. BuildPriorityQueue.

++++++++++++++++++++++++++++++++++

BuildPriorityQueue builds a list of all the to_be_routed terminal pairs, and then by using binary sort sorts the list from the least distant to the most.

```
(returns list) procedure BuildPriorityQueue {
    for each net in the cell {
        if net has only two terminals Then { push net into priority_queue }
        else {
            get all combinations of it terminal pairs.
            push each pair as a net into priority_queue
        }
    }
    sort priority_queue in ascending order, by binary sort, based on [$\Delta x * \Delta y$]
    return priority_queue
}
```

BuildPatternQueue builds a priority list of indexes of routing procedures.

```
(returns list) procedure BuildPatternQueue {
  return list { 1.Iroute, 2.Lrouter, 3.Γrouter, 4.Zrouter, 5.Nrouter, 6.Wrouter, 7.Σrouter,
  8.combinations of Urouter, 9.combinations of LUrouter }
```

}

Zrouter demonstrates the algorithm that generates a list of connected segments that together represent a 'Z' path. Note that how 'Z' router uses two simpler 'I' and 'L' router. A similar algorithm finds a 'N' path, which starts with a vertical segment and looks for a connected 'Γ'. 'N' has to search both in layer 1 and 3 for the vertical segment.

```
(returns boolean) procedure Zrouter(node1, node2, return list_of_segments) {
    node_2 = [ node_right ].
    for each node_1 [ from x_left to (x_right-1), y_left, horizontal_layer ] {
        if Iroute(node1, node_1, list_of_segment1) is false then skip this node.
        if Lrouter(node_1, node_2, list_of_segment2) succeeds then {
            list_of_segment = concatenate list_of_segment1, list_of_segment2
            return true
        }
    }
}
```

Similarly **Wrouter** uses one I router and one Z router to find a W. One can use this approach to provide more jagged pattern (e.g. a connected L to N, or a Z to N). In this project, we stopped on W, because practically more jagged patterns did not add any gain to the result.

Building **Urouter** is quite similar to a Z router, except that it looks for an 'I' connected to 'Γ' (Zrouter is 'I' and 'L'). As mentioned before, there are four directions of searching for a U. The following procedure only depicts one direction. Other directions are implemented the same way. The variable **depth** controls the depth of U and is set to half of the distance between the bounding box and the border of the cell.

```
(returns boolean) procedure Urouter(node1, node2, depth, return list_of_segments) {
    node_2 = [ node_right ].
```

```
        for each node_1 [ from x_left to depth, y_left, horizontal_layer ] {
                if Iroute(node1, node_1, list_of_segment1) is false then skip this node.
                if Trouter(node_1, node_2, list_of_segment2) succeeds then {
                        list_of_segment = concatenate list_of_segment1, list_of_segment2
                        return true
                }
        }
}
```

DoesPatternFit simply tests the *pattern* procedure (e.g. Zrouter) on **pair**, and returns true if it succeeds, false otherwise. Before that, it checks for those conditions that there are slim chances of success and returns false without testing.

```
(returns boolean) Procedure DoesPatternFit(pair, Pattern, returns list_of_segments) {
        if first node of pair is on horizontal border of the cell then if the head of pattern
        is horizontal return false.
        if first node of pair is on vertical border of the cell then if the head of pattern is
        vertical return false.
        if second node of pair is on horizontal border of the cell then if the tail of pattern
        is horizontal return false.
        if second node of pair is on vertical border of the cell then if the tail of pattern is
        vertical return false.
        if pattern can generate list_of_segment on pair then return true;
        else return false;
}
```

UpdateLayoutMatrix gets the approved list of segments from DoesPatternFit and updates the cell_grid_matrix.

```
Procedure UpdateLayoutMatrix(list_of_segments,updates cell_grid_matrix) {
        for each segment in list_of_segments {
                set the corresponding grid points to used
        }
}
```

The main routine is simply putting the above procedures in order and loop over the terminal pairs and patterns.

```
// main routine
{
cell_grid_matrix = BuildLayoutGridMatrix(cell, number_of_layers, orientation_list)
pair_periority_queue  = BuildPriorityQueue().
pattern_periority_queue  = BuildPatternQueue().
for each pair in pair_periority_queue {
        if both terminals of pair are labeled as connected then skip this pair
        for each pattern in pattern_queue {
                if  DoesPatternFit(pair, Pattern, list_of_segments) succeeds Then {
                        UpdateLayoutMatrix(cell_grid_matrix, list_of_segments).
                        Attach the list_of_segments to pair
                        label the terminals as connected.
                        skip the inner loop.
                }
        }
}
// end.
```

}

++

5.1.3 A pictorial example of P.A.M.

Figure set 5.5 explains the idea of PAM algorithm for routing of two nets in a one layer layout by an example.

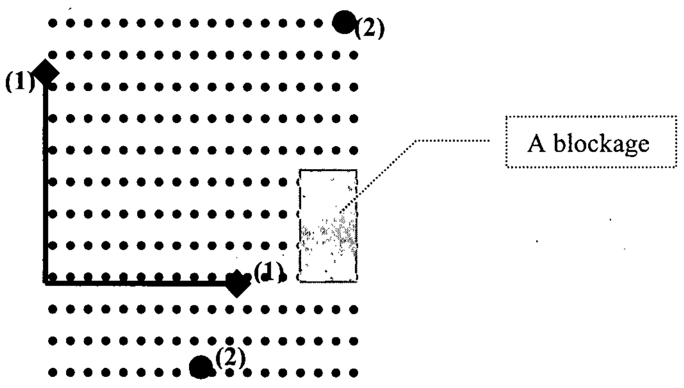


Figure 5.5.1 - There are two sets of terminals to be routed. Assume that the algorithm has already made a successful 'L' pattern route on pair (1), and now is considering pair (2).

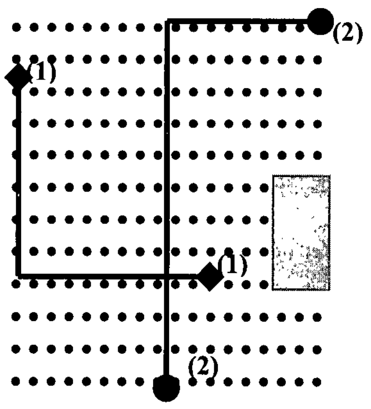


Figure – 5.5.2 The algorithm tests a 'Γ' pattern on pair (2). The solution is infeasible because the paths cross each other (in one layer).

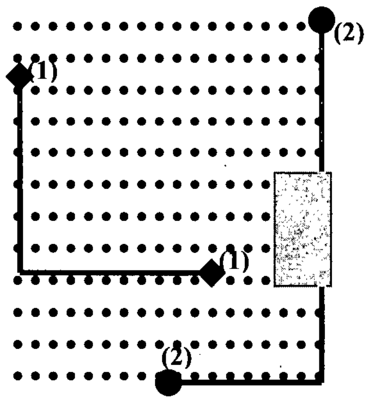


Figure – 5.5.3 Next, the algorithm tests a 'L' pattern. The solution is infeasible because a blockage hinders the path.

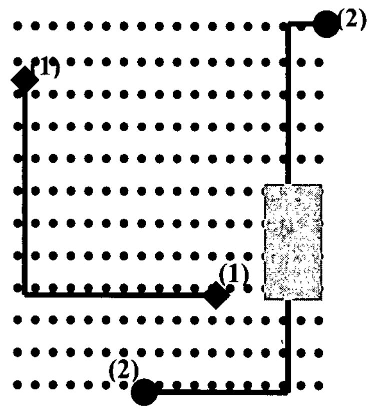


Figure – 5.5.4 The path search of the terminal set (2) has been degraded to a sub-optimal option (a 'Z' pattern). The new routing is still infeasible, because the blockage hinders the path.

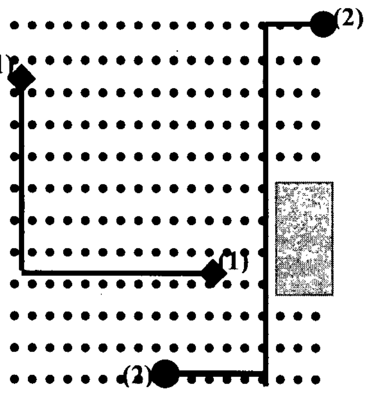


Figure – 5.5.5 The next 'Z' pattern is feasible, hence considered as a solution to the pair (2). The algorithm implements it in the layout and stops, as all the pairs are routed now.

Table-A shows the result of running PAM algorithm on the BCD data set. Comparison between the results of Dijkstra-based algorithm and PAM in Table A shows the advantage of employing PAM over the Dijkstra-based algorithm. As seen, the scale of run time has been reduced from days to hours. This is a direct consequence of reducing the search space.

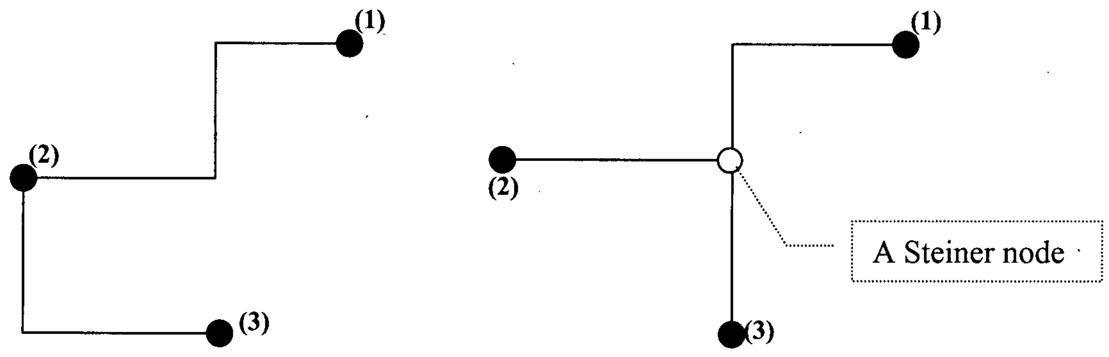
5.2 Deficiencies of PAM and avenues for improvement

The PAM algorithm represents a good improvement compare to the Dijkstra's Algorithm. However the outcome of the algorithm shows that there are weaknesses in our algorithm that need improvement. These weaknesses primarily fall into two groups.

1) *First-Feasible-First_Choice* is a weak strategy. Because the selection of the paths is sequential, there is a risk that a chosen path blocks one or more feasible paths of the other still un-routed pairs. It is worthwhile if we improve the PAM algorithm so that the best path is chosen more intelligently from a pool of choices. The "best path" means selecting a path so that it minimizes the chance of future obstruction; so in general it increases the percentage of successful routing.

2) The second problem with PAM algorithm is that when it routes a pair, during the routing, it has no knowledge that it might be a portion of a multi-node net. For such nets, the solution of the algorithm might be far from a near-optimal solution. Figure set 5.6 demonstrates a possible routing result of PAM and its alternative ideal solution, i.e. Minimal Rectilinear Steiner Tree (MRST).

The result of our effort to address these weaknesses is a new algorithm, called Segment Utilization Method, that we explain it in the next section.



(a) (b)

Figure 5.6 - In Figure [a] suppose that (1) → (2) is routed first and then (2) → (3). It is quite possible that the result becomes a tree as shown in [a] while the optimal connecting tree is in fact Figure [b], i.e. MRST.

6. Segment Utilization Method (SUM)

In previous section we explained that although PAM is a good improvement over Dijkstra's Algorithm, it still suffer from a poor strategy for choosing a good route to route a net. In fact, PAM is a lazy strategy and sticks to the first solution that it finds. The optimization is assumed to be inherent in the concept of PAM and it does not look for any dynamic optimization when searches for a solution.

The Segment Utilization Method (SUM) algorithm is implemented to overcome the deficiencies of the *First-Feasible-First-Choice* strategy in PAM by a more careful choice of feasible paths. There are a few similarity and some differences between the two algorithms. SUM is built on the same concept of PAM, i.e. it still looks for simple patterns in sequential order to find the best fit. So - like PAM - SUM is also a "pattern assignment method". The difference – or improvement - is that when SUM chooses a pattern to search for, it also takes into account an optimization objective by looking among the all alternatives of that specific pattern and choosing the one with the best *fit*. The "*fit*" is a variable goal and depends on if SUM works on a pair or on a multi-node net. On the multi-node side, with a small change in the goal function of the algorithm, i.e. *fit*, SUM is able to route a multi-node net more elegantly than PAM.

SUM, basically, starts from the same logic that PAM is built on. However, most of the improvement in processing speed of SUM is based on the idea of search in the segment pools instead of node by node checking in PAM. For checking the feasibility of a path in PAM we have to check all the grid points located on it for availability. In SUM, the feasibility test is very much faster, because the algorithm only checks the availability of few segments (e.g. three segments in a 'Z' pattern, etc.) in an indexed fast-access data structure, i.e. the *pool*, instead of individual node-by-node checking.

The following two sections describe the SUM algorithm. The first part of section one (6.1.a) describes the algorithm in detail. The second part of section one (6.1.b) demonstrates the changes in the goal function that enable SUM to route the 3+ node nets. Then in subsection two, a pseudo code demonstrates the structure of the algorithm and then the data structure of SUM is described in detail.

6.1 The algorithm

The four stages of PAM, which are explained in 5.1.1, are also valid in SUM with some modifications. We revisit those steps and explain the modifications.

1. *Constructing and ordering the routing patterns* remain the same in SUM. It originates from our arguments about optimality of simple patterns and that the less bends exist in a path the better that route is. A different argument suggests that by a different order of patterns a better completion rate might be possible in the cost of sacrificing more routing time. This argument is addressed in section 7.3 when revised SUM is explained.

2. *Execution on terminal pairs only.* Like PAM, SUM also works on one terminal pair at a time. Both algorithms split a multi-node net into all combinations of its terminal pairs and treat each

pair individually. The difference is that PAM treats all of the pairs of a net as if they are independent pairs. The result, as shown in section 5.2, can be poor. In SUM, if the first pair of terminals in a net is already routed successfully, then the next pair will be treated differently. Section 6.1.b explains multi-node routing in more detail.

3. *The ordering of terminal pairs.* The ordering technique of the terminal pairs for both algorithms is the same and explained in section 6.1.3.

4. *Routing a terminal pair.* SUM picks each terminal pair sequentially and tries to fit the least cost patterns that fit that pair – like PAM. However, the way that SUM does this search is different than PAM as it uses a different data structure and different search technique to route a pair. The following section explains the details of this stage of SUM.

The steps of SUM for 2-node path routing

The basic idea of SUM is implementation of the following four steps:

I. Build up a pool of available segments. In the early stage a data base is constructed – called *pool.* The function of *pool* is to carry all the available segments for routing in all the layers at all times. After each routing it will be immediately updated (step III), so it is always valid. For our data that has three layers *pool* will be a vector of three lists - one for each layer – and each list carries the available segments in that specific layer (list 1 and 3 contain vertical segments of layer 1 and 3; and list 2 carries horizontal segments of layer 2). We call each element of a *pool* as a *pool_segment.* Figure (6.1) explains an example of how *pool* is built.

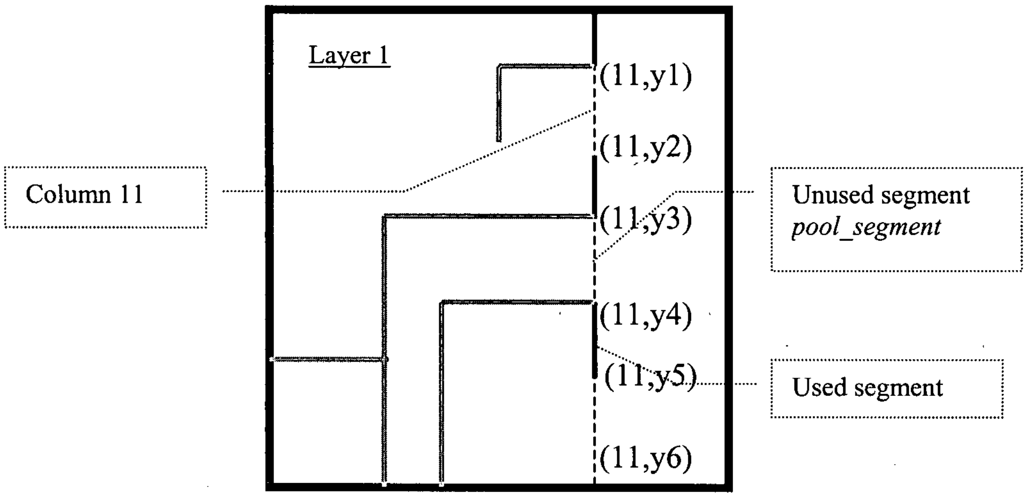


Figure 6.1 - this is an example that explains how the *pool* of available segments is built up. Assume there are three layers and each layer is dedicated to one direction (layer one and three to the vertical direction – move in direction of y axis; and layer two to the horizontal direction – move in direction of x axis). Three arrays are defined and each one is associated with one layer: C1[# of Columns] – stands for Columns in layer #1, R2[# of Rows] - R2 stands for Row in layer #2, and C3[# of Columns] - C3 stands for Column in layer #3.

C1 represents the data structure of the available segments in layer-1, R2 for layer-2 and C3 for layer-3. The three data structures together are called the *pool*. The purpose of each data structure is to hold the information of available segments in that layer. Each element of the arrays keeps two kinds of information: (i) the number of available segments (the segments that are not already occupied by routed paths) in that row or column, and (ii) the begin-end addresses of those segments. Notice that the index of the array shows the x or y coordinates of the segments depending on whether they belong to layer two or to layer one and three. For example, in the above picture column 11 in layer one is involved with three already routed trees (those segments of the trees that are not in column 11 shown as gray to be distinguished from the other in-column segments). Then, based on the figure the 11th element of C1 is filled as follows

C1[11].n = 3

C1[11].segment[0] = (y1):(y2) meaning: segment[*0*] = (x=*11*,y=*y1*,z=*0*) to (x=*11*,y=*y2*,z=*0*)

C1[11].segment[1] = (y3):(y4)

C1[11].segment[2] = (y5):(y6)

II) Using *pool* for routing. The *pool* structure is used to find all the feasible variations of a certain pattern between two terminals (refer to stage 4 of PAM algorithm for an example on variations of a 'Z' pattern). The algorithm search through the pool data base and looks for those sets of crossing *pool_segments* that together represent a pattern and connects a source node to its destination. Remember that as *pool* will always be updated to hold available segments, a found path is guaranteed to be a feasible one. If the search succeeds to find at least one feasible solution then it goes to the nest step (step 3, below) and finds the best solution among the found solutions. Otherwise it picks another pattern and reinitiates the step 2, until either finds a solution or no more pattern is left to examine.

Figure 6.2.1 shows an example of how *pool* is used to build a feasible pattern.

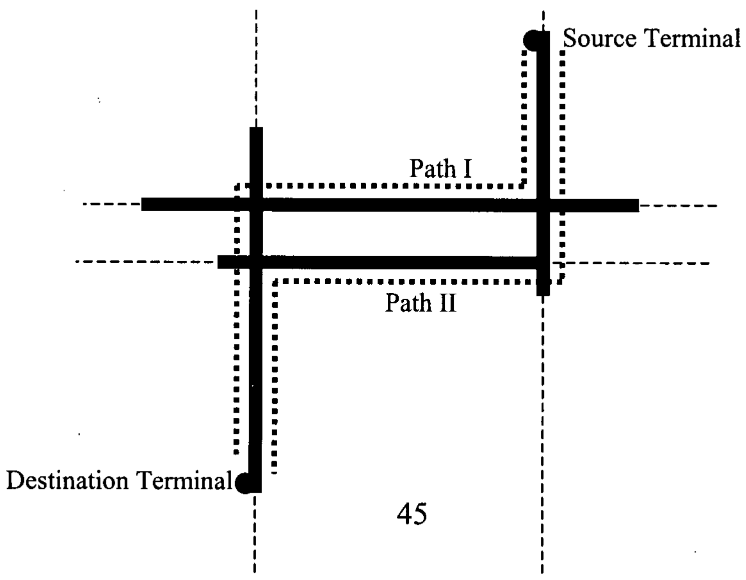
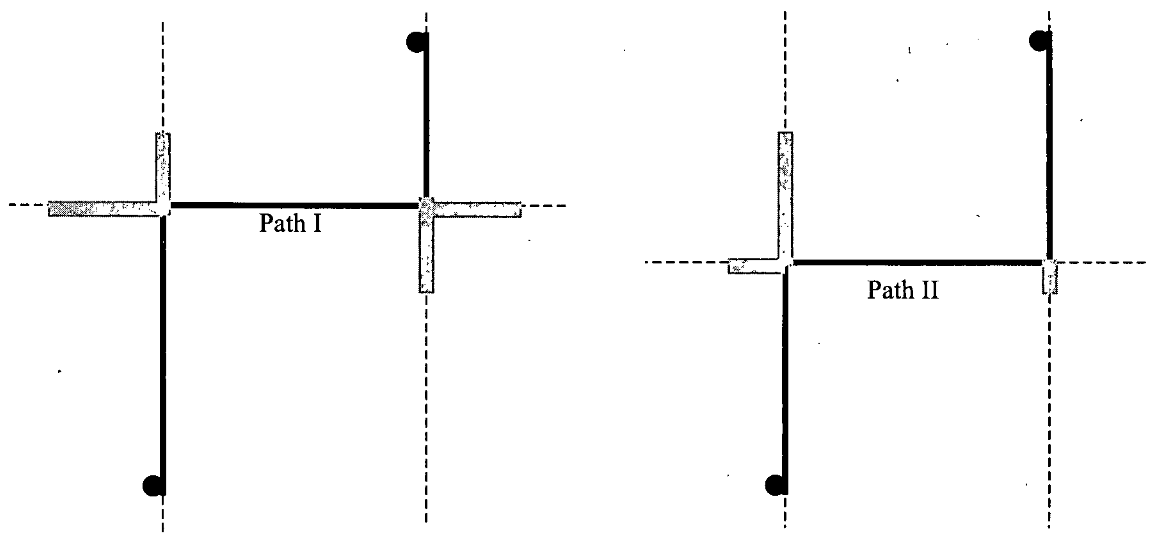


Figure 6.2.1 - assume that the thick lines are the only relevant *pool_segments* that allow 'N' pattern routes between the source and destination terminals. There are four *pool_segments* that together can provide two 'N' pattern options – path (I) and path (II).

III) Choosing among the options. Among the selected paths from section 2, the one with the least amount of *leftover* segments is chosen as the best *fit*. In other word, *leftovers* are our *fit* objective. '*Leftover*'s are those parts of the *pool_segments* that are left out after using them in routing of a path in the layout – see Figure 6.2.2. *Leftovers* are the new available segments, i.e. new *pool_segment*, after *pool* is updated by the new route. Notice that *pool_segments* are our resources for routing a net and the more *pool_segments* with bigger size we have in the *pool* the more chances are there to find simpler pattern to route a net. Examining the leftovers to decide if a routing is good, is a good instrument for routing optimization; because it is a measurement of *waste*. It is an indicator that tells the router how much a path will *waste* the resources if it is exercised in the layout. In our implementation we use one simple way of considering *leftovers* to define the *waste*. We define the *waste* to be the accumulated length of total *leftovers* in each path, and the router tries to minimize it. The justification behind this definition is that among all of the feasible paths the ideal choice is the one that produces no *leftover*. If not possible, at least it uses those *pool_segments* that "almost" fit the required segments for a route – with the least remained leftover segments. There are also other ways of defining *waste* based on *leftovers*. One way is to look at the number of created *leftover* segments instead of the length of them, so the *waste* is the number of created *leftovers* and not the size of them. Justification behind this observation is that the less number of leftover is created the less *pool_segments* will be fragmented. The less fragmentation we have the more simple patterns are possible. Hence, *waste* can be defined based on *leftovers* as either the best fit or the least fragmented. A weighted average of both can be another consideration.

In SUM, when the total length of *leftovers* between two options are the same then it looks at the number of the *leftovers* and picks the one with least number of created *leftovers.* The project is open for further research on which objective function best utilizes the usage of *leftovers* in computation of *waste* and optimal routes.

Figure 6.2.2 explains how *leftovers* are created for any of N patterns from figure 6.2.1.



(a) Path-I with its leftover segments **(b) Path-II with its leftover segments**

Figure 6.2.2 - The algorithm argues that option (b) is a better choice than (a) because as the two figures show the leftover segments (the thick gray segments) of path (II) is less than of path (I). The objective behind the argument is to minimize the amount of fragmentation in the free segments when doing a routing. The less fragmentation means the more availability of large segments and consequently the more probability of routing of the next trees with simpler patterns. For example, if path (II) would not produce any leftover segment - i.e. all the three segments were exactly fitted in the required sizes then path (II) would be the ideal choice because it would produce no leftover segments, so no fragmentation.

IV) Updating the pool and the layout data base. After the "best" path is chosen it is implemented in the layout and the *pool* data structure is updated accordingly. Unlike PAM, SUM does not need a grid-matrix to hold the track of routed paths. This is another advantage of SUM over both PAM and Dijkstra's because when the layout is big the memory usage for holding the grid matrix can be significant compare to the size of the program. As a side effect, it might slow down the algorithm because if the size of grid matrix does not fit in the memory then the operating system has to use the hard disk as a virtual memory; and this significantly slows down any program. SUM uses two – comparatively - small data structure that keeps the track of available segments (*pool*) and another similar data base – call it **layout** data base - that keeps the track of used segments; and represents the layout of the cell. When the program runs, when a net is routed, both *pool* and *layout* are updated. At the end, the *layout* data base represents the outcome of the algorithm that was run on the data. In section 7, we explain that with a minor change in the structure of *pool*, it can represent both the available segments and the used segments; hence the layout data base is no longer needed – that is even less usage of run time memory. This optimization pays back when the program runs on huge data.

Figure 6.2.3 explains how *pool* is updated to carry the new *pool_segment*s in our example after routing of that pair.

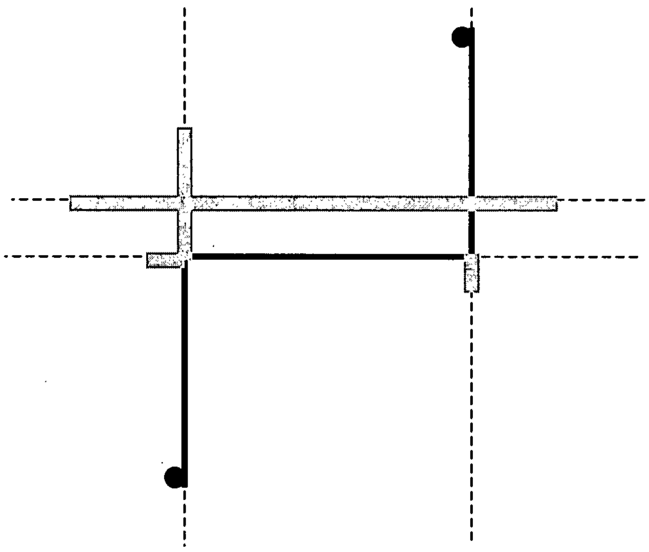


Figure 6.2.3 - Path (II) is implemented in the layout and the *pool* data base is updated based on the new available segments.

6.2. SUM Extension for routing of multi-node nets

The discussed algorithm in the previous section explained the routing of a path. If a net has more than two nodes then the algorithm behaves slightly differently.

When a pair is picked up from priority queue, SUM checks if this pair belongs to a multi-node net. If yes, then it checks if its terminals are already connected (if they are labeled as *connected*). If none of them are connected then SUM checks if any other pairs in the net are already routed. If so, then it pushes this pair back to the bottom of the priority queue. The reason is that the algorithm does not like to end up with the problem of routing between two previously routed paths; it prefers to route one pair and then sequentially add the next terminal to a connected tree. If one terminal of the picked pair is routed and one not then it tries to route it the same way as the other pairs except that when looking for the optimal path it has a different goal in mind other than minimizing the leftover. Figure set 6.3 explains the algorithm for a 3-node net. For higher number of nodes the algorithm works similarly.

We assume that in the net the first two terminals are already connected (figure 6.3.1)

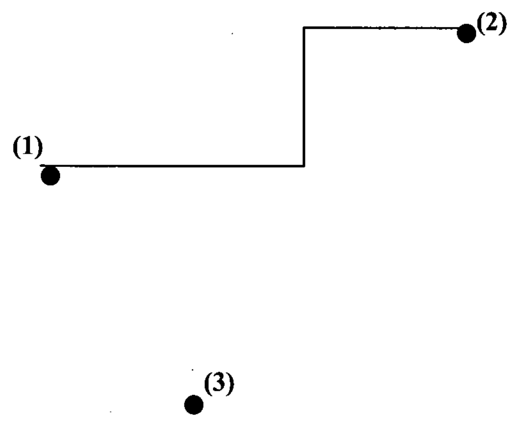
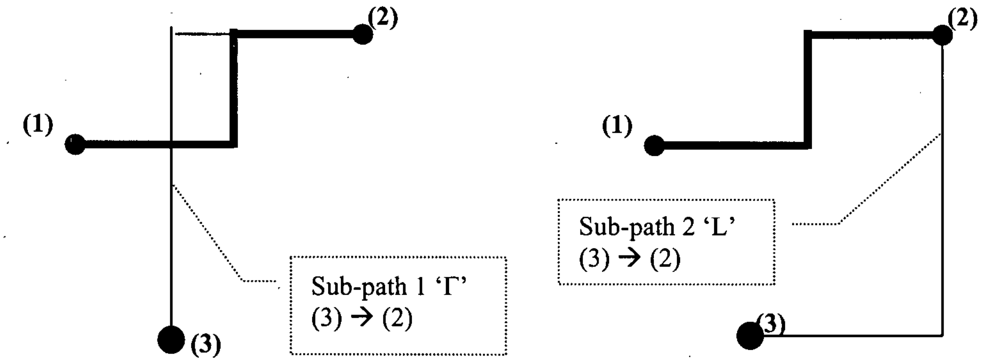
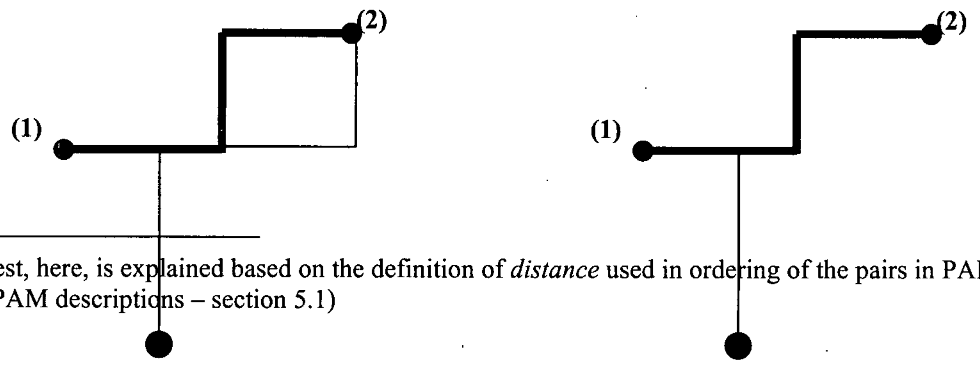


Figure 6.3.1 - An example of 3-node tree routing by the SUM algorithm. The first step is connecting the first two nodes. A pair of nodes of the tree is chosen and the 2-node SUM algorithm is applied to them (e.g. it connects node 1 and 2).

Now the second pair is picked up, for example (#2,#3), and the algorithm tries to connect the third node, i.e. #3, to the already routed path, i.e. (#1,#2). Notice that the to-be-routed pair

(#2,#3) are coming from the priority queue and because the priority queue is already sorted then terminal #2 should be the *near*est[9] to #3. Then SUM tries to route them the same way as before but only with a different objective goal. The first part of the procedure is the same as 2-node SUM algorithm. Different paths, which connect node #3 to the other node, are tested for feasibility and a list of feasible paths is built (Figure 6.3.2). The second part of the procedure, which chooses the best path, is slightly different from the previous one. In SUM, for a pair, the optimal path is chosen based on the least accumulated leftover segments. Here, the path is chosen based on the least amount of un-allocated grid points that each path uses to connect the new terminal #3 to the previously routed path (#1,#2). Figure set 6.3.2 demonstrates the example.





[9] The *Near*est, here, is explained based on the definition of *distance* used in ordering of the pairs in PAM (refer to phase-I of PAM descriptions – section 5.1)

Sub-path 3 'N'
(3) → (2)
(3)
Sub-path 4 'W'
(3) → (2)
(3)

Figure 6.3.2 - Show a few feasible paths that connect node (3) to (2). It is easy to judge from the pictures that path (4) uses the least amount of unallocated grid points to connect node (3) to the previously routed path, so is the best choice.

Notice that the algorithm does not necessarily provide an optimal tree. The example in Figure 6.3.3 shows the difference. But it also shows that the difference might not be significant.

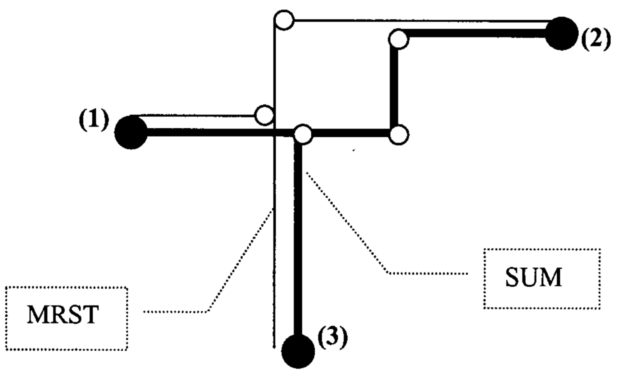


Figure 6.3.3 - A possible outcome of the algorithm (thick line) and a MRST (thin line). Here there is a small difference between the *costs* of the two routes. The MRST uses one less via (the hollow circles) than SUM.

Table-A shows the result of executing SUM algorithm on the BCD data set. As shown, it has an essential improvement over the PAM algorithm - both in terms of speed and completion rate.

The outcome of SUM also supports the claim that - on the average – it produces routes that are close to optimal. Table-6.1 shows the total number of used edges by the two versions of SUM on BCD data (r-SUM is an extension of SUM and will be explained in section 7.3) and compares them with the lower bound from Table-B (appendix B).

Table-6.1 reveals some interesting facts about BCD data and the outcome of SUM. First, the comparisons of rows #1.1 with #1.2, and #2.1 with #2.2 show that the lengths of routed paths of both SUM and revised SUM, on the average, are not very different from ideal path (Because there are very few nets with more than 3 nodes, computation of actual optimal route was not difficult – see appendix B for more explanation and also underline of table 6.1 for numbers). On the horizontal layer SUM uses 10% more edges than minimum (revised SUM uses 8% more), and in the vertical layers SUM uses 11.6% more edges than minimum (revised SUM uses 8.3% more). Second, comparison of row 2.1 with 3.1 shows that revised-SUM does a good job in utilization of the space. It used 93% of the available horizontal edges (the middle layer) meaning that it does not waste the provided space very much (the ideal usage obviously is 100%). Third, comparing row 2.2 with 3.2 shows that the un-routed paths are those ones with the highest *lengths* (32% of the needed horizontal edges and 21% of the needed vertical edges are for 20% un-routed trees – 20% of the nets need more than 20% of the resources so on the average they have to be longer). One, however, should not be surprised as the algorithm orders the trees based on increasing length, so the shorter nets are routed first.

1.1	The number of vertical edges used in SUM The number of horizontal edges used in SUM	3,695,168 2,429,412
1.2	The minimum # of vertical edges required for the same routed nets The minimum # of horizontal edges required for the same routed nets	3,311,237 2,208,552
2.1	The number of vertical edges used in revised-SUM[10] (r-SUM) The number of horizontal edges used in revised-SUM (r-SUM)	3,799,008 2,569,265
2.2	The minimum # of vertical edges required for the same routed nets The minimum # of horizontal edges required for the same routed nets	3,509,101 2,379,213
3.1	Total # of available vertical edges in layer 1 and 3 (from Table-B) Total # available horizontal edges in layer 2 (from Table-B)	5,511,138 2,759,749
3.2	Minimum # of needed vertical edges (from Table-B)	4,464,637

[10] Revised-SUM is an improved version of SUM that will be discussed later in section 7.3.

	Minimum # of needed horizontal edges (from Table-B)	3,514,991

Table 6.1 - The numbers show the vertical and horizontal edges used by SUM (and revised SUM), and their minimum equivalent requirements. Considering the fact that most of the nets in the BCD data are either 2-node of 3-node, the manual computation of close to optimal paths is quite feasible. There are only 47 four-node nets and one 5-node net. The numbers in row 1.2 and 2.2 represents the lower bounds of required edges for routing the same nets of their previous rows.

The last point to mention again is that, testing an algorithm only on two data sets does not prove the power of that algorithm. More testing and evidences are needed to confirm the abilities of SUM. But considering the fact that the BCD data has been a very challenging data set, so that Intel has offered it as a benchmark, we may claim that SUM (and its its Revised version, described next) succeeds in being a good algorithm for this sort of routing problem.

6.3 A detailed description of the SUM algorithm

Because SUM is the algorithm that our project's final report is based on, we explains the body of our algorithm in more detail, with explanation of all the data structure and functions that are used in our C++ codes. Thus this section is effectively the documentation of the code. The code, itself, is also full of comments; and meaningful names are chosen for variables and functions.

6.3.a. The pseudo code

The following pseudo code explains the structure of SUM, and then a detailed description of functions and variables explains how the pseudo code matches the C++ code.

In the following pseudo code, the lines preceded by "//" are comments, they are comments on the next line(s) codes.

```
// main routine
{
// The equivalent code in C++ is the function initialization. The function is described later
// in this section and explains in details how pool and layout are built up.
(pool, layout) = BuildPoolAndLayout(cell, number_of_layers, orientation_list)
// The functions with PAM prefix means that they are borrowed directly from PAM pseudo code.
// The following two functions tell us that SUM uses the same algorithm to build the priority queue
pair_periority_queue  = PAM_BuildPairPriorityQueue().
pattern_periority_queue  = PAM_BuildPatternQueue().
for each pair in pair_periority_queue {
        if both terminals of pair are labeled as connected then skip this pair
        if none of terminals are connected but there is a connected terminal in net
        then push back this pair to the bottom of pair_priority_queue and skip
        // NodeToNodeSUM calls PAM_DoesPatternFit to find all feasible solutions.
        // Then it returns the best fit path (i.e. the least leftover)
        if it is a terminal to terminal connection
        then call NodeToNodeSUM(list_of_segments, list_of_pool_segments)
        // if it is a 3+ node net routing then it calls a different routing. In C++ code it calls
        // Do3NodeRouting. The name is misleading. It is responsible for connecting a terminal to
        // the rest of already routed tree.
        if it is a terminal to a tree connection then call NodeToTreeSUM
        if routing succeeds then
        {
                // In C++ the function Connect() is responsible for updating pool and layout
                UpdatePoolDataBase( use list_of_pool_segments)
                UpdateLayoutDataBase( use list_of_segments)
                label both terminals as connected
                // this means that net has at least one terminal that labeled as connected
                label net as has_routing
            }
        }
    }
    // end.
    }
```

The procedure NodeToNodeSUM is the pair routing SUM algorithm. In C++ it is the **workOn()** function. The name is chosen because this is the main case in SUM so it works on it!

```
(returns boolean) procedure NodeToNodeSUM(returns list_of_segments,
                                          returns list_of_pool_segments)
{
    // note that pattern is an index to a function, e.g. Zrouter. The Zrouter is the same
    // algorithm described in PAM but it uses pool to extract a feasible path.
    for each pattern in pattern_queue {
        // notice that DoesPatternFit is the same function in PAM except that when it
        // calls pattern the SUM version of pattern is called; so it needs pool data base
        if PAM_DoesPatternFit(pair, Pattern, pool, list_of_segments) succeeds
        Then Push list_of_segments to the feasible_list
    }
    if feasible_list is empty then return false.
    for each item in feasible_list compute its leftovers from pool
    find the items in feasible_list with the least leftover.
    if items>1 then find the one [i] with the least number of leftover segments
    list_of_segments = feasible_item[i]
    find those pool_segments from pool that cover the segments in list_of_segments
    and put them in list_of_pool_segments
    return true.
}
```

The procedure NodeToTreeSUM is part of the multi node routing, when SUM tries to connect an un-routed terminal to the rest. As one may notice the only difference between this function and NodeToNodeSUM is the way that the best path is chosen.

```
(returns boolean) procedure NodeToTreeSUM(returns list_of_segments,
                                          returns list_of_pool_segments)
{
    // note that pattern is an index to a function, e.g. Zrouter. The Zrouter is the same
    // algorithm described in PAM but it uses pool to extract a feasible path.
    // The other difference is that if a segment is not available in pool the Xrouter looks in
    // layout to see if it is taken by the same net. If so, it is still a valid path.
    for each pattern in pattern_queue {
        // notice that DoesPatternFit is the same function in PAM except that when it
        // calls pattern the SUM version of pattern is called. It also needs layout to find
        // the overlap segments
        if PAM_DoesPatternFit(pair, Pattern, pool, layout, list_of_segments) succeeds
        Then Push list_of_segments to the feasible_list
    }
    if feasible_list is empty then return false.
    // in C++ code here is where Do3NodeRouting is being called. It computes the amount of
    // grid points that is used in each feasible_list item. It uses both pool and layout data base
    // to compute the portion of segments that are already
    for each item in feasible_list compute its grid_usage from (pool, layout)
    find the item [i] in feasible_list with the least grid_usage.
    list_of_segments = feasible_item[i]
    find those pool_segments from pool that cover the segments in list_of_segments
    and put them in list_of_pool_segments
    return true.
}
```

6.3.b The data structure of SUM

In this section the data structure of SUM is explained in detail, the way that it is coded. The names in bold are the exact variable or function in the software code.

Initialization() initiates the variables and sets up the data structures needed for execution of the program. There are four data structures that mainly represent the needed data for execution of the program.

I) Layout represents the layout grid network. It is a 3-dimensional array of integers such that its each element represents a grid. The value of **Layout**[*x,y,z*] indicates if the grid in layer *z* at position (*x,y*) is taken. It is zero if it is untaken. Otherwise it contains the index of the tree that it belongs to.

NodeType and **NetType** are the two key data structures, which hold the information of the terminals and the trees in the program respectively. They play key roles in speed and compactness of the code.

II) NodeType holds the information of the terminals in a useful way for the purpose of interaction with the rest of the program. It consists of five elements:

1) **g** is an array of **GRID** that represents all the terminals in the tree that node belongs to. **GRID** is a data structure that an instance of it represents a grid in the layout. It consists of three integer variables **x**, **y**, **v**. **(x,y)** indicate the two dimensional position of the grid, and **v** indicates which tree it belongs to, if it does. It does not need to hold the index of layers because the program assumes that all of the terminals are located in the lower layer (layer number three). **g[0]** is the address of the indexed node. The subsequent indexes (**g[1]**, **g[2]**, ...) are the addresses of the other terminals in the same tree.

2) **net** holds the index of the tree which the terminal belongs to.

3) **i** holds the position of the node in the tree (starting from 0).

4) **n** holds the number of nodes in the tree (indexed by **net**).

5) **done** is a logical variable, **TRUE** when the node is successfully connected to the other side and **FALSE** otherwise.

For example, assume that node number 45, located at (x,y) = (1,1247), belongs to tree number 12 as its second node, which has three terminals (450,112), (1,1247), (1233,976); and the tree is not routed yet. Then **Node[45]** is encapsulated as follow

Node[45] = { **g[0]**=(**v**=12,**x**=1,**y**=1247), **g[1]**=(12,450,112), **g[2]**=(12,1233,976), **net**=12, **i**=1, **n**=3, **done**=**FALSE** }

III) NetType represents the data structure of the trees in the program. It consists of three elements:

1) **g** is an array of **GRID** that represents all the terminals in the tree.

2) **n** is number of nodes in the tree.

3) **done** is a logical variable, **TRUE** when the tree is successfully routed, **FALSE** otherwise.

For example, consider tree number 12 in the previous example. Then **Net[12]** is encapsulated as follow

Net[12] = { **g**[0]=(v=12,x=1,y=1247), **g**[1]=(12,450,112), **g**[2]=(12,1233,976), **n**=3, **done=FALSE** }.

IV) C1, **R2**, **C3** together represents the free segment pool that Segment Utilization Method idea is built upon. Each one is an array of **FreeSegmentType** that holds the following two elements[11]:

1) **n** is the number of free segments in the indexed row or column.

2) **segment** is an array of **AB** structure that holds the geometrical information of the free segments in that row or column. **AB** is a data structure consists of two elements, **mv** and **Mv** that stand for *minimum value*, and *maximum value* respectively. **mv** and **Mv** represent the *x* or *y* coordinate of the beginning and end of a segment. In **C1** and **C3** the index of the arrays represent the *x* coordinate of the segments, so **mv** and **Mv** represent the *y* of the beginning and end of the segments. The same way in **R2** the index of the array represents the *y* coordinate, so **mv** and **Mv** represent the *x*.

The body of the program loops on the number of nodes (i.e. terminals). In each loop, if the indexed node is not routed then the routing algorithm initiates the SUM algorithm to connect that terminal to its destination counterpart (see section 7.1.a). If it is part of a *3+ node* tree, then the algorithm connects it to the rest of the tree (see section 7.1.b). Inside the main loop the function **WorkOn(**node number**)** executes SUM on the current node. **WorkOn()** sequentially tests the patterns to see if they connect the node to its destination counterpart. For example it starts with

[11] In section 8.2 we demonstrate a different data structure of the pool that remove the necessity of holding **Layout** in memory.

execution of **L_Routing**(node number), then executes **Z_Routing**(node number), etc.. The order that the algorithm chooses the patterns is the same as in PAM. A **Continue()** procedure determines if any of x_**Routing()** functions are useful to exercise (refer to section 5). If not then the algorithm ignores that function and moves to the next x_**Routing()** function.

In each sub-routine whenever it finds a successful routing it implements the chosen route to the **Layout**. Then it updates **C1, R2, C3** based on the new remaining free segments, labels the node as done, labels the associated net as done if every node in that net is connected, and restarts the main loop with the next node.

The structures of pattern routing functions are very similar. For example **Z_Routing(**node number**)** searches for three connectable segments in a 'Z' format in the pool set (two horizontal segments in R2 connected by one vertical segments in either C1 or C3, or two vertical segments in either C1 or C3 connected by one horizontal segment in R2). The function **DoSegmentFit(…)** does the inspection job. If there is an option then among the found paths it picks up the one with the smallest number of computed leftover segments. Once that it decided which path to choose it calls the function **Connect(…)** that implements the path in the **Layout** and updates the pool set - i.e. **C1, R2, C3**. If no path of type 'Z' is found it ends the **Z_Routing()** function and the program runs the next pattern routing function.

In the main loop when a pair is picked up from the priority queue, the algorithm determines if routing of this pair is a 2-node routing or 3+ node routing (i.e. a node connects to already routed terminals). If it is a 3+ node routing then it sets a flag – i.e. **extendedSUM** – so the routing sub-routines (**L_Routing()**, **Z_Routing()**, etc.) won't implement the routing. They only find the feasible paths and pass it to the outside. The subroutine **Do3NodeRouting(feasible,n)** is responsible for choosing from the list of feasible paths. Integer **n** holds the number of paths to be selected from. The **feasible** data structure is an array of **PathStructureType** that holds the information of each path. **PathStructureType** is a data structure that holds the addresses of all the segments that together build that path in the pool set. **Do3NodeRouting(...)** decides which path uses the minimum number of grids (i.e. the idea of extended SUM) and implements it in the **Layout**. In each path, it simply sums up the lengths of those segments, which are not part of already routed net, and then chooses the minimum.

Note – In practice the program does not need to keep the information of all the paths and choose the minimum one at the end. It dynamically selects the minimal path by holding the information of the up-to-now best feasible path in an extra variable (i.e. **bestFeasible**). The future tested paths are only compared with this variable. Each time that a shorter path is found it will be replaced with the current one. The final path held in **bestFeasible** is the minimal path.

When all the nodes are considered once the algorithm finishes. The function **TheEndOfStory()** is responsible for saving the processed layout and releasing the memory.

The above description explains the key functions and the structure of the SUM algorithm in the way that had been actually developed in the program. The code uses meaningful variable and function names, appropriate indentation, and useful comments. Therefore the program code is self-documented for those who want further details.

7. IMPROVMENTS TO THE SUM ALGORITHM

Two search algorithms are usually compared based on their speed and amount of memory usage for doing the same job. Most of the time there is a trade-off between the memory usage and processing time. Advantage on one usually leads to disadvantage on the other one. The SUM algorithm does very well in both processing time and memory usage compare to Dijkstra's; so it is a win-win situation, except that it is not as general an algorithm as Dijkstra's is.

With a small change in the *pool* data structure we are able to use an even smaller amount of memory and end up with a faster computation. The two issues are explained in detail in the next two sections.

7.1 Improvement to memory usage

A small change in the *pool* data structure saves even more in the amount of memory used in the algorithm. The amount of memory needed for execution of SUM is bounded by the amount of memory that represents the layout (i.e. the **Layout** data structure). Compared to **Layout**, the other major data structures in the program – i.e. **Net**, **Node**, **C1**, **R2**, and **C3** - use very small amount of memory. A rough estimation of the required memory for execution of SUM on the BCD data is as follows.

The memory needed for holding layout matrix in the memory:

*2143 (y direction) * 1291 (x direction) * 3 (layers) * 2 (integer) ≅ 17 megabytes*[12]

The **Layout** is needed because at the end of the procedure the algorithm must provide the output. Notice that layout is not used in any stage of the algorithm except for output generation, at the

[12] By comparison the amount of memory that the *pool* uses is (≅ 80 bytes) * (1291 + 1291 + 2143) ≅ 378,000 bytes

end of the algorithm. This is not true for either PAM or the Dijkstra-based algorithm. In PAM we need the layout to be updated continuously and the algorithm uses it to check for routing feasibility. The same is true for the Dijkstra-based algorithm. With a minor change in the structure of *pool* we are able to regenerate the layout from the information in *pool*; so we do not need to keep the layout in the memory anymore. This is doable by a little change in the format of the data structure of the *pool*, as follows.

Consider Figure 7.1 and compare its subtle differences with Figure 6.1. Instead of holding the coordinates of the unused segments, the coordinates of the used segments are saved along with a sequence of numbers that shows the indexes of the nets that those segments belong to. The algorithm works like before. The only change is that instead of checking the beginning and end coordinate of a segment for availability of that segment, the end of previous segment and the beginning of next segment is checked to see if the in-between segment is free (see Figure 7.1).

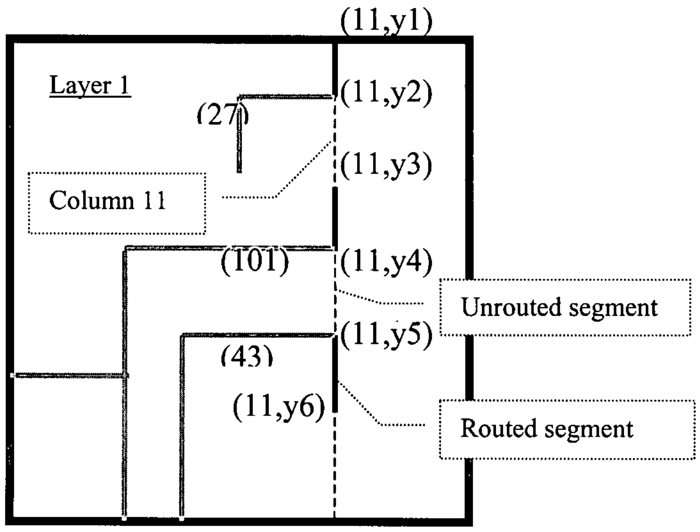


Figure – 7.1 A small change in building up the segment data pool enables the algorithm to generate the layout from the *pool* data structure without having to keep the layout in the memory. Instead of indexing the coordinate of free segments it indexes the coordinates of the taken segments. We also need to associate each segment with the index of the tree that it uses.

Each element of the arrays keeps three sets of information. (i) The number of allocated segments (the segments that are already occupied by routed paths) in that row or column, (ii) the begin-end addresses of the segments, and (iii) an array whose elements contain the index of the tree that the corresponding segment belongs to. In Figure 7.1 the 11th element of C1 is filled as follows.

C1[11].n = 3

C1[11].segment[0] = (y1):(y2)

C1[11].segment[1] = (y3):(y4)

C1[11].segment[2] = (y5):(y6)

C1[11].segn[0] = 27

C1[11].segn[1] = 101

C1[11].segn[2] = 43

The new data structure can always reproduce the layout matrix completely. Therefore, there is no need to keep the layout in the memory. This technique saves around 17 megabytes of memory in BCD data. It means that by employing the new data structure, the amount of memory usage in SUM is bound by only the pool data structure in the program, which is very small.

7.2. Improvement to the completion rate (revised SUM)

From the very beginning, the idea behind the assignment was shaped based on the idea of routing as close as to an ideal solution. Based on this idea the algorithm starts from an optimal route (that most of the time is a 'L' or 'Γ' pattern), checks for feasibility and, in case that the choice is infeasible, moves to the sub-optimal patterns until it finds a feasible solution. Search for optimality dictates that we had better start with the 'L' pattern (with the least *cost*), and then increase the number of segments and check 'Z', 'W', 'U', etc. sequentially until a feasible solution is found. However, the idea of segment pooling in SUM suggests a different search order. The idea is that it might be more productive – in terms of successful routing percentage – if the algorithm starts by searching for patterns with more segments (e.g. 'W' patterns) – and consequently smaller segments – rather than the ones with fewer segments (e.g. 'L' patterns) that need big segment allocation. The justification behind this argument is that the number of large segments is limited in the pool. If the large segments are used at the early stages by 'L'-pattern routes then they consume all of the useful segments and the leftover segments are a bunch of small useless ones. Segment "utilization" suggests that it might be more fruitful if first we test the patterns with more segments - and consequently the smaller ones. The reason is that the choice of smaller segments ends up with bigger leftover segments, which possibly are more useful in the future. We ran the SUM algorithm with changing the order of selected patterns (we call the new algorithm: *revised SUM*, or *r-SUM*). Revised SUM considers the pattern routing functions (i.e. x_**Routing()**) in the following order: 'W', 'Z', 'L', 'U'[13]. The results in Table-A

[13] 'U' still remains after 'L', 'Z', and 'W' because all those three patterns have smaller *length*s (and *cost*s) than 'U'.

show that the above argument is empirically correct (compare the result of SUM with *revised-SUM*).

One critique against this approach is that if this argument is valid then why not start with even more jagged patterns (hence, smaller segments)? An inspection of the result and comparison with the original SUM shows that the new search strategy takes more time to do the same amount of routing. The reason is that it has to perform more feasibility checking for available segments (patterns with larger number of segments take more time to do feasibility checking). Moreover, the more crooked patterns mean using more vias, which is against one of our primary goals, i.e. using less vias.

Notice that, by choosing the patterns with more segments, fragmentation in the layers gets worse. This observation, somehow, contradicts the primary idea of minimizing the fragmentation of *pool_segments*. It seems that although revised-SUM still tries to avoid fragmentation (because the algorithm is inherently Segment Utilization) - at the same time - it tries to optimize the length of leftover segments for future use. The outcomes show that mixing these two considerations together should end up to a better result than considering only one. However this argument, like the any others, needs more testing to confirm.

All together, the processing time difference between the two algorithms suggests that the *revised SUM* should be preferred as long as it does not go too far in reversing the order of patterns.

8. OPEN PROBLEMS FOR FURTHER RESEARCH

Our algorithm is only one heuristic approach to one specific problem. There are also other heuristic approaches that are able to attack the routing problem from different angles.

One of the most interesting areas of expanding this research is developing parallel algorithms for even a faster search. In fact, PAM has the potential to execute on parallel computers. How it can be extended to SUM and how much improvement will be accomplished are challenging topics to explore.

Moreover, our approach suffers from lack of a solid theoretical foundation. The question of "how can we improve the algorithm?" does not have any non-experimental answer. An intuitive observation of SUM indicates that it can be formulated as a Markov process [12]. Formulation of the algorithm as a Markov Decision Process (MDP) might be helpful to develop a solid foundation of the method. Because of producing a huge decision tree, the MDP formulation does not help to solve the problem, but it might help answering some unanswered questions, like under what conditions a 100% routing is doable.

APPENDIX A. Why 100% routing of the BCD data is not achievable

One of the basic goals of automated routing is achieving hundred percent routing. Even leaving a few percent of un-routed nets means that they have to be done manually. Manual routing has drawbacks. First, it contradicts the goal of automating routing. Second, manual routing is usually very time-consuming and it might also need changes in the already routed paths in order to provide rooms for the next routings. This process usually ends up with an inefficient routing, hence should be avoided as much as possible. However, achieving 100% routing implies that there must be enough room so the algorithm is able to complete its job. We prove that there is not enough space in the given BCD data so it is not 100% routable.

Table-B shows the number of horizontal and vertical available edges, and the number of edges needed if all the trees are ideally routed, i.e. on minimal Steiner trees.

The type of nets in BCD data lets us to compute the minimum number of required edges in each direction easily. A simple code can compute the MRST for the nets with two or three nodes. For the nets with more than three nodes we were fortunate there were only 44 nets with 4 nodes and only one with 5 nodes. We computed their optimal route manually (we might have made a mistake and our computation were not exact, but it does not matter very much because the difference is negligible to the scale of the data – the numbers in table B are dominated by 2 and 3 node nets).

Total # of vertical edges (2 layers)	2,766,613 x 2 = 5,533,226
Total # of horizontal edges	1291 x 2143 = 2,766,613
Minimum number of needed vertical edges	(5510 pairs) = 4,464,637
Minimum number of needed horizontal edges	(5510 pairs) = 3,514,991
Maximum number of excessive edges vertical	row#3 – row#1 = 1,068,589
Minimum shortage of horizontal edges	row#4 – row#2 = 748,378

Table B - The BCD data minimum edge requirement for routing. Row 3 and 4 are calculated based on the optimistic optimal route. In the real applications it is very idealistic to achieve this solution. The above calculation provides a lower bound to the number of required edges in each direction.

Notice that, although there is an excess amount of vertical edges, it is rarely possible to lend those extra edges to the horizontal segments to compensate the shortages in the middle layer. The routed segments are so densely distributed that they hinder any replacements of this kind. A small procedure has been developed that runs at the end of the algorithm and examines all the horizontal edges to see if it is possible to transfer any of them from the middle layer to either the upper or the lower layers (procedure **ShiftSegment()** in appendix C). It was only able to transfer three segments of length one!

Another thing that worth noticing about the above lower bound is that one might argue that in some cases it might be possible to change to a non-optimal route that uses more vertical than horizontal edges, which would lead to a decrease in the number of horizontal edges – see Figure set [A.1].

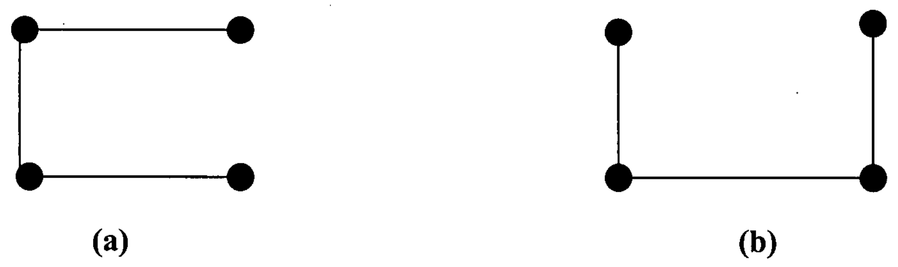


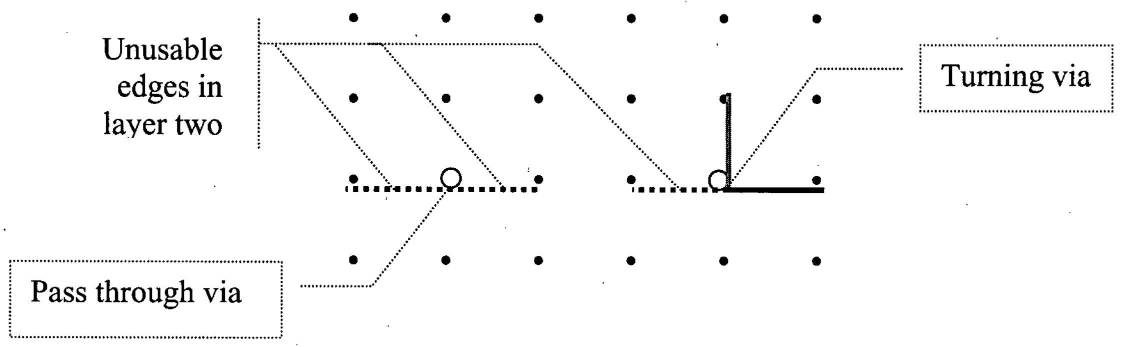
Figure A.1.1 - Consider the above 4-node tree and the two ways of routing it. Option (b) is more desirable in our case because it uses fewer horizontal edges, which are more critical in the BCD layout.

Figure A.1.2 Consider the above 5-node tree and the two ways of routing it. Routing (b) is not a minimum path but because is vertically oriented it can save more horizontal edges so is more desirable.

The first argument is correct and is one of the areas that could be improved in the algorithm. The second argument, although is generally correct, for our given data it is not very helpful in our case and for solving the BCD.

The second consideration can only save horizontal edges in multiple-node trees with more than three nodes. In the BCD data, there are only 57 four-node trees plus one 5-node tree. Employing the above consideration can only save a small number of horizontal edges[14], which will not change the result.

The shortage of horizontal edges is worse than what is calculated in the Table-B. In practice there are circumstances that some edges can not be used by any route at all. Figure A.2 demonstrates the situations that the existence of vias causes some edges to become unusable.



[14] The computation shows that only 4507 horizontal edges will be saved in the four-node trees and nothing in the 5-node one.

Figure A.2 - Each use of a 'via' makes one or two edges around it unusable. If it is a turning via then one edge in layer two becomes unusable to the other paths. If it is a pass through one – i.e. changing surface from layer one to layer three but not the direction – then the both attached edges in layer two become unusable.

As a lower bound, approximately, 6125 via are used[15]. Therefore the total amount of shortages in the horizontal edges is (748378 + 6125) = 754,503 edges.

APPENDIX B. The Software Codes

The complete software code is available in the COE lab, upon request.

REFERENCES

[1] "Wire-path Assignment problem" Technical report by Susan Martonosi; Cornell University (Spring 1999)

[2] "Provably good global routing by a new approximation algorithm for multicommodity flow" by Christoph Albercht, 1999

[3] "On Optimal Interconnections for VLSI" by Andrew B. Kahng and Gabrial Robins, Kluwer Academic Publishers, 1995.

[4] "Combinatorial Algorithms for Integrated Circuit Layout" by Thomas Lengauer, John Wiley Publisher, 1990. The Steiner tree is the subject of almost all of the books on Integer Programming and Network Flow Programming.

[15] (4954 x 1 via used in a 2-node tree) + (498 x 2 vias used in a 3-node tree) + (57 x 3 vias used in a 4-node tree) + (1 x 4 vias used in a 5-node tree) = 6125 approximate number of vias used (it can be smaller number if two terminals in one tree are in the same horizontal or vertical level).

[5] “Minimal Networks: The Steiner Problem and Its Generalizations” by A. O. Ivanov and A. A. Tuzhilin, CRC Press, 1994.

[6] “The Steiner Tree Problem” by F. K. Hwang, D. S. Richards, and P. Winter, North-Holland, 1992.

[7] “Combinatorial Algorithms for Integrated Circuit Layout” by Thomas Lengauer, John-Wiley Publication, 1990

[8] “Algorithms for VLSI Physical Design Automation” by Naveed Sherwani, Kluwer Academic Publishers, 1995

[9] “An Artificial Intelligence Approach to VLSI Routing” by Rostam Joobbani, Kluwer Academic Publishers, 1986. He accepts the definition from other sources. The definition is modified to fit our needs.

[10] The Dijkstra’s algorithm is a classic combinatorial algorithm so that all books written on Integer Programming and Network Programming cover. One good reference is Kenji Ikeda’s web site at “http://www-b2.is.tokushima-u.ac.jp/~ikeda/suuri/main/index.shtml” that also provides detailed example shows of the algorithm.

[11] “Rectilinear Group Steiner Trees and Applications in VLSI Design” by M. Zachariasen and A. Rohe, 2000