MASTER

K-layer topological routing

Jansen, J.W.H.

Award date:
1993

Link to publication
K-Layer Topological Routing

By J.W.H. Jansen

Master Thesis

performed on August 12, 1993
by order of Prof. Dr. Ing. J.A.G. Jess
supervised by Ir. E.P. Huijbregts

The Eindhoven University of Technology is not responsible for the contents of training and thesis reports.
Contents

Abstract iii

1 Introduction 1

2 Topological routing. 3
   2.1 The topological routing problem 3
   2.2 Modeling the topological routing problem 4
   2.3 Basic steps. 7

3 Determination of the net routes 9
   3.1 Definitions 9
   3.2 Routing using the routing graph 10
      3.2.1 Usage of nets in the routing graph 10
      3.2.2 Net routing in linear time 11
      3.2.3 Setting up the maze router 12
   3.3 Complexity 12

4 Net crossings and edge ordering 13
   4.1 Definitions 13
   4.2 Ordering the nets on the edges 14
   4.3 Complexity 17

5 k-Maximum independent set 19
   5.1 Definitions 19
   5.2 A heuristic to find the k-MIS 20
      5.2.1 Explanation of the k-MIS heuristic 20
      5.2.2 Description of the k-MIS algorithm 20
   5.3 k-MIS and routability 21
   5.4 Complexity 22

6 Routability. 23
   6.1 Checking routability 23
   6.2 The algorithm for checking routability 26
      6.2.1 The algorithm to check routability 26
      6.2.2 Finding the polygon in the check cut algorithm 27
   6.3 Complexity of the algorithm 28

7 Vias. 29
   7.1 Properties of vias 29
   7.2 Problem definition 29
7.3 The algorithm (overview) ........................................... 30
7.4 Creating a polygon around the vianet ............................. 30
7.5 Ordering the hulledges .............................................. 31
7.6 Determination the topological viapositions ......................... 34
7.7 Searching candidates for via placement ............................ 36
7.8 Via placement ....................................................... 37
7.8.1 Placing the nets over the 3 new edges. ......................... 38
7.8.2 Determining the exact viaposition .............................. 38
7.8.3 Placing the other nets in the triangle ......................... 41
7.8.4 Routability after placing a via ................................. 42
7.9 Complexity ......................................................... 42
7.10 Placing all vianets ............................................... 43
7.11 Complexity of placing all vianets. ............................... 43

8 Conclusions ......................................................... 45
8.1 Implementation .................................................... 45
8.1.1 Determination of the netroutes ................................. 45
8.1.2 Calculation of the net crossings ............................... 45
8.1.3 k-MIS ......................................................... 46
8.1.4 Routability check .............................................. 46
8.1.5 Viaplacement ................................................... 46
8.2 Future work ....................................................... 47
8.3 Conclusions to the complete router .............................. 47

A Implementation ...................................................... 49
A.1 About the program and the system ............................... 49
A.2 The inputfile ..................................................... 50
A.3 The outputfile .................................................... 51
A.4 The datastructures ............................................... 51
A.4.1 Datastructure for points ..................................... 51
A.4.2 Datastructure for edges ..................................... 52
A.4.3 Data structure nets .......................................... 52

Bibliography ........................................................... 55
Abstract

This thesis describes a $k$-layer topological router that will be part of a new routing system being designed at the Technical University of Eindhoven. Starting with the modules which are placed in $k$ layers and a list of terminals that have to be electrically connected, the router generates a Rubber Band Sketch (RBS). Such a sketch represents the minimal wire length route of all nets in each layer of the design.

A number of heuristics have been designed to find a solution for this so-called Unconstrained Via Minimization-problem. First the routing area is divided into triangles (triangulated). Then all nets are routed. After this, for each pair of nets it is decided if these nets cross. Now as many nets as possible are placed in the $k$ layers, using a $k$-MIS heuristic. A routability check is done to assure that the selected nets can actually (read: geometrically) be placed. At last for the remaining nets the route is divided over two adjacent layers, using vias. The number of vias needed to accomplish this is minimum for these two layers.

The steps above are a so-called backbone algorithm. This means that all basic needed steps are there, but a lot of work still could be done to extend the class of problems the router can handle.

The topological router generates a sketch and all technology-constraints must be checked. If there is a violation of these rules, another sketch will be generated or the router will decide that no sketch can be generated. The most expensive part in the complete topological routing algorithm is this routability check. Its complexity turns out to be quadratic in the number of points in the sketch. At this point there is no successful method to decrease this complexity. The total complexity of the router is $O(k \times (N_N \times N_P)^2)$, with $N_P$ the number of points, $N_N$ the number of nets (pairs of terminals to be electrically connected) and $k$ the number of layers.
This thesis concerns a small part of the silicon compiler project of the Eindhoven University of Technology. A silicon compiler is a computer tool for automatic design of integrated circuits. Basically, given a functional specification of the circuit that has to be implemented in hardware, the silicon compiler should create a layout implementation of this circuit. On the path from specification to implementation, a number of steps can be distinguished:

![System Overview Diagram](image)

A functional specification is an algorithmic description of the desired behaviour of the system using any suitable language. High level synthesis creates a network of hardware components (modules) that implement the algorithmic description, and a controller (finite state machine). The modules are connected to the outside world by pins (terminals). Tasks to be performed here are e.g. scheduling and allocation. The next step is logic synthesis, which uses the controller and/or the description of combinatorial logic. It results in a list of all connections (netlist) between the pins of the modules for which the layout of the IC is known. Finally the layout synthesis generates a set of masks given a set of modules and a list of all required connections.
The scope of this thesis concerns the **layout synthesis**, or low-level synthesis. Although it is at the bottom of the design-traject, it is not the least important part! The quality of the final result depends greatly on this phase.

A chip is usually built on a piece of silicon, called a *wafer*. In each step a new *layer* of material is placed on the wafer, using a so called *mask* for that layer. So what should be placed where is defined by a set of masks. The number of masks used is determined by the technology used to implement the chip. There are a number of *designrules* to make sure the layout will be implemented correctly, e.g. the minimal distance of two pieces of the same material on the layout. These constraints are strongly dependent on the technology used.

The purpose of layout synthesis is to place all modules (from which the layout is already known) and wire all connections in the netlist, to create a complete layout of the integrated circuit. In this step demands like no violation of technology constraints, minimum area and minimum crosstalk between nets should be taken into consideration.

The routing of the nets can be divided into the following steps:

- **topological routing.** This step must find a topological route for each net, consisting of the layer the net is placed in and between which modules the net has to be placed to interconnect its terminals. Furthermore the topological router must only generate so called sketches that can actually be implemented, that is: no technology-constraints may be violated.

- **geometrical routing.** Here the *exact geometrical* route of each net has to be found. This step uses the information generated by the topological router. It is also possible that the layout generator tries to create a geometrical route for each net without topological routing. However then it is extremely difficult to optimize the route for each net and in many situations the router will not be able to generate a legal route at all. A route is legal if none of the technology-constraints is violated and if all connections in the netlist are established.

The geometrical router will generate *rectilinear* routes for the nets, which can be directly transformed to wires in the layout. The topological router however generates a *Rubber Band Sketch*. This is a sketch in which the geometrical positions of all modules and terminals are determined, but the nets are only topologically determined.

This thesis describes the design of a **topological** router, which generates a *Rubber Band Sketch*. This *Rubber Band Sketch* can then be used by the geometrical router to generate a set of masks.
Chapter 2

Topological routing.

This chapter describes the topological routing problem and defines a suitable model for it. After this, an algorithm is proposed to solve the k-layer topological routing problem.

2.1 The topological routing problem

Let $M$ be a set of modules. Each module has terminals located along its boundary. Furthermore terminals can be located along the boundary of the chip, to connect the chip with the outside world. Let $P$ be the set of terminals. A netlist is given, which decides which of the terminals should be connected. In this report we assume that only two-terminal connections have to be made.

The problem is to topologically wire all connections in the netlist using $k$ wiring layers ($k$ dependent on the technology used), in such way that in each layer none of the interconnections cross. The wires may switch layers using vias to avoid crossings. Furthermore no technology constraints may be violated.

The topological routing problem can be divided into four steps:

- Finding a topology for each net.
- Layer assignment.
- Routability check.
- Via placement and minimization, using the routability check.

The second step and the last step are NP complete and therefore heuristics have to be found to find a polynominal-time solution for them. Proof of NP-completeness of step 4 can be found in [Sarr89], proof of NP-completeness of step 2 can be found in [Gare79].

There are two types of via minimization. The first one is Constrained Via Minimization (CVM). Given the location of all nets and each possible via, find a layer assignment for the wire segments and a minimum subset of the possible vias that make sure that no crossings between nets occur [Kuo88] [Stal90].
Topological Routing

The second one is *Unconstrained Via Minimization* (UVM). This is a much broader problem. Here, a routing topology, layer assignments and vias (place and number of) have to be found. The problem addressed in this thesis is the UVM problem. In most literature heuristics for the UVM problem are only designed for a fixed number of layers (often 2 or 3 layers). We try to find an algorithm that can deal with a variable number of layers.

Also, most routers stick to the so-called switchbox routing-problem or the channel routing problem [Cole84][Pin83][Haru92]. This means that no obstacles (modules or vias) appear in the area the nets have to placed in. All terminals are positioned at the border of the area. In the initial situation these problems have an empty area. We try to deal with internal blockings in a routing area here, so in the initial situation the routing area does not have to be empty at all. Also, nets might be attached to the modules by terminals. This is a far more difficult problem, as will be explained later.

### 2.2 Modeling the topological routing problem

Now the topological routing problem is stated, we will construct a proper model for it. Given a discrete three-dimensional space \( \mathbb{N}^3 \)

**Definition 2.1** A *position* is a triple \((x,y,z)\), with \(x, y, z \in \mathbb{N}\).

**Definition 2.2** A *plane* or *layer* is defined as \(L_k = \{(x, y, z) | z = k\}\)

**Definition 2.3** A *point* \(p\) is a plane-independent position, with coordinates \((x_p, y_p)\).

Points are used to indicate modules, terminals and vias.

**Definition 2.4** A *feature* is a straight line between two points. A feature cannot be crossed by other features.

**Definition 2.5** A *module* is an object in the routing area consisting of features in such a way that the module is a “closed” figure.

A module can be seen as a blocking; no wires are allowed to cross it on any layer.

These definitions state that in our model modules and terminals are identical on each layer and as a result terminals can be accessed on any layer. An example is given in figure 2.1.

**Definition 2.6** A *via* is a connection between 2 adjacent layers. For convenience, a via is also modelled as a point.

Physically a via uses some space due to technology constraints, therefore introducing a blocking. This could also be modelled as a set of features.

The next step is to model a topological connection. In chapter 1 we stated that a topological route of a net consists of “between which modules the net is placed” and a layer. The space to place nets can be defined by edges.
Definition 2.7 An edge $e$ defines the geometrical space between two points $p_1$, $p_2$. An edge is defined as a straight directed line from $\text{begin}(e) = p_1$ towards $\text{end}(e) = p_2$ and can also be denoted by $(p_1, p_2)$.

The edge is said to be connected to both points, and the points are said to be connected to the edge.

Definition 2.8 The routing area $(P, F, E, L)$ is defined by:

- $P$: set of points
- $F$: set of features, $F \subseteq P \times P$
- $E$: set of edges, $E \subseteq P \times P$, $E \cap F = \emptyset$
- $L$: set of $(k)$ layers

The only thing that needs to be determined in this definition is the set of edges $E$. To find these we create a so-called triangulation of the set of points $P$.

Definition 2.9 Given $p_1, p_2, p_3 \in P$. Let $e_1$ denote $(p_1, p_2)$, $e_2$ denote $(p_1, p_3)$ and $e_3$ denote $(p_2, p_3)$. If $e_1, e_2, e_3 \in E \cup F$ then $(p_1, p_2, p_3)$ forms a triangle. $(p_1, p_2, p_3)$ may also be denoted as $(e_1, e_2, e_3)$.

Definition 2.10 A triangulation $T$ of a set of points $P$ is a straight-line planar graph such that the edges of the convex hull are part of $T$ and the inside of this hull is divided in triangles such that every point is only a corner point of one or more triangles.

An example of a triangulation is given in figure 2.2.

In fact we create a so-called Constrained Delauney Triangulation (CDT). This CDT demands that the features are a part of the triangulation and it guarantees we have $O(1P1)$ edges and/or features [Ruys92].

Lemma 2.1 Given is a set $P$ of points. Let $N_P$ the number of points in the routing area, $N_h$ the number of points in the convex hull of $P$. The number of edges $N_E$ in the CDT is given by $N_E = 3(N_P - 1) - N_h$ and number of triangles $N_T$ in the CDT is given by $N_T = 2(N_P - 1) - N_h$.
After creating the CDT, the routing area has a hull: \( E_h \) is the set of edges and/or features bounding the routing area, \( E_h \subseteq E \cup F \). Now we have completely defined the routing area, we can define a topological netroute or net.

**Definition 2.11** A net \( n \), or topological netroute can be modelled as:

- \((p_1, p_2)\): a pair of points \( n \) should connect. The net is defined from \( p_1 \) towards \( p_2 \).
- \( P(n) \) is the ordered set of edges net \( n \) crosses to connect point \( p_1 \) with point \( p_2 \), called the topology or path of the net.

A topology of a net can be converted into a geometrical route [Ruys92].

**Definition 2.12** A rectilinear route of a net is a route in which every wire can have only two directions: horizontal and vertical. A geometrical route of a net is a rectilinear route which is exactly determined (positions, wire widths).

An example of topological routes and possible resulting geometrical routes is given in the figure 2.3

In this thesis we assume that a multiterminal net is partitioned into several 2-terminal nets that are routed sequentially.
Finally we need to know if two nets cross each other.

**Definition 2.13** A *planar embedding* of a topology \( P(n) \) for net \( n \) is a route such that \( n \) only crosses edges \( e \in P(n) \) and furthermore each edge \( e \in P(n) \) is only crossed once. \( \square \)

**Definition 2.14** Two nets \( n_1, n_2 \) cross if all possible planar embeddings of \( P(n_1) \) and \( P(n_2) \) have at least one point in common. \( \square \)

According to this definition in figure 2.3a, net 1 and net 2 cross, and net 3 and net 4 cross.

### 2.3 Basic steps.

Having introduced the routing area and nets, we now can now state a heuristic that might solve the \( k \)-layer routing problem in polynominal time. The complete problem is divided in a number of basic steps. Of course there are numerous ways to come to a solution. The idea is to keep each part of the problem clear to understand and easy to adapt to the properties of the area to be routed.

Many of the steps are NP-complete, so a number of heuristics have to be found. This means that these steps itself will result in a sub-optimal solution.

The most important demands here are:

- If we generate a sketch, it must be possible to geometrically place all nets.
- The amount of vias should be kept as small as possible.

We distinguish the following steps:

1. Make a CDT. This part is extensively described in [Ruys92].
2. Find a topology for each net. A number of possible routes can be stored to be able to pick the best of a number of possible routes for a net.
3. Determine the crossing-graph.
4. Assign as many nets as possible to the \( k \) layers. This problem equals finding a \( k \)-Maximum Independent Set (\( k \)-MIS) on the crossing graph.
5. Do a routability check for all nets placed in the \( k \)-MIS. In case a net is not routable, try another realisation of the net or find another \( k \)-MIS.
6. Reroute all remaining nets if possible and place these nets in the MIS of the layer they fit in.
7. For all remaining nets, place these nets on 2 adjacent layers, with the use of vias, in such a way that no nets cross each other. If a net cannot be placed on any 2 adjacent layers, the problem is not routable with this algorithm.
Determination of the net routes

In this chapter we create routes for all nets, given a routing area and two terminals per net.

3.1 Definitions

Suppose we have a routing area \((P, F, E, L)\), let \(E_h \subseteq E \cup F\) be the set of hulledges. For routing purposes we need a special graph, the so called routing-graph.

**Definition 3.1** The *routing graph* is a graph \(G_r(V_r, E_r)\) in which

- \(V_r\) is a set of vertices. Let \(\delta: E \setminus E_h \rightarrow V_r\) be a function that assigns a vertex \(v \in V_r\) to each edge \(e \in E \setminus E_h\).
- \(E_r\) is a set of edges; each edge is placed between two vertices if these vertices are part of one triangle in the routing area.
- \(C: E_r \rightarrow \mathbb{R}^+\) a cost function that assigns a cost to each edge \(e \in E_r\).

![Figure 3.1: Routing area and routing graph.](image-url)
In figure 3.1 an example of a routing area and the corresponding routing graph is given. The solid points and edges denote the routing area; the dashed lines and points represent the routing graph. Note that in the routing graph the vertices do not have a geometrical position, the vertices in the routing area do. Important is the fact that if we travel from one vertex to another vertex over an edge in the routing graph, we actually cross a triangle in the routing area.

The edge-costs in figure 3.1 are randomly chosen. This cost should be a measure of the "difficulty" to travel the edge.

Definition 3.2 Let $e \in E$ be $(p_1, p_2)$. Define middle $(e)$ as

\[
\text{middle} (e) = ((x_{p_1} + x_{p_2})/2, (y_{p_1} + y_{p_2})/2).
\]

Let $e_r \in E_r$ be $(v_1, v_2)$ with $v_1, v_2 \in V_r$. Let $e_1, e_2 \in E$ such that $\delta(e_1) = v_1$ and $\delta(e_2) = v_2$. Then $\text{cost}(e_r)$ is the distance between middle$(e_1)$ and middle$(e_2)$. Such a cost function can be used to minimize the (total) length of the netroutes.

3.2 Routing using the routing graph

3.2.1 Usage of nets in the routing graph

For each net the router should find a path of edges. As defined, a netroute starts and ends on a point in the routing area. However points in the routing area are not mapped on the routing graph. So to find a route for a net we must locally extend the routing area with two vertices: a source and a target.

Definition 3.3 The source vertex $v_s \in V_r$ corresponds to the begin point $p_b \in P$ of the net that has to be routed. The target vertex $v_t \in V_r$ corresponds to the end point $p_e \in P$ of the net that has to be routed.

Definition 3.4 An edge $(p_b, p_e) \in E$ is opposite to a point $p \in P$ if the edges $(p, p_b) \in E$ and $(p, p_e) \in E$ exist.

This can be seen in the following figure:

Let $\text{Opp}(p)$ denote the set of edges that are opposite to a point $p$ in the routing area. Thus:

\[
\text{Opp}(p) = \{ e \in E | e \text{ is opposite to } p \}
\]
Let $p_b, p_e$ be the begin and endpoint of a net. Introduce two vertices $v_s, v_t$ in the routing graph. Let $E_{add} = \{(v_s, \delta(e))| e \in Opp(p_b)\} \cup \{(v_t, \delta(e))| e \in Opp(p_e)\}$

Now we extend the routing graph as follows:

$V_r = V_r \cup \{v_s, v_t\}$

$E_r = E_r \cup E_{add}$

An example is given in the following figure.

![Figure 3.3: Example of extended routing graph.](image)

In figure 3.3a, $Opp(p_s) = \{e_a, e_b\}$ and $Opp(p_t) = \{e_c, e_d\}$. Thus $E_r = E_r \cup \{(v_t, v_c), (v_t, v_d), (v_s, v_a), (v_s, v_b)\}$. The fat dashed lines indicate the edges added to $E_r$.

### 3.2.2 Net routing in linear time

**Theorem 3.1** Both the number of vertices and the number of edges in the routing graph are linear with respect to the number of points in the routing area.

**Proof:** Let $N_E = |E|$, $N_P = |P|$, $N_h = |E_h|$, and $N_T$ be the number of triangles in the routing area. According to lemma 2.1, $N_T = 2(N_P - 1) - N_h$ and $N_E = 3(N_P - 1) - N_h$. Let $N_{Th}$ be the number of triangles that have an edge $e \in E_h$. The number of points on the hull of the routing area $N_{ph} = N_h$.

Let $n_v = |V_r|$, $n_E = |E_r|$ be the number of vertices resp. the number of edges in the routing graph. The following properties hold:

- $n_v$ is equal to $N_E - N_h$, which is linear to $N_P$ because $N_h \leq N_E$, and $N_E$ is linear to $N_P$.
- $n_E$ is equal to $3N_T - 2N_{Th}$, because for each triangle $t$ in the routing area there are 3 edges in the routing graph, except for the triangles that have an edge $e \in E_h$. For these triangles there is only one edge in the routing graph.

$N_{Th} = N_h$, so $n_E = 3(2(N_P - 1) - N_h) - 2N_h = 6(N_P - 1) - 5N_h$, which is also linear to $N_P$.

**Conclusion:** both the number of vertices $n_v$ and number of edges $n_E$ are $O(N_P)$.

**Lemma 3.1** Routing of each net can be done in time linear to $|N_P|$.

**Proof:** For the actual routing we use a maze router [Lee61] on the routing graph. Such a router uses “bi-directional search wave expansion” and with $n_E = O(N_P)$, routing of a single net can be done in $O(N_P \cdot C)$, with $C$ the maximal cost in the routing graph.
3.2.3 Setting up the maze router

Given a net n, with begin point $p_b \in P$ and end point $p_e \in P$. Extend the routing graph with a source $v_s$ and target $v_t$, and the set $E_{add}$. Now we can start the maze router, to find the cheapest path from $v_s$ to $v_t$, adjust the cost for each edge on the path and try to find the next cheapest net realisation. We continue this until we are satisfied with the number of net realisations. The extensions of the routing graph are then removed. We do the same for the next net until we routed all nets.

Note that features have no vertex in the routing graph. This means the router cannot use features in paths, so nets can never cross features.

3.3 Complexity

As described in section 1.1 for each net a route can be found in a time linear to the number of points in the routing area. So the complete complexity is:

$$O(N_N* R_N* N_P + C)$$

with $N_N$ = Number of nets, $R_N$ = number of realisations per net and $N_P$ = the number of points in the routing area, and $C$ the maximal cost in the routing graph.
4.1 Definitions

In the previous chapter, a topology $P(n)$ was found for each net $n$. Now for each edge $e \in E$ the set of crossing nets $N(e)$ is defined by $N(e) = \{n \in N | e \in P(n)\}$. However, this set is not ordered yet. In this chapter we want to know which nets cross each other. Furthermore we try to order the sets $N(e)$ for all $e \in E$ in such a way that for each pair of crossing nets, the number of crossings is minimized.

Given a routing area $(P, F, E, L)$.

**Definition 4.1** A *common point* $CP$ of two adjacent edges relative to a net is the point that connects both edges. Thus:

$$CP(e_1, e_2) =$$

begin $(e_1)$ iff begin$(e_1) = begin(e_2)$ or begin$(e_1) = end(e_2)$

end $(e_1)$ iff end$(e_1) = begin(e_2)$ or end$(e_1) = end(e_2)$

Edges that have a common point are called to be *connected*.

**Definition 4.2** The *other point* of edge $e \in E$ and point $p \in P$, $OP(e, p)$ is equal to:

begin $(e)$ iff $p = end(e)$

dern $(e)$ iff $p = begin(e)$

Let $n$ be a net, with topology $P(n) = (e_1, e_2, ..., e_m)$, begin point $p_b$ and end point $p_e$. Now there is a triangle $t_b (p_b, e_1)$ and a triangle $t_e (p_e, e_m)$. Let $e_i, e_i+1 \in P(n), p = CP(e_i, e_i+1)$. Suppose $e_i = (p, p_1)$ and $e_{i+1} = (p, p_2)$, then edge $e' = (p_1, p_2)$ must exist. So for each pair of adjacent edges $e_i, e_i+1 \in P(n)$ a third edge can be determined that defines a triangle $(e_i, e_{i+1}, e')$. Let the sequence of triangles constructed from $P(n)$ be $T_P(n)$.

**Definition 4.3** The *triangle topology* $T(n)$ of net $n$ is the sequence of triangles $t_b$ followed by $T_P(n)$, followed by $t_e$. 


Definition 4.4 A crossing graph \( G_c(V_c, E_c) \) is a undirected graph in which

- \( V_c \) is the set of vertices. Let \( \gamma : N \to V_c \) be a function that assigns a vertex \( v \in V_c \) to each net \( n \in N \).
- \( E_c \) is the set of edges. An edge is placed between two vertices if the nets corresponding to these vertices cross.

4.2 Ordering the nets on the edges

Initially \( E_c = \emptyset \) and suppose that for each edge \( e \in E \) the set \( N(e) \) initially is empty. Each net \( n \in N \) is placed on the edges \( e \in P(n) \), in such a way that the set of nets \( N(e) \) on each edge \( e \in E \) maintains ordered. Suppose we are placing net \( n \in N \), and all nets \( m, 1 \leq m < n \) are already placed.

- Let \( T(n) = (t_b, ..., t_i, ..., t_e) \) and suppose that net \( n \) already is placed in the triangles \( \{t_b, ..., t_{i-1}\} \). Now net \( n \) has to be placed in triangle \( t_i = (e_i, e_{i+1}, e') \). Net \( n \) is already placed on edge \( e_i \), so \( N(e_i) \) is already ordered for all nets \( m, 1 \leq m \leq n \).
- Add net \( n \) to \( N(e_{i+1}) \) such that a minimal number of crossings is introduced.
- For each net \( m \) that crosses the triangle, we determine if it crosses net \( n \). If so, the crossing graph is extended with an edge \( (\gamma(m), \gamma(n)) \).

Definition 4.5 Let \( n \in N \). Let \( e_i, e_{i+1} \in P(n) \) be connected. Let \( m \in N(e_i) \). Net \( m \) lies left of net \( n \) on edge \( e_i \) iff net \( m \) lies closer to \( CP(e_i, e_{i+1}) \) than \( n \). In the case net \( m \) lies left of \( n \) on \( e_i \), \( n \) is said to be right of net \( m \). The same goes for nets on \( e_{i+1} \).

- If triangle \( t_i \notin \{t_b, t_e\} \):
  - If net \( m \in N(e_i) \) and \( m \in N(e_{i+1}) \) then net \( m \) should have the same relative position (left or right) to \( n \) on \( e_{i+1} \) as on \( e_i \). This is indicated in the figure 4.1.
  - If net \( m \in N(e_{i+1}) \) and \( m \in N(e') \) then net \( n \) should be placed right of \( m \), relative to \( CP(e_{i+1}, e') \). This situation is indicated in figure 4.2.

\[ \text{Figure 4.1: Relative netpositions.} \]
Chapter 4 Net crossings and edge ordering

Figure 4.2: Example of planarizing a routing area.

- If triangle $t_i \in \{t_b, t_e\}$:
  
  -- If $n$ is connected to $p_e$ then $t_i = t_e$ so there is no edge $e_{i+1}$ to place $n$ on. This is indicated in figure 4.3a.

  -- If $n$ is connected to $p_b$ then $t_i = t_b$. Let $e_1$ be the first edge in $P(n)$. Let $e_a = (p_b, \text{begin}(e_1))$ and let $e_b = (p_b, \text{end}(e_1))$. Suppose net $m \in N(e_a)$ and $m \in N(e_1)$, then net $n$ should be placed right of $m$ relative to $CP(e_a, e_1)$. If net $m \in N(e_b)$ and $m \in N(e_1)$, then net $n$ should be placed right of $m$ relative to $CP(e_b, e_1)$. This is indicated in figure 4.3b.

Figure 4.3: Example of planarizing a routing area.

Now we have placed net $n$ on edge $e_{i+1}$, we want to decide which of the nets that cross triangle $t_i$ cross net $n$. For all nets $m \in N$ on edge $e_i$ we already know if $m$ if is left or right of $n$ on $e_i$. The same goes for edge $e_{i+1}$. With this knowledge we can decide if nets $n$ and $m$ cross, which is indicated in the following figure:
If \( n \) is connected to a point \( p \in \{p_b, p_c\} \) nets \( n \) and \( m \in N \) cross if:

- \( m \in N(e'') \) and \( m \in N(e_i) \), \( m \) right of \( n \) on \( e_i \), relative to \( \text{CP}(e_i, e'') \)
- \( m \in N(e') \) and \( m \in N(e_i) \), \( m \) right of \( n \) on \( e_i \), relative to \( \text{CP}(e_i, e') \)
- \( m \in N(e') \) and \( m \in N(e'') \).

In figure 4.4b, \( n \) crosses 2 edges. In this case only nets of type \( m \) can cross with \( n \), which means \( m \in N(e') \) and \( m \in N(e_i) \), \( m \) left of \( n \) on \( e_i \) (rel. to \( \text{CP} \)).

Only nets \( m \in N \) that correspond to one of the cases indicated in figure 4.4 cross net \( n \). If such a net is found, an edge \((\gamma(n), \gamma(m))\) is added to \( E_c \).

A net \( m \in N, m \neq n \), connected to a point \( p \in P \), is regarded as crossing an edge \( \epsilon \in t_i \) infinitely close to \( p \) on an edge \( \epsilon \in t_i \) that is connected to \( p \). This is indicated in the following figure.

As can be seen in figure 4.5, net \( m \) can be faked infinitely close to point \( p \).
Let $N_N$ be the number of nets, and $N_E$ be the number of edges in the routing area. For each net we travel its path. Worst case each net crosses each edge of the routing area and worst case all nets cross each other, so the worst case complexity of this step is:

$$ O (N_E^* N_N^* (N_N-1)/2), $$

which is equal to:

$$ O (N_E^* N_N^2), $$
Chapter 5

k-Maximum independent set

In this chapter we try to place as many nets as possible in the \( k \) layers by finding a \textit{k-Maximum Independent Set} on the crossing graph.

5.1 Definitions

Definition 5.1 An \textit{Independent Set (IS)} of a graph \( G(V, E) \) is a subset \( S \subseteq V \) such that none of the vertices in \( S \) are adjacent in the graph \( G \).

Definition 5.2 A \textit{Maximum Independent Set (MIS)} of graph \( G \) is an independent set of maximum cardinality.

A MIS of the crossing graph \( G_c(V_c, E_c) \) represents a maximum set of nets that can be placed in one layer in the routing area, without introducing net crossings.

Definition 5.3 A \textit{k-Maximum Independent Set (k-MIS)} consists of \( k \) pairwise disjoint subsets of \( V \), each being an independent set, such that the sum of cardinalities is maximum.

![Figure 5.1: Example of a crossing graph.](image-url)
A $k$-MIS of the crossing graph therefore represents a maximum set of nets that can be placed in the $k$ layers without introducing net crossings. An example is given in figure 5.1.

In the crossing graph of figure 5.1b we see that nets a does not cross nets c and d, so a 2-MIS for figure 5.1a can be: $S_1 = \{ a, c, d \}$ and $S_2 = \{ b \}$

The $k$-MIS problem is a well known NP-complete problem [Garb90][Gare79], and some heuristics have been found to find a solution for it, e.g. [Sarr93].

5.2 A heuristic to find the $k$-MIS

Definition 5.4 Let $G(V, E)$ be a graph. The **degree** of a vertex $v \in V$, $\text{degree}(v)$, is equal to the number edges $e \in E$ that is attached to $v$.

5.2.1 Explanation of the $k$-MIS heuristic

First we take the vertex $v \in V$ with the smallest degree. We call $v$ a **winner** and place it in the IS. All vertices that are adjacent to $v$ cannot be placed in the same IS as $v$. We call these vertices **losers**. We delete $v$ from $V$ and we delete all edges attached to $v$ from $E$. Now all losers are deleted from $V$, and also all edges attached to all losers are deleted from $E$. We continue these steps until $G(V, E)$ is empty. The IS is now a MIS. For the remaining graph we repeat the same procedure until $k$ maximum independent sets are found.

5.2.2 Description of the $k$-MIS algorithm

Definition 5.5 Given a graph $G(V, E)$. Let $V' \subseteq V$. The **induced subgraph** $G'(V', E')$ of $G(V, E)$ is a graph such that $E' \subseteq E$ and $E' = \{(v_1, v_2) \in E \mid v_1, v_2 \in V'\}$.

**Algorithm k-MIS**

Given a graph $G(V, E)$

\[\forall \text{layer } l, 0 \leq l < k \{
\begin{align*}
V' &= V_0, IS_0 = \emptyset; V = \emptyset \\
(* & \text{ let } E' = \{(v_1, v_2) \in E \mid v_1, v_2 \in V'\} *) \\
\text{while } (V' \neq \emptyset) \{
& \text{Let } v \in V' \text{ be the vertex with the smallest degree } (* \text{ winner } *) \ \\
& IS_l = IS_{l-1} \cup \{v\} \\
& L = \{ w \in V'|(v, w) \in E' \} (* \text{ losers } *) \\
& V'' = V'\backslash L \\
& V = V \cup L \\
& V' = V'\backslash\{v\} \\
\}
\} \]
For a fast algorithm a so called bucket is used. For each entry this bucket has a list of all vertices that have the same degree. So in bucket-entry 2 all nets that cross with two other nets are listed, in entry 5 all nets are listed that cross with 5 nets and so on. The advantage of using a bucket is the fact that all necessary operations like the retrieval of the net with the smallest degree (mindegree), inserting and deleting can be achieved in constant time.

5.3 \textit{k-MIS and routability}

As can be seen, all nets will be placed in the layers in a \textit{greedy} way. This means it has no interest in dividing the nets over the layers. In other words the algorithm might place too many nets in one layer because they simply do not cross. So if a routability problem occurs it might be possible to try to swap some of the nets (that cause the problem) to other layers.

An example is given in the following figure:

![Figure 5.2: Example of a crossing graph.](image)

In figure 5.2a the routing area is given. Suppose there are 3 layers. Then a possible \textit{k-MIS} is given in figure 5.2b. Suppose that the nets are placed in the layers as indicated. Edge XY might be too small for the two nets that cross it. If, however, we place net 1 in layer 1 then no such problem occurs.
5.4 Complexity

Let $k$ be the number of layers available, $N_E$ the total number of edges, and $N_V$ the total number of vertices in $G(V, E)$.

Inserting and deleting from the bucket can be done in $O(1)$ time, so this involves no extra complexity.

**a:** First we must initialize the bucket. For each vertex we must count the number of edges attached to the vertex. Each edge is connected to exactly two points, so we meet each edge exactly twice. Initializing the bucket therefore costs:

$$O(N_V + 2 \cdot N_E) = O(N_V + N_E)$$

**b:** The MIS algorithm on the crossing graph is based on 3 steps:

1. Take a winner and delete it; this costs $O(1)$.
2. Delete all losers.
3. Updating all nets adjacent to the losers: delete them from the bucket, and insert them one position lower in the bucket.

These steps can be done more than once. The overall complexity of step 1 & 2 is $O(N_V + N_E)$, because each vertex/edge can be removed only once. The overall complexity of step 3 is worst case another $O(N_E)$, because removing each edge leads to an update of exactly one vertex degree.

So if each action would take constant time, the complexity of the MIS would be also $O(N_V + N_E)$. However finding the mindegree for every vertex-deletion is not a constant-time problem.

- Suppose each time the new mindegree must be calculated, we start searching from the bottom of the bucket.
- If a vertex with degree mindegree is removed, exactly mindegree edges must be removed from $G_c(V, E)$. This is however exactly the number of steps that are maximally needed in the bucket to recalculate mindegree, because each edge can be removed only once and the new mindegree is always smaller than or equal to the old mindegree. So finding the new mindegree can be done in the same complexity as deleting all edges connected to the deleted vertex.

So the sum of finding all new mindegrees costs $O(N_E)$.

**C:** This means that the total complexity of the MIS heuristic is:

$$O(N_V + 2 \cdot N_E + N_V + N_E + N_E + N_E) = O(N_V + N_E)$$

And for $k$ layers, this must be done $k$ times, so the complexity of the $k$-MIS is:

$$O(k \cdot (N_V + N_E))$$
Now the $k$-MIS has been determined, we want to know if it is possible to geometrically place these nets in the $k$-MIS. Since we have a topological model, we must have a check on our topological system that is also sufficient for the geometrical model.

### 6.1 Checking routability

To be able to define routability we need a extended definition of our routing area. We call this a sketch.

**Definition 6.1** Sketch $= (P, F, E, L, N)$

where:

- $(P, F, E, L)$: the routing area
- $N$: the set of nets.

A sketch is routable if there exists a geometrical route for each topology and no technology constraints are violated.

A net, defined as a topology with a begin- and an end-point (see def. 2.7), can cross an edge on a certain layer. Thus given an edge of the triangulation and a layer, we know how many nets cross this edge in this layer. Of course this is only possible if for each edge $e \in E$ the set of crossing nets $N(e)$ is known.

**Definition 6.2** A cut is a straight line between two points in the routing area, that crosses no other points.

**Definition 6.3** The capacity of a cut is equal to the maximum amount of nets that can cross the cut in $x$-direction $N_x$ or in $y$-direction $N_y$, thus:

$$
cap(cut) = MAX(N_x, N_y),
$$
**Definition 6.4** A cut is safe if the number of nets that cross it (flow) is smaller than the capacity of the cut. Thus:

\[
\text{safe}(\text{cut}) = \begin{cases} 
\text{True, if } \text{cap}(\text{cut}) \geq \text{flow}(\text{cut}). \\
\text{False, otherwise.}
\end{cases}
\]

There are a number of ways to define routability. The one used here is the next one:

**Theorem 6.1** A sketch is routable if and only if every critical cut is safe

This was defined by Leierson and Maley [Leie85]. The question is: “What is a critical cut?”.

If we know the number of nets that can maximally cross an edge on each layer, we can find out if an edge of the CDT is safe. It would be convenient if we only had to check the edges of the CDT, because then we would only have to check \(O(N_E)\) edges (see Chapter 2). However, this is not the case. As F. Ruys [Ruys92] described, checking only the edges of the CDT will no suffice to guarantee that all nets can be placed. This is indicated in figure 6.1.

![Figure 6.1: Checking only edges of the CDT fails.](image)

The dashed lines indicate a grid on which the nets can be placed. The solid lines between the points (a through e) are the edges of the triangulation. We can easily see that each of the edges of the triangulation is safe. However if there were an edge \((p_a, p_c)\) the capacity of this edge would be 5 and 6 nets would cross this edge. Thus the example is **not** safe. Therefore, we need to have a routine that can also check an “edge” between two points that are not connected by an edge.

**Definition 6.5** A **virtual** edge is a cut between two points that is not an edge or feature of the triangulation.
To be able to check the whole routing area for safeness, we must also be able to check the virtual edges for safeness. The only problem is that these edges do not exist and so we do not know which nets cross it.

What we do is this: we construct a polygon around the virtual edge, that only has points on the hull and no points on the inside. The polygon is built with existing edges. From these edges we know the flow, so it is possible to decide how many nets cross the virtual edge. An example of such a polygon is given in figure 6.2.

![Figure 6.2: Polygon around cut (p_b, p_e).](image)

The solid lines in figure 6.2 are the polygon, the dashed lines are edges (other) edges of the triangulation.

**Definition 6.6** Let's say a cut is defined from point \( p_b \) to point \( p_e \). A point \( p \) lies left of cut \( (p_b, p_e) \) if \( \text{angle}((p_b, p), (p_b, p_e)) > 0 \). A point \( p \) lies right of cut \( (p_b, p_e) \) if \( \text{angle}((p_b, p), (p_b, p_e)) < 0 \).

The polygon is divided into two sides: a left side \( E_l \subset E \) and a right side \( E_r \subset E \). The left side consists of edges \( e_l = (p_1, p_2) \), with \( p_1 \) and \( p_2 \) left of \( (p_b, p_e) \). The right side consists of edges \( e_r = (p_1, p_2) \), with \( p_1 \) and \( p_2 \) right of \( (p_b, p_e) \).

With a polygon the flow over cut \( (p_b, p_e) \) can be calculated by looking at the total flow over \( E_l \) and the total flow over \( E_r \). Because of the fact that there are no points inside the polygon, no nets can start from within the polygon, thus checking all nets that cross the polygon or start or end at the polygon is enough to check how many nets cross cut \( (p_b, p_e) \).

Look at figure 6.3.

![Figure 6.3: Useful net crossings.](image)
In figure 6.3 the only time net $n$ crosses cut $(p_b,p_e)$ is between edge $e_x$ and edge $e_y$. This is because edge $e_x$ and edge $e_y$ are on different sides of the polygon. We call this a useful net crossing. The number of times a net crosses cut $(p_b,p_e) \ V_n$ is now equal to the number of useful net crossings on the path of the net. For each net $n$ we can find $V_n$ using the next algorithm:

- Let $E_l \in E$ be the set of edges on the left side of the polygon, and $E_r \in E$ be the set of edges on the right side of the polygon.
- Let $V_n = 0$.
- For each edge $e \in P(n)$,
  - If $e \in E_l$ and the previous time an edge $\in P(n)$ was found on the polygon, this edge was $\in E_r$ then $V_n = V_n + 1$.
  - If $e \in E_r$ and the previous time an edge $\in P(n)$ was found on the polygon, this edge was $\in E_l$ then $V_n = V_n + 1$.

Now for each net $n$ that crosses the polygon, we calculate $V_n$. Note that each net is placed in a layer, thus for each layer $k$ the flow over cut $(p_b,p_e)$ can be calculated with:

$$flow((p_b,p_e),k) = \sum_{n \in \text{layer } k} V_n$$

This means the flow over cut $(p_b,p_e)$ in a certain layer is equal to the sum of all useful net crossings of the polygon in that layer.

### 6.2 The algorithm for checking routability

We now present the algorithm for checking the complete sketch. This algorithm mainly consists of calling the cut-check algorithm many times. The cut-check algorithm is also given. The latter algorithm will be extensively explained.

#### 6.2.1 The algorithm to check routability

**Algorithm Check System**

\[
\forall \text{layer } l
\forall \text{point } p_1 \in P
\forall \text{point } p_2 \in P, \ p_1 \neq p_2
\text{Check Cut } (p_1, p_2, l)
\]

**Algorithm Check Cut** $(b,p_e,l)$

if $((p_b,p_e) \in E)$ check if $flow ((p_b,p_e),l)$ does not exceed $cap((b,p_e),l)$

else {
  - Find the left part $E_l$ and the right part $E_r$ of the polygon around $(p_b,p_e)$.
  - $\forall \text{net } n \in \text{layer } l$ calculate $V_n$
  - $flow((p_b,p_e),l) = \sum_{n \in \text{layer } l} V_n$
  - Now cut $(p_b,p_e)$ is safe if $cap ((p_b,p_e),l) \geq flow((p_b,p_e),l)$
}
6.2.2 Finding the polygon in the check cut algorithm

Given two points $p_b, p_e \in P$, we want to find the polygon for cut $(p_b, p_e)$.

Let $p, p_1 \in P$, $e \in E$. Let $E_{ccw}(e, p)$ denote the edge $e_1 = (p, p_1)$ such that $(p, p_1) \in E$ and $(p, p_1)$ is the first edge encountered by walking around $p$ in a counter clockwise way, as indicated in figure 6.4.

![Figure 6.4: Counter clockwise edge ordering.](image)

The polygon can be found with the following algorithm:

Let $p = p_b$
Let $e = E_{ccw}(e_1, p)$ such that $OP(e_1, p)$ is left of cut $(p_b, p_e)$ and $OP(e, p)$ is right of $(p_b, p_e)$.
$E_l = \{e_1\}$; $E_r = \{e\}$
p = $OP(e, p)$$e = E_{ccw}(e, p)$$\textbf{while } (OP(e, p) \neq p_e) \{$$\textbf{if } (OP(e, p) \text{ is left of } (p_b, p_e)) \textbf{ then } E_l = E_l \cup E_{ccw}(e, OP(e, p))$$\textbf{ else } \{$$
\quad E_r = E_r \cup \{e\}$$\quad p = OP(e, p)$$\quad e = E_{ccw}(e, p)$$\}$
e = E_{ccw}(e, p)$$E_l = E_l \cup E_{ccw}(e, OP(e, p))$$E_r = E_r \cup \{e\}$$\}$

An example is given in figure 6.5.
The hull around cut $(p_b, p_e)$ has to be found. Initially, $E_l = \{e_a\}$ and $E_r = \{e_h\}$. First $e = \text{edge } e_a$. $OP(e_a, p_a)$ is left of $(p_b, p_e)$, so edge $e_h$ is added to $E_l$. Then $e$ becomes edge $e_p$. $OP(e_p, p_a)$ is left of the cut so $e_i$ is added to $E_l$. Now $e$ becomes $e_q$, and we must add edge $e_j$ to $E_l$. Then $e$ becomes edge $e_c$, and $OP(e_c, p_a)$ is right of cut $(p_b, p_e)$, so we add $e_c$ to $E_r$...

Finally $E_l = \{e_a, e_c, e_d, e_f, e_g\}$ and $E_r = \{e_h, e_i, e_j, e_k, e_l, e_m, e_n\}$. 

6.3 Complexity of the algorithm

The worst case complexity of checking one cut, is the worst case complexity of creating the polygon and checking each edge and point on it. Checking the edges that form the polygon takes

\[ O(N_N) \], with \( N_N \) = the number of nets in the routing area.

This makes the worst case complexity of checking a layer:

\[ O(N_N \times N_P(N_P - 1)/2) \], with \( N_P \) the number of points

because we must check each possible cut.

Therefore the worst case complexity of checking all layers:

\[ O(k \times N_N \times N_P^2) \] with \( k \) the number of layers
In this chapter the remaining nets will be placed using vias, if this is possible.

7.1 Properties of vias

In chapter 2 a via was defined to be a connection between two adjacent layers. Throughout this chapter we use the following extended definition:

Definition 7.1 A Via is a point in the routing area with the following properties:

- The via connects two netsegments that are placed on different adjacent layers.
- A via must be placed inside a triangle, therefore introducing 3 new edges; from each cornerpoint of the triangle to the via.
- A via has a width, so it influences routability.

A vianet is a net that intersects with at least one net on each layer. Such a net should be divided over 2 adjacent layers; each time it changes from one layer to the other layer a via has to be introduced. These two layers are defined by:

\{layer_1, layer_2\} with layer_2 = layer_1 + 1.

7.2 Problem definition

Let's first define the problem discussed here: Given two layers \{layer_1,layer_2\}, all nets that are placed in these two layers and a vianet that should be placed in these two layers. For the vianet the netroute is already known. Find the minimum number of vias needed and the exact geometrical positions of these vias. Then divide the old vianet into netsegments and determine the layer these segments should be placed in.
Note we have a difficult problem here; we have to find exact geometrical positions out of topological information. Luckily we are not completely without geometrical information: we know the exact positions of all points. With this information we have the geometrical information of each triangle. It is obvious that we will try to place the via inside an existing triangle in the routing area. We will try to avoid placing a via on an edge, because an edge is the smallest distance between two points. Thus the worst location to place the via is exactly between these points (on the edge).

7.3 The algorithm (overview)

The problem is solved by taking the next steps:

1. Create a hull of edges around the vianet.

2. Order the nets on the hull of the polygon in such a way that the minimal number of layer changes on this hull is achieved.

3. Determine the minimal number of vias and their topological positions.

4. For each topological via, determine all viacandidates (triangles the via can be placed in).

5. For each via we place the via in the first candidate it can be placed in. If none of the candidates of a via will do, we have to try to place the via on two other adjacent layers.

7.4 Creating a polygon around the vianet

Given net \( n \), with path \( P(n) = (e_1, e_2, ..., e_{|P(n)|}) \), begin point \( p_b \) and end point \( p_e \). As in the routability check we create a left \( E_L \in E \) and a right set of edges \( E_R \in E \) that form the polygon around \( n \). The edges of \( E_L \) and \( E_R \) are called hulledges. The edges inside the polygon are called internal edges. The hulledges are defined by: \( T(n) \setminus P(n) \).

We can construct the left and right set of edges in the following way:

The first edges of both sets are found by finding the edges \( (p_b, \text{begin}(e_1)) \) and \( (p_b, \text{end}(e_1)) \). Then we decide which of these 2 edges is the first edge of the left set and which edge is the first edge of the right set. Now for each pair of edges \( e_i, e_{i+1} \in P(n) \) we search a third edge in the same way as in chapter 4. Each of these third edges we add to the left or right set, by checking to which set (left or right) it is connected.

In figure 7.1, in the first step we find edge \( e_A \) which is left of the vianet and we find net \( e_G \) which is right of the vianet, so

left set: \( (e_A) \); right set: \( (e_G) \)
In the next step we find edge $e_B$ between edge $e_1$ and $e_2$. This edge is connected to edge $e_A$ so $e_B$ should be added to the left set.

In the last step we find edges $e_F$ and $e_K$. Edge $e_F$ is connected to the last edge in the left set and $e_G$ is connected in the right set. So we get:


### 7.5 Ordering the hull edges

The basic idea is explained in the following figure:

If we have the situation 7.2a, we should need 3 vias (squares) for the vianet. But if we swap nets 2 and 3, we get situation 7.2b, which results in only one via.
There are three types of hull edges:

1. Edges that contain no nets in layer 1 or layer 2.
2. Edges that contain only nets in one layer, layer 1 or layer 2.
3. Edges that contain nets of both layer 1 and layer 2.

The ordering of the hull comes down to swapping all nets of layer 1 and layer 2 on certain edges of the polygon in order to minimize the number of layer changes on the polygon. The only edges we can swap nets on are edges of type 3.

Not all nets of layer \{layer 1, layer 2\} that cross the edges of the polygon or are connected to the polygon are to be taken into consideration in the ordering. A number of nets can cross only one side of the polygon or they can cross one side of the polygon a few times before they cross the other side. This is indicated in figure 7.3.

![Figure 7.3: Useful net crossings.](image)

We see that the vianet only crosses the net segment between edge \(e_x\) and edge \(e_y\). Thus we must again find all useful net crossings for each net. This can be done in the same way as described in chapter 6.

We can now perform the ordering step with all useful crossings on the polygon.

To order the complete polygon we can order both sides individually. This is because both sides are connected only at the begin and endpoint of the vianet and have therefore no other points or edges in common. Thus: we have two independent problems.

**Definition 7.2** A break point is a point \(p \in P\) such that there is an edge \((p, p') \in E_l \cup E_r\) and there is a useful net segment attached to \(p\). A break edge is an edge \(e \in E_l \cup E_r\) of type 2. Break points and break edges are called breaks.

**Definition 7.3** Let \(p \in P\) and let \(l \in \{\text{layer 1, layer 2}\}\). Now we can assign \(l\) to \(p\). We say that point \(p\) has color \(l\), or \(C(p) = l\). The other color \(O C(l) = \)

- layer 1 if \(C(p) = \text{layer 2}\)
- layer 2 if \(C(p) = \text{layer 1}\)

In case a point \(p \in P\) has a connected edge \(e \in E_l \cup E_r\) and edge \(e\) is of type 3, the nets on edge \(e\) can be ordered in such way that the nets of layer \(C(p)\) are the closest to \(p\) and the ordering within each of the layers is maintained. This is called contraction of nets to a point. An example is given in the following figure:
To order one side of the polygon we use the following algorithm:

Algorithm Order One side
Let begin(vianet) be $p_1$ and end(vianet) be $p_{p+1}$.
Let $E_j \in \{E_l, E_r\}$ be $(e_1, e_2, ..., e_p)$ and $e_i$ be $(p_i, p_{i+1})$, so $E_j$ can be denoted as $(p_1, p_2, ..., p_{p+1})$
Walk along $E_j$, searching for a break.
if (there is a break) {
  if (breakpoint $p_i \in P$) then
    $C(p_i) = \text{layer(net attached to } p_i)$
  else (* break edge $e_i \in E_j$ *)
    $C(p_i) = \text{layer(nets on edge } e_i)$
    Trace($p_i$, backward)
    Trace($p_i$, forward)
} else {
  $C(p_{p+1}) = \text{layer1 (* just pick one *)}$
  Trace ($p_{p+1}$, backward) (* trace backward all edges *)
}

Algorithm Trace(point $p_i$, direction)
If direction is backward, then for each $j$, $1 < j < i$ let $e_j$ be $(p_{j-1}, p_j)$, and denote $p_{j-1}$ as $p_{\text{next}}$. If direction is forward, then for each $i < j < p + 1$ let $e_j$ be $(p_j, p_{j+1})$, and denote $p_{j+1}$ as $p_{\text{next}}$. Now for each of the edges $e_j$ we do the following:

- if $(e_j \text{ of type 1})$ then $C(p_{\text{next}}) = C(p_j)$
- if $(e_j \text{ of type 2})$ then $C(p_{\text{next}}) = C(\text{nets on edge } e_j)$
- if $(e_j \text{ of type 3})$ then $C(p_{\text{next}}) = OC(C(p_j))$
- Contract all nets on edge $e$ in layer $C(p_{\text{next}})$ to $p_{\text{next}}$

To make things clearer we look at the algorithm applied on figure 7.5. In this figure the layer changes are indicated with numbers. We can see that have 6 layer changes in this example.

The result is given in the figure 7.6. After ordering the edges, we find only 3 layer changes on that side of the polygon.
7.6 Determination the topological viapositions

At this point we have two ordered sets of edges and we know for each edge which of the net crossings on the polygon are of importance for placing the vianet (useful). Also the nets on the hulledges are ordered in such a way we have minimum number of layer changes on both parts of the polygon. For both parts of the polygon the useful net crossings are stored in a netlist, resulting in a left netlist and a right netlist. The nets in these lists are ordered in the same way as their appearance on the polygon.

To get a clearer view of how to find the minimum amount of topological viapositions, we represent the useful net crossings on the polygon in the way indicated in figure 7.7b.

Figure 7.7: Example before layerswapping.

Figure 7.6: Example after layer swapping.

Figure 7.7: Example of constructing a net-diagram.
From this diagram we can easily determine the topological viapositions. In figure 7.7, if we want to place the viannet, we have to route a wire from X to Y, using a minimum amount of vias.

**Definition 7.4** The *topological* viaposition of a via \((\text{lnet}, \text{rnet}, \text{side})\) is defined by:

- The two nets the via must be placed between, called \(\text{lnet}\) and \(\text{rnet}\). \(\text{lnet}\) is the net that lies *before* the viaposition on the polygon (relative to the direction of the viannet), \(\text{rnet}\) lies *after* the viaposition.
- The *side* of the polygon the via must be placed at: left list or the right list.

The algorithm to find the minimum amount of topological viapositions is given below.

1. Let the *left netlist* be denoted by \((n_l, \text{rest}_l)\), such that \(\forall n \in n_l\) layer\((n)\) is the same and layer(first element of \(\text{rest}_l\)) \(\neq\) layer(elements of \(n_l\)). Let the *right netlist* be denoted by \((n_r, \text{rest}_r)\), such that \(\forall n \in n_r\) layer\((n)\) is the same and layer(first element of \(\text{rest}_r\)) \(\neq\) layer(elements of \(n_r\)).

2. *if* \(|n_l| > |n_r|* *then* \(n_{lr} = n_l\)* *else* \(n_{lr} = n_r\)

3. Let \(l\) be layer(nets in \(n_{lr}\)).

4. Remove \(n_{lr}\) from its corresponding netlist

5. Now we found a topological viaposition (see definition 7.4):
   - \(\text{lnet}\) is equal to the last net deleted from layer \(l\) from list \(n_{lr}\)
   - \(\text{rnet}\) is equal to the first net in the remainder of list \(n_{lr}\)
   - \(\text{side}\) \(n_{lr}\), which is *left* or *right*.

6. Delete all nets in \(n_{lr}\) from the other netlist too.

7. Go to 1.

If initially nets of both layers can be chosen, the algorithm should be done for both possibilities and the solution with the least amount of vias should be used. In figure 7.7 two possible solutions can be found, which are given in the figure 7.8.

In figure 7.8a, 3 topological viapositions are found, indicated with the arrows. These positions are: a: \((2, 1, \text{right side})\), b: \((5, 3, \text{right side})\) and c: \((6, 7, \text{left side})\). In figure 7.8b, the topological viapositions are: a: \((1, 2, \text{left side})\), b: \((3, 4, \text{left side})\) and c: \((7, 6, \text{right side})\).
7.7 Searching candidates for via placement

We now have a minimum number of topological via positions, needed to place the vianet in the two layers \{layer1, layer2\}. For each of the topological via positions we must find the exact geometrical position. The first thing we can conclude from our topological via positions is that they are positioned on the polygon. But we demanded that we did not want any via on an edge. So the vias must be placed inside the polygon. This means that we must place a via in a triangle. But the polygon consists of a number of triangles. For each via we can therefore find a number of via candidates, which are triangles where the corresponding via might be placed.

**Definition 7.5** A via candidate is a triangle in the triangulation in which a via might be placed.

Given a topological via position \( v \) the set of candidates for \( v \) is denoted by \( C_v \).

It is obvious how we should find \( C_v \) for each \( v \): given a topological via position \( (\text{lnet}, \text{rnet}, \text{side}) \), the via might be placed in each of the next triangles:

1. The triangles \( \text{lnet} \) runs through.
2. The triangles \( \text{rnet} \) runs through.
3. All triangles in between the triangles found in step 1 and 2.

Because it suffices to place one via in a triangle, we will never have to place more than one via in a viacandidate. This gives us an upper bound of the number of vias the vianet maximally has to be placed with:

\[
\text{Max. } |\text{Vias per vianet}| = |\text{T(vianet)}|
\]

This obviously is an extreme bound.

An example is given in figure 7.9.

With our algorithm we find triangle I as candidate for via 1, and for via 2 we find: II, III and IV as candidates.
7.8 Via placement

Given a topological via position $v$ and the set of candidates $C_v$. For each set $C_v$ we must now find one candidate $c_v$ in which the via $v$ geometrically can be placed. If there is a set $C_v$ for which such a candidate cannot be found the vianet cannot be placed on the layers {layer1, layer2}.

Suppose we want to place via $v$ in a candidate $c \in C_v$. This means a new point and 3 new edges are introduced in the triangle, partitioning $c$ into 3 new triangles. The exact position of the via however is not known yet.

We try to determine this position as follows:

1. Choose a $c \in C_v$

2. Introduce a virtual via $v_v$, which is a via from which the exact position is not known.

3. Introduce 3 new edges to the routing area, one from each corner point of $c$ towards $v_v$. Let the edges be denoted by $e_{i1}$, $e_{i2}$ and $e_{i3}$.

4. Decide which nets that cross $c$ (that are in layer {layer1, layer2}) have to cross which of the edges $e_{i1}$, $e_{i2}$ and $e_{i3}$.

5. Now for each of the edges $e_{i1}$, $e_{i2}$ and $e_{i3}$, the flow is known for layer1 and layer2, we can try to select an exact position for the via. If no such position can be found, because of technology-constraints, we must choose another candidate from $C_v$ and goto 2.

6. If we found a legal position in the previous step, we must still place all nets not in {layer1, layer2} on the 3 new edges. If this is not possible due to technology constraints, we must try another candidate from $C_v$, else (if it is possible) we try to place the next via $v$ and goto 1.
7.8.1 Placing the nets over the 3 new edges.

We now determine for all nets from layer1 or layer2 that cross triangle \( c \), which of the 3 new edges they must cross in such way that planarity for both layers is preserved. What we do is the following: we walk around triangle \( c \) one time to decide where each net is placed relatively to the vianet (left/right, see chapter 4). Then we are able to decide for each net on the triangle on which internal edge \( e_{i1}, e_{i2} \) or \( e_{i3} \) it has to be placed. After this we can actually place the nets of \{layer1, layer2\} on the 3 new edges. Note that planarity of both layers must be maintained. This can easily be done because the nets on the edges of \( c \) are already ordered such that triangle \( c \) is planar. An example is given in in the following figure:

![Figure 7.10: Placing nets over new edges](image)

In figure 7.10a, a triangle \( c \) with crossing nets is given, in which via \( v \) should be placed. The vianet is indicated by a fat line with the arrow. In figure 7.10b, the via is placed and the 3 new edges are drawn. In figure 7.10c the vianet is placed and all nets are placed over \( e_{i1}, e_{i2} \) and \( e_{i3} \) such that planarity is preserved for both layers (dashed and solid lines).

7.8.2 Determining the exact viaposition

Given a triangle \( c \), defined by 3 edges \( e_1, e_2 \) and \( e_3 \). We want to find the exact geometrical coordinates of a via \( v \) inside \( c \) if there exists such a point.

Note: The order of the nets on the new edges edges is not important in this problem, only the maximum number of nets that cross each of the 3 new edges in layer1 and layer2.

The first thing we have to do is to find the proper metric for this problem. For each of the 3 new edges, we know the number of nets that cross it. Because the capacity of an edge is defined by:

\[
\text{cap}(e) = \text{MAX}(\text{dx}, \text{dy}), \text{ with } \text{dx} = |X(\text{begin(edge)}) - X(\text{end(edge)})| \\
\text{and } \text{dy} = |Y(\text{begin(edge)}) - Y(\text{end(edge)})|
\]

we can create minimal squares around each of the cornerpoints of the triangle. This is indicated in figure 7.11.
Na, Nb and Nc are the amount of space needed for the nets that have to cross the internal edges $Av(e_{1})$, $Bv(e_{2})$ and $Cv(e_{3})$.

In this figure we can see that if we are able to find a position outside the three squares and inside the triangle, we can place all nets of layer1 and layer2 on the internal edges, and we can find an exact via position. So we must find a point in the dashed area.

We do the following:

First, we decide which of the squares cross each other, which is easy, because of each square we know its middle-point (A, B or C) and the length of half the rib (Na, Nb or Nc respectively). Now we have the next situations:

1. None of the squares overlap: find a point inside the triangle on one of the squares (fig 7.12a).

2. There is one square that does not overlap the two others, which do overlap each other: Now find a point on the square that does not overlap the other two squares; this point must of course be inside the triangle (fig. 7.12b).

3. There is one square that overlaps the two other squares which do not overlap each other (fig. 7.12c). Now we cannot simply find a point on a square. We now calculate the cross-points of the two non-overlapping squares with the edge that connects the middle-points of these squares (in fig. 7.12c these points are ta and tb). Now we pretend to have a new triangle, (C, ta, tb) and we find a point on square C that is inside triangle (C, ta, tb). If such a point cannot be found, the via cannot be placed.

4. All squares overlap: obviously, we cannot place the via in this situation, because there cannot be found an area that lies outside each square.
But how do we find a point on a square? Look at figure 7.13 first:

In figure 7.13a, a corner of the square lies inside the triangle. This point can easily be found. In figure 7.13b no cornerpoint of the square lies inside the triangle. What we do is the following: We determine the middle of edge BC (point X) and draw a line from A to X. We calculate the point where line AX crosses the square and this point is our candidate.

If we have a grid of points, eg. all coordinates must be integer, we might round our viaposition in such a way, we find a place outside the triangle. In this case we walk along line AX and take the first gridpoint that is

- Inside the triangle.
- Outside each square.

If we cannot find such a point, the via cannot be placed in the triangle.
7.8.3 Placing the other nets in the triangle

Suppose we found a correct position in the previous step. Now only the nets from \{ layer1, layer2 \} are placed on the 3 new edges yet. Because we decided to model the via as a point the via appears on each layer. This means we also have to place all nets not in \{ layer1, layer2 \} on the 3 new edges, and we must maintain routability and planarity of each layer. The viaposition is known, so the capacity of each of the 3 new edges is known. We must therefore try to place all remaining nets on edges they fit on. If we cannot place all these nets (because the flow of an edge exceeds the capacity of that edge on that layer) the via cannot be placed in the triangle, although we found a legal viaposition.

Here we see a disadvantage of modeling the via with a point: the routability check was succesfull before the viaplacement, and now it can fail on a layer, the vianet is not placed in.

The steps that have to be done are given below.

- We decide for each net \( n \not\in \{ \text{layer1, layer2} \} \) which of the 3 edges of \( c \) it crosses.
- We travel the edges of \( c \) again. Each net \( n \not\in \{ \text{layer1, layer2} \} \) we place on the edge that leads to the shortest path, such that for each layer the planarity is maintained. If there is not enough capacity left on that edge, we place the net over the two other edges. If this is not possible either, we stop the routine, because we have a routability failure. For each of the 3 edges we keep track of how much capacity is left on each layer.

This is indicated in figure 7.14.

![Figure 7.14: Placement of other nets](image)

We see that net a crosses edges \( e_1 \) and \( e_3 \), so we try to place it on \( e_{i2} \) first, because this is the shortest path. If this is not possible (maybe the capacity of \( e_{i2} \) is not big enough on that particular layer), we try to place it over edges \( e_{i1} \) and \( e_{i3} \).
7.8.4 Routability after placing a via

After a correct point has been found for the via, still all cuts towards this point have to be checked for safeness, to guarantee routability for the complete routing area. This is done in the way indicated in chapter 6. Note that now only \( O(N_P) \) cuts have to be checked, with \( N_P \) equal to the number of points in the routing area. If the routability check fails on a cut, the via must be placed in any of the other candidates for the via. If no candidates are left, the vianet cannot be placed on the layers \( \{\text{layer1, layer2}\} \).

7.9 Complexity

Although the algorithm is quite complex, we will try to make a worst case estimation of the complexity of placing one vianet here.

Let \( N_N \) be the number of nets in the routing area.

Let \( N_P \) be the number of points in the routing area.

Let \( N_E \) be the number of edges in the routing area.

Let \( E_n \) be \( |\text{P(vianet)}| \).

1. Finding the polygon: This complexity is worst case \( O(E_n) \), because the path of the vianet must be travelled once.

2. The ordering of the hull edges: Seeking of the useful netcrossings: here we must trace the complete path of the involved nets, so the worst case complexity will be \( O(N_N \cdot N_E) \). The ordering of the hull edges has the same complexity, because each net can appear on each edge of the hull.

3. Determination of the minimum number of vias with topological viapositions: For this we must travel both netlists once, worst case costing \( O(N_N \cdot E_n) \), because worst case each net crosses each edge of the polygon.

4. Seeking all viacandidates for all vias: Here, we travel along each of the triangles of the polygon once and mark it if it is a candidate for a via, thus costing \( O(\text{number of triangles in polygon}) = O(E_n) \).

5a. Exactly placing a via in ONE triangle We must place the vianet in the triangle such that planarity is maintained, create 3 new edges and place all nets crossing the triangle over these edges such that planarity is preserved, and finally the exact viaposition must be checked. This worst case costs \( O(N_N) \).

Checking all cuts towards the via for safeness: this means checking \( N_P \) cuts, resulting in a complexity of \( O(N_P \cdot N_N) \).

5b. Exactly placing all vias of one vianet: This has to be done maximally for each triangle in the polygon: do step 5a \( E \)-times; thus resulting in a worst case complexity of:

\[
O(N_E \cdot N_N \cdot N_P)
\]

Total Complexity: \( O(E_n) + O(N_N \cdot N_E) + O(N_N \cdot E_n) + O(E_n) + O(N_N) + O(N_E \cdot N_N \cdot N_P) \)

which is worst case equivalent to:

\[
O(N_E \cdot N_N \cdot N_P)
\]
7.10 Placing all vianets

In the complete previous section, for two adjacent layers \{ layer1, layer2 \} a vianet was tried to be placed. But what happens if this net cannot be placed on these two layers? This occurs if one via cannot be placed in any of its viacandidates. This means the number of nets in all of the via's candidates is too big to be able to fit the via in these candidates. It is obvious that we must try two other layers then. But what should be optimized? We could optimize the \textit{the total number of vias}. This is obviously neccessary. However, to calculate that we should store each possibility of each net on each layer and of course this is too expensive.

Therefore we just place the vianet on the first two layers it can be placed on, trying from the toplayer downward to the last layer. We will not have to store any possible solution then, and we still get a suboptimal solution, because the algorithm of the previous section still minimizes the number of vias for the two layers it tries to fit the vianet in. Furthermore, the netroute of the vianet is the same in each layer.

If any vianet does not fit on any of the layers, this problem is not routable.

After this, the sketch is complete.

7.11 Complexity of placing all vianets.

The worst case complexity of placing one vianet has been described in section 7.9 already. However it might be possible that a net cannot be placed on the two layers, so we have to retry placement on two other layers all over again.

Let \( k \) be the number of layers.

Let \( N_v \) be the number of vianets.

Worst case we have to try to place each vianet on each \((k - 1)\) pair of layers. So for placing \textit{one} vianet we worst case need:

\[(k - 1) * O(N_N * N_E * N_P)\]

If we have much more nets than layers, which is the case in normal situations, \( N_v \) can worst case be almost as big as \( N_N \). This means the worst case complexity of placing all vianets (which is equal to placing all vias) will be:

\[(k - 1) * O(N_N^2 * N_E * N_P}\]

and because \( N_E \) is linear to \( N_P \) (lemma 2.1)

\[(k - 1) * O((N_N * N_P)^2)\].

Note that in the current technology \( k \) can be as big as 3 to 5 layers, so this factor might be seen as a constant.
Chapter 8

Conclusions

8.1 Implementation

At this point, a backbone-implementation has been designed. That is, a program is implemented using the algorithms described in this thesis. Most important demands were that if the router generates a sketch, this sketch must be correct, that is: it must be possible to geometrically route the sketch. The routability check that was designed should assure this for all the sketches generated by our topological router. Secondly, we would like to be able to generate a correct sketch in as many cases as possible.

8.1.1 Determination of the net routes

An algorithm is proposed that routes each net in time linear to the number of points in the sketch. For each net a number of net routes might be introduced. This feature is not implemented yet, however the data structure is already suitable for this. The algorithm tries to minimize the number of edges in the path for each net, to minimize overall complexity. This is not the same as minimizing the total wire length. The user could implement his own optimizing function if required.

8.1.2 Calculation of the net crossings

An algorithm is found that determines the net crossings in time quadratic to the number of nets. The algorithm planarizes the whole routing area at the same time. In the current implementation of the router these 2 steps are still separated as 2 independent procedures. The memory usage is quadratic in the number of nets. This is because the net crossing information is stored in a two dimensional array. If lists are used, this memory usage could be made much smaller, especially for large sketches where this two dimensional array will be a sparse matrix.
8.1.3 \textit{k-MIS}

For this NP-complete problem a linear time heuristic is proposed, which is greedy in placing as many nets as possible in each layer. Using a so-called \textit{bucket}, allows us to perform all actions in constant time, except for the recalculation of the mindegree. The overall complexity is linear to the number of edges in the crossing graph. The \textit{k-MIS} heuristic does not optimize the spreading of the nets over the layers. Multiple netroutes can easily be handled, by adding an edge for each pair of netrealizations of the same net in the crossing graph.

The memory-usage of the bucket is linear in the number of nets, each net appears only once in the bucket.

8.1.4 Routability check

Checking the complete routing area for routability is still an expensive step, its complexity is always quadratic in the number of points, times the number of nets. However, this complete check is done once after the \textit{k-MIS} calculation. After placing a new via, all cuts towards the new point must be checked too, but this check has complexity of the number of nets times the number of points. A significant improvement of the complexity of these checks has not been found yet.

The memory-usage here is linear to the number of points. For each net the number of times it crosses a side of the polygon is stored.

8.1.5 Viaplacement

The placing of the vianets is \textit{worst case} quadratic in the number of nets times the number of points. This is an extreme bound. Given a topology, the minimum number of vias is found for a net. The algorithm is greedy, it tries to place the vianet in the first two adjacent layers the net fits in. Finding the optimal two layers to place the vianet in turns out to be too expensive. Also the extension might be made to place the vianet using all possible layers instead of only two layers in such a way the minimum amount of vias are found. This is not implemented in order to keep complexity low and still get a satisfying result.

The memory-usage is not extremely high in this part. First all edges of the hull of the polygon are stored in lists. This amount is certainly smaller than the total number of edges (except for the trivial examples). Secondly all useful net crossings on the hull are stored in a list. Worst case this amount equals the number of nets times the number of edges on the hull. However such a situation will not appear, because the router minimizes the amount of edges on the path of each net.
8.2 Future work

For each of the steps in the complete topological routing algorithm the future work is already discussed in the previous sections. Some more things could be done:

- In this thesis the assumption is made that the routing area is the same for each layer. It might be possible to extend this and make each layer unique. This should result in router that can handle a much broader set of examples. However, in this case a topology of a net would not be the same for each layer anymore, so the determination of the crossing graph, determination of a $k$-$MIS$ and the viaplacement routines should be changed.

- The rerouting of nets after the $k$-$MIS$ is not implemented yet. The router, as described in chapter 3, cannot place nets using information of previously placed nets to avoid net-crossings. This is a neccesary feature to guarantee planarity of each layer after rerouting. Note that after (or during) the rerouting-phase, the routability of each layer must be checked.

- An extension has to be made to handle multiterminal nets.

- Although the router is already prepared to handle features, features cannot be introduced yet. The program should be extended to read the features from file.

8.3 Conclusions to the complete router

The $k$-layer topological routing problem is obviously a NP-complete problem. The problem is split in a number of smaller problems, from which a number itself are NP-complete. We always must weigh the complexity of the routines and memory-usage against the optimality of the solution. The algorithm implemented so far is a backbone algorithm in which a lot can be altered! The total complexity at this point is worst case:

$$O(N_N \times N_P \times C + N_E \times N_N^2 + k \times (N_N + N_E) + k \times N_N \times N_P^2 + k \times (N_N \times N_P)^2)$$

which is worst case:

$$O(k \times (N_N \times N_P)^2)$$

with $N_N$ the number of nets, $N_P$ the number of points, $C$ the maximal cost in the routing graph, $k$ the number of layers, $N_E$ the number of edges in the crossing graph.

In the current router all basic steps are only executed once. If after the $k$-$MIS$ a routability failure would happen, the program simply stops because at this point there is no other netroute available to try.
The final conclusion is that a polynomial-time topological router has been designed and implemented that can handle a variable amount of layers and obstacles within the routing area. Given a topology for each net, the number of vias is minimized on a 2-layer basis. The current algorithms represent a backbone system which can and should be extended to be able to handle a bigger class of sketches. The datastructures used in this router are compatible with the datastructures F. Ruys used in his work, so coupling the programs is possible without transferring data via files or interface-programs.
Appendix A

Implementation.

A.1 About the program and the system

The complete router is designed in ANSI C on a HP-9000:750 machine using UNIX as operating system and X Windows. It uses a program called Backdraw to plot the graphs read and generated. However, this drawing-feature can easily be left out or changed, because it is just an application to plot the generated data and not an essential part of the program.

The program has option -g to show the next steps:

- Show the data from the input file, immediately after reading it.
- Show the crossing graph
- Show the result after the maximal independent set calculation and layer assignment of all nets that can be placed immediately.
- Show the result after placing all viannets. This figure is the complete generated sketch.

For more information about the steps taken by the program, the option -v can be used. This is the so called verbose mode.

Without the view-option used, the program should be portable to any system.
A.2 The inputfile

The inputfile for the program should be as follows:

Number of layers

Number of points

X(point 0) Y(point 0)
X(point 1) Y(point 1)
...

X(NOP-1) Y(NOP-1)

Number of nets

X(beginpoint net 0) Y(beginpoint net 0) X(endpoint net 0) Y(endpoint net 0)
X(beginpoint net 1) Y(beginpoint net 1) X(endpoint net 1) Y(endpoint net 1)
...
X(beginp net NON-1) Y(beginp net NON-1) X(endp net NON-1) Y(endp net NON-1)

Example

consider the next inputfile:

3
5
0 0
100 0
0 100
100 100
50 50
2
0 0 100 100
0 100 100 0

This example has 3 layers and 5 points. Point 0 has coordinate (0,0), point 1 has coordinate (100,0) and so on.
There are two nets, one from (0,0) towards (100,100) and the other from (0,100) towards (100,0). The direction of the net is not of importance on the outside of the program. On the inside, the order of begin and endpoint is used. This however, is of no concern to the user. The resulting graph could look like this:
Appendix A Implementation.

We see that each point is drawn and the number of the point is drawn too. The edges are calculated by the program itself (CDT routines by F. Ruys; [Ruys92]). The numbers of the edges are drawn too. The program routes the nets 0 and 1 and places them in the layers (thick black line and thick dashed line). This example is trivial, because there are only two nets that can be placed on three layers.

Note that if we translate each terminal and cornerstone of a module by a point, we can easily make an inputfile for our routing area, by passing the coordinates of these points and passing the coordinates we wish to be connected by nets. If a multiterminal net is needed, we must divide it in two-terminal segments ourselves.

A.3 The outputfile

If the router can create a legal sketch, an outputfile is generated from the data in the datastructures. The name of the outputfile is “inputfilename”.sketch. All message-lines start with a #-sign.

A.4 The datastructures

An overview of the important datastructures is given to indicate what typical information is necessary to make topological and geometrical routing possible.

A.4.1 Datastructure for points

```
struct point {
    int    : id;    /* point-identifier */
    int    : x;     /* x-coordinate of the point */
    int    : y;     /* y-coordinate of the point */
    int    : type;  /* terminal or via */
    EDGE.CIRCLE : *edges; /* list of edges connected to point */
    CROSS.NET : *net;  /* list of nets connected to point */
    some more flags...  
};
```

Figure A.1: Example of inputfile

We see that each point is drawn and the number of the point is drawn too. The edges are calculated by the program itself (CDT routines by F. Ruys; [Ruys92]). The numbers of the edges are drawn too. The program routes the nets 0 and 1 and places them in the layers (thick black line and thick dashed line). This example is trivial, because there are only two nets that can be placed on three layers.

Note that if we translate each terminal and cornerstone of a module by a point, we can easily make an inputfile for our routing area, by passing the coordinates of these points and passing the coordinates we wish to be connected by nets. If a multiterminal net is needed, we must divide it in two-terminal segments ourselves.
Topological Routing

We do not actually need an identifier for a point, but this field makes it easier to identify a point anywhere in the program. The x and y positions are integers, mainly to keep all things simple to calculate and to spair memory. This means we have a sort of a grid. But it is a changeable grid, because if we want to have a twice as high resolution, we simply multiply all coordinates by 2. The type field can be used to identify if a point is a terminal or a via. There is a small difference between these points, because a via might (and will) have a different geometrical sizes than a terminal. Edges is a double linked circular list of edges that are connected to the point. We can walk along these edges clockwise or counter clockwise. The net-field is a linked list of nets that are connected to the point. There is no ordering in this list. It is left to the user to add extra flags if needed.

A.4.2 Datastructure for edges

```
struct edge {
  int : id;        /* edge-identifier */
  int : nocn;      /* number of crossing nets*/
  CROSS_NET : *list; /* list of nets that cross the edge */
  point : *begin;  /* begin point */
  point : *end;    /* end point */
  edge : *backtrace; /* routing purposes */
  some more flags...
};
```

The identifier is used again to be able to immediately identify the edge anywhere in the program. The number of crossing nets is useful to see how much nets cross an edge. List is a pointer to a linked list of nets that cross the edge. The edges are defined from the begin point towards the end point, so the nets are ordered in that way too. The first net in the list is closest to the begin point of the edge. This ordering is needed to guarantee planarity. Begin- and end point are the begin and end point of the edge, defining the direction of the edge. Backtrace is a pointer to an edge, needed to route the nets over the edges. With this pointer we can keep track of the previous edge a net was placed on. It is left to the user to add extra flags if needed.

A.4.3 Data structure nets

```
struct net {
  int : id;          /* net-identifier */
  int : noce;        /* number of crossing edges */
  int : layer;       /* layer number the net is placed in */
  point : *begin;    /* begin point */
  point : *end;      /* end point */
  REAL : *real;      /* list of netrealizations */
  some more flags...
};
```
The identifier is again a way to identify the net anywhere.
The number of crossing edges is useful to decide immediately how much edges the net
crosses. For a number of routines this value can help to keep complexity low.
The layer-field is a simple identifier, which is equal to the layer the net is assigned to.
It is best to initially assign a special value, say \texttt{NIL} (= -1) to this field. When a net has
to be assigned to a layer, we only have to set the layer-field of the net.
Begin- and end point are the points (terminals and/or vias) the wire has to connect.
The \texttt{realization}-field is a linked list of possible netroutines. This feature might be used in
the router to choose from a number of netroutines and take the one that gives the most
satisfying solution. Each netroute itself is again a linked list of edges, the net crosses.
After the best realization is chosen, this realization should be the first element in the
realization-list, to avoid unneccessary complexity.

To be complete, the structure of the \texttt{realization} looks like:

\begin{verbatim}
REAL {
    REAL      : *next;
    CROSS_EDGE : *list;
    CROSS_EDGE : *last;
};
\end{verbatim}

We keep track of a list-pointer, to indicate the first element (pointer to edge) in that
specific realization. The last-pointer is a pointer to the last element in the realization.
This pointer showed to be useful many times!
Structures like \texttt{CROSS.EDGE} and \texttt{CROSS.NET} are linked lists of the corresponding
elements (nets or edges).


Bibliography


