Chapter 1
Introduction

A VLSI circuit is usually modeled to be composed of *modules* and a set of *nets*. Each net specifies on the boundary of the modules as a subset of points, called *terminal*s, to be connected by wire. The *layout* problem is to interconnect the modules as specified by the nets in accordance with different technological design rules. Normally the *layout* problem is solved in four steps [6]:

Placement: the modules are placed on the plane.

Global routing: the routing region is partitioned into simple subregions, each called an elementary region, and global assignment of the wiring paths is determined for each net.

Routing region definition and ordering: The routing region is usually decomposed into rectangular channels and/or L-shape channels. Channels have to be ordered properly such that their size can be adjusted without rerouting the previous completed channels.

Detailed routing: detailed wirings of the individual routing regions are given.

A VLSI layout (Figure 1-1) consists of some modules, regions and nets. In Figure 1-1 a sketch of a global routing is shown, in which each net is specified by a list of regions through which its wiring passes. For instance, for Net 5, the sequence of regions is (*Rc, Rd, Rg*). Net 1 and Net 3 will have a crossing, if we require that the sequence of regions passed through for each net,

i.e., the homotopy, be fixed as specified. The positions crossed by the nets at the boundary of two adjacent regions are referred to as the junction terminals. Once the ordering of the junction terminals, say left to right for each horizontal boundary edge and bottom to top for each vertical boundary edge, is fixed, the number of crossings for each region is thus determined. Indeed, crossings in VLSI layout imply the usage of vias and require one or more layers in detailed routing [5]. As vias take up routing area, the number of vias allowed for each routing region may have to be restricted. The crossing distribution problem (CDP) calls for a specification of the ordering of the junction terminals for all the nets such that every routing region has no more crossings than allowed. That is, we must appropriately distribute these crossings into regions in order not to violate the quota of vias in each region. The problem was recently studied by Groenveld [3]; Marek-Sadowska and Sarrafzadeh [5]; Wong and Shung [7]; and Song and Wang [6]. In [5] Marek-Sadowska and Sarrafzadeh presented an O(MlogM) algorithm for solving a CDP, where M is the number of non-redundant crossings of the global routing. Song and Wang [6] presented an $O(n^2)$ algorithm for CDP, where n is the number of two-terminal nets between two regions. In this thesis we study the crossing distribution problem of two-terminal nets in two regions and present an $O(n\log n)$ algorithm. The result can be generalized to cases where there are multiple regions. In Chapter 2 we give the problem definition and provide detailed algorithms for solving the CDP for two-terminal nets. In Chapter 3, we consider the problem for three-terminal nets and give an $O(m+n\log n)$ time algorithm, where m is the ill combination number (we will define in chapter 3). Another popular problem in VLSI routing is the crossing minimization problem. This problem has been studied by [1] [3] and [5]. In Chapter 4 we will introduce a totally different model about this problem which is motivated by the problem considered in Chapter 3.

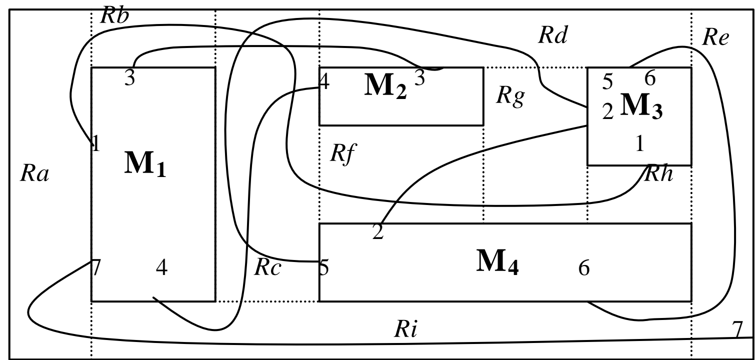


Figure 1-1 An example of VLSI layout.

Chapter 2
Crossing Distribution Problem with Two-terminal Nets

2.1 Problem definition

Without loss of generality, we consider the case where the routing region can be modeled as a circle and terminals are located on the circumference, as shown, e.g. in Figure 2-1. Imagine we have a cut line, the boundary, which cuts across the circle and divides it into two regions. This routing region can be further represented as a two-shore channel routing region as shown on the lower of Figure 2-1.

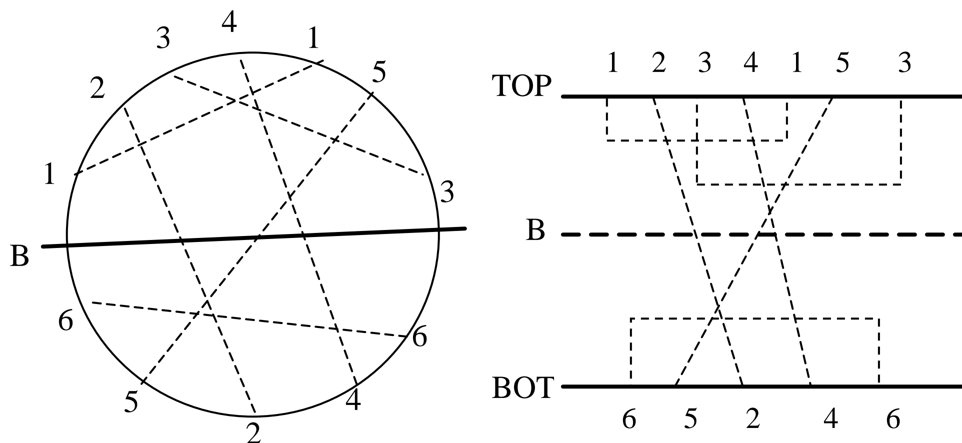


Figure 2-1 The crossing distribution problem (CDP) of two-region.

Let TOP and BOT be two horizontal lines on which terminals are placed. A two-terminal net $v = (p, q)$ is *two-sided* if v has a bottom terminal p on BOT and a top terminal q on TOP. A two-terminal net $u = (p, q)$ is one-sided if both p and q are on either BOT or TOP. A *crossing* is an intersection of two

different nets. We distinguish the *inherent* (*necessary* or *force*d) crossings and *redundant* crossings [6]. Intuitively, an inherent crossing between two nets is the one that cannot be removed by a connection homotopy [5], Consider two nets a and b, three different types of inherent crossings between a and b are shown in Figure 2-2. Some redundant crossings are shown in Figure 2-3.

In this chapter we eliminate redundant crossings whenever possible. It is trivial to eliminate redundant crossings for two-sided nets. For one-sided nets this task is not difficult either, and it will be discussed in Section 2.4.

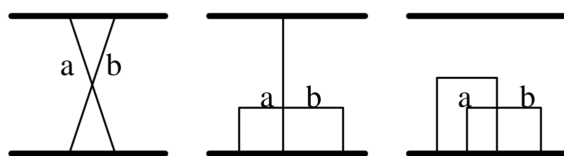


Figure 2-2 Three types of inherent crossings between a and b.

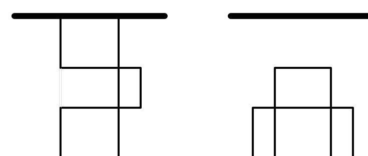


Figure 2-3 some redundant crossings.

Let R_1 and R_2 denote two routing regions sharing a common boundary B. Given a global routing of N two-terminal nets whose terminals are located on the boundary of the routing regions R_1 and R_2 respectively. Let C denote the total minimal number of crossings that exist in R_1 and R_2, and K an integer (K **£** C). The two-region crossing distribution problem is to find an ordering of the nets (junction terminals) at the boundary B such that exactly K crossings are located in R_1, and C-K crossings are located in R_2. As mentioned before, we assume that the boundary of the two adjacent routing regions, R_1 and R_2, can be represented by a circle whose circumference contains the terminals. Furthermore we assume that the top part of the circle refers to region R_1 and the bottom part to region R_2. Fig. 2-1 gives an illustration. Let TOP and BOT

represent the top and bottom boundary, respectively, that contain an ordering of the nets whose global routes are dissected by boundary *B*.

The crossing distribution problem in two regions R_1 and R_2 is to determine net orderings of the junction terminals at the boundary *B* such that no redundant crossings are introduced and crossings are “properly” distributed between these two regions. Formally, we define the following:

Problem I. Let $\boldsymbol{\Psi}$= {N_1 , N_2, ..., N_n} be a set of *n* ($n \geq 1$) nets. Let TOP = {$a_1, a_2, ..., a_u$} and BOT = {$b_1, b_2, ..., b_v$} be two sequences of terminals of the nets on the top and bottom lines, respectively, where a_i(or b_j) refers to a net number representing a net terminal and *i* denotes the terminal position ($1 \leq i \leq$ u, $1 \leq j \leq v$) on TOP or BOT. Given a boundary *B* and an integer quota *K*($K \leq C$, *C* is the total number of crossings), distribute exactly *K* crossings to R_1 and *C*-*K* crossings to R_2.

Let *P* denote the set of crossings in the global routing in two regions. Let *X, Y, Z* denote the set of one-sided nets on TOP, two-sided nets and one-sided nets on BOT, respectively. The set *S* of nets is thus partitioned into three pairwise disjoint subsets, *X, Y, and Z*. That is, we have $S = X \cup Y \cup Z$, $X \cap Y$ = *Æ*, $X \cap Z$ = *Æ*, and $Y \cap Z$ = *Æ*. Let (*A, B*) denote the crossing between nets *A* and *B*. Define the following three sets of crossings [6]:

$$P_1 = \{(A, B) \,/\, ((A \in X) \wedge (B \in X)) \vee ((A \in Y) \wedge (B \text{ Î } X)) \vee ((A \text{ Î } X) \wedge (B \text{ Î } Y))\}$$

$$P_2 = \{(A, B) \mid (A \in Y) \wedge (B \text{ Î } Y)\}$$

$$P_3 = \{(A, B) \mid ((A \text{ Î } Z) \wedge (B \text{ Î } Z)) \vee ((A \text{ Î } Y) \wedge (B \text{ Î } Z)) \vee ((A \text{ Î } Z) \wedge (B \text{ Î } Y))\}$$

Figure 2-4 shows an example, in this case P_1 = {(1,2), (1,3), (1,4), (2,5), (2,6)}, P_2 = {(3,4),(5,6)}, P_3 = {(7,8), (7,3), (7,4), (8,5), (8,6)}. Note that *P*= P_1

$\cup P_2 \cup P_3$, and these three case are mutually exclusive. We therefore divide our original problem into three sub-problems, one for two-sided nets (P_2), and two for one-sided nets (P_1 and P_3).

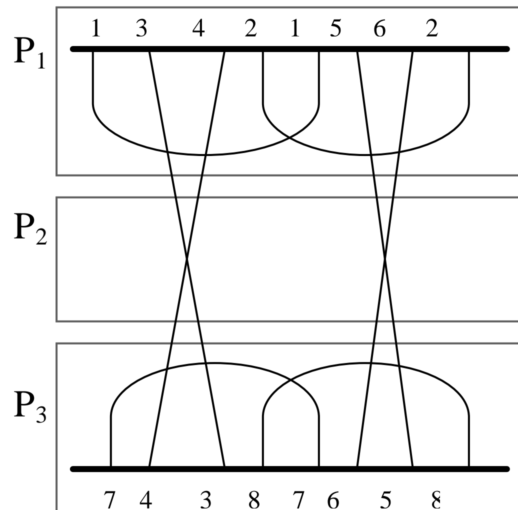


Figure 2-4 Three types of crossing P_1, P_2 and P_3

With the definition above, we can divide *problem 1* into *problem II* and *problem III* below:

Problem II. Crossing distribution for two-sided nets only (P_2). Let $\Psi = \{N_1, N_2, ..., N_n\}$ be a set of n(n ≥ 1) two-sided nets. Let TOP = $\{a_1, a_2, ..., a_n\}$ and BOT = $\{b_1, b_2, ..., b_n\}$ be two sequences of terminals of the nets of Ψ on the top and bottom lines, respectively, where a_i(or b_i) is a net number representing a net terminal, and i denotes the terminal position ($1 \le i \le n$). Given a boundary B and an integer quota K ($K \le C$, the total number of crossings), find a permutation of net terminals at the boundary B such that exactly k crossings are located at R_1 and $C–K$ crossings are located at R_2.

Problem III. Crossing distribution for two-sided nets in the presentce of one-sided nets(Figure 4 P_1,P_3). Let $\Psi = (N_1, N_2, ..., N_n)$ be a set of n ($n > 1$) nets such that N_i ($i = 1, ..., n$) is either a one-sided net on a line G or a two-sided net having a terminal on G. Let $L = (t_1,t_2, ..., t_r)$, $r \leq 2n$, be the sequence of terminals of the nets in Ψ on G. Given a boundary B and an integer quota K ($K \leq C$, C is the total number of crossings), distribute exactly K crossings above B.

2.2. Crossing distribution for two-sided nets

Without loss of generality, we assume TOP = (1, 2, 3, …, n) and BOT is a permutation of (1, 2, 3, …, n). Two-sided nets $N_i = (p,q)$, $N_j = (r,s)$ have crossing iff ($p<r$ and $q>t$) or ($p>r$ and $q<t$). In [6] Song and Wang note that the number of crossings is equal to the number of inversions of the permutation [2] [4]. For instance, permutation (4,2,1,3) has inversions (4,2), (4,1), (4,3), (2,1). When we place (1,2,3,4) on TOP and (4,2,1,3) on BOT, and then connect the same numbers with straight lines, we will find 4 crossings and they are exactly (4,1), (4,2), (4,3) and (2,1).

Note that given a sequence of numbers, we know that number i at position s_i and number j at position s_j have an inversion if $j < i$ and s_j is larger than s_i. For each integer $i \in$ (1, 2, …, n), the number of inversions induced by i can be obtained as follows. Since there are i -1 numbers less than i, if we know the number m of such integers lying to the left of i in the permutation, then the number of inversions induced by integer i is i -1- m. We call the number m the *magic number* of i. We will discuss this in more detail in Section 2.3.

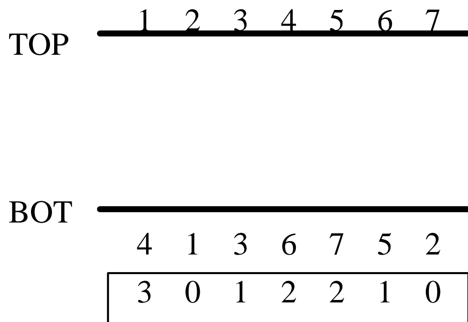


Figure 2-5 Crossing number with net' s number less than it

Consider Figure 2-5. The magic numbers for the integers in the sequence on BOT are respectively 0, 0, 1, 3, 4, 3, and 1 respectively (i-1-m). The numbers shown in the rectangular box in Figure 2-5 are the inversion numbers induced by the integer immediately above.

How to count this number efficiently is described in the next section. Now we consider the lemma below:

Lemma 2.1: *The sum of all the magic numbers of a permutation of (1, 2, …,n) is equal to the inversion number of the permutation.*

Proof: Obvious. ð

As will be shown later, CDP for two-sided nets can be solved easily once we compute the magic number for each integer. We now give an algorithm to compute a permutation of (1,2,…,n) whose inversion or crossing number is $k \leq$ C, where C is the total number of inversions of a given permutation BOT of (1,2,…,n). We build a list B-list to record the permutation on boundary B. Initially the permutation is the identity permutation, i.e., the same as TOP. In other words, all crossings are located at R_2 (in the lower region). We also

maintain a pointer array P-ary to every node of B-list. If we output a node from the B-list, we delete it at the same time.

Algorithm 2.2: Crossing distribution for two-sided nets

Input: k an integer; $BOT=(b_1,\ldots,b_n)$

Output: a sequence of numbers

1. Build *B-list* as (1,2,…,n) and pointer array *P-ary,* one for each entry in *B-list.*
2. Scan *BOT* from b_1 to b_n
    2.1. *if* $k = 0$ *then* break
    2.2. find current node b_i*'s* magic number m, and its inversion $r = b_i - 1 - m$
    2.3. *if* $(r > k)$ *then* output first r-k nodes of *B-list*
        output b_i; k=0
    2.4. *else* output b_i; $k = k - r$
3. Output the rest of nodes of *B-list* in linear order.

Let us illustrate this algorithm with an example. In Figure 2-6 the initial *B-list* is the same as TOP, we assume k= 5 and that the list of inversion numbers for each integer in BOT has been pre-computed. The first number of BOT is 4 and its inversion number is 3. Since $k = 5 > 3 = $ r, we output 4 (and delete 4 from B-list) and k is set to k-3 = 2 (See Figure 2-7).

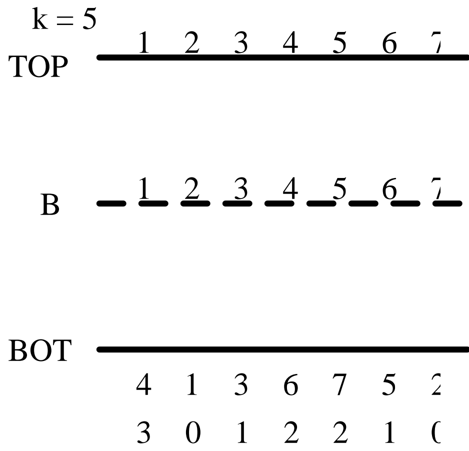


Figure 2-6 Initial situation, assuming magic numbers are given

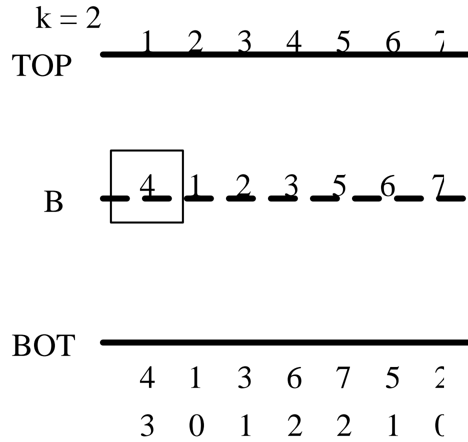


Figure 2-7 After an iteration, 4 has been output and *k* becomes 2

In Figure 2-7 the square containing 4 on *B* is the current output sequence. Let us pause for a moment here. The initial sequence routing of the first four nets as shown in Figure 2-8 causes four crossings in R_2. In Figure 2-9 we move 4 to the head of the permutation. This movement will cause 3 crossings to relocate in region R_1. This is due to the fact that each time we interchange two adjacent elements in a permutation, we will increase or decrease the total number of inversions by one [4]. Now the next element is 1 and its inversion number is 0. We do nothing but output this number (Figure 2-10). Next we find 3 and its inversion number 1 (Fig.2-11). Since $k = 2 > 1 = r$, we output 3 and update k to be 1. Now we have four crossings in R_1.

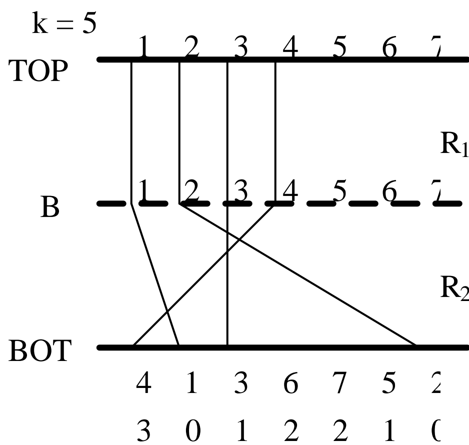


Figure 2-8 Four crossings in R_2

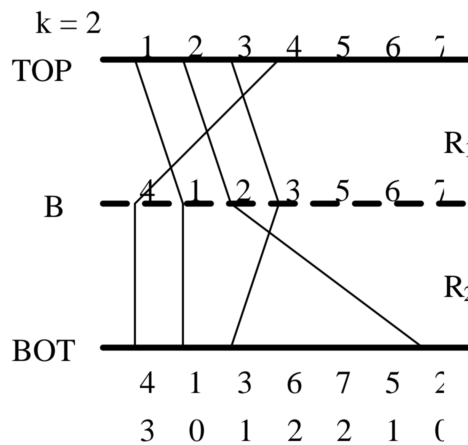
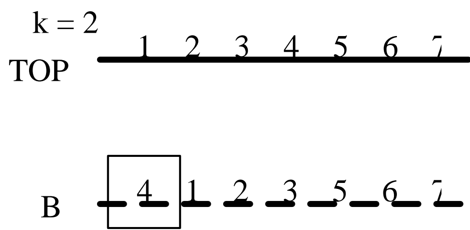


Figure 2-9 Three crossings in R_1 and one crossing in R_2



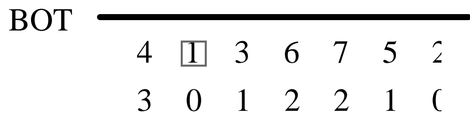
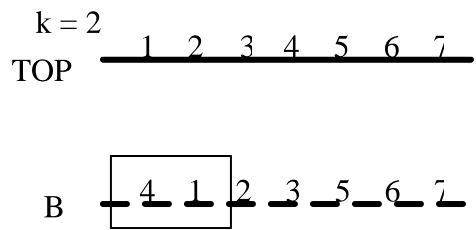


Figure 2-10 1 is scanned next.



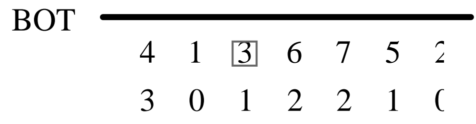
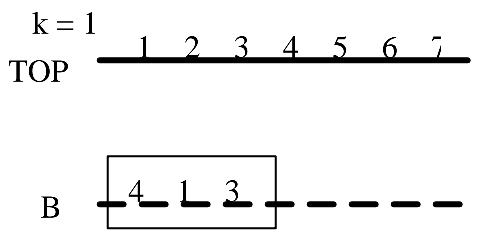


Figure 2-11 3 is scanned next.

When we find 6, $k = 1 < 2 = r$, we according to Step 2.3 start to output B-list r-k nodes. It means that we only need to make k exchanges of adjacent positions to meet the quota requirement. So we output r-k nodes, then output 6, and set k to 0. We will break this iteration and do step 3 to output other nodes of *B-list* (see Figures 2-12, 2-13). Figure 2-14 is the final routing result, which satisfies the quota requirement and there is no redundant crossing.



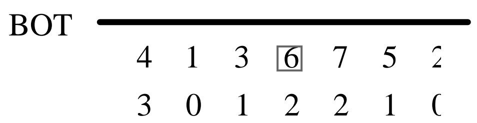
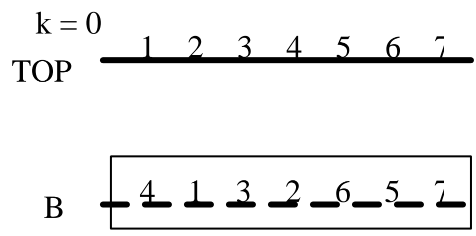


Figure 2-12 6 is scanned next but m > k.



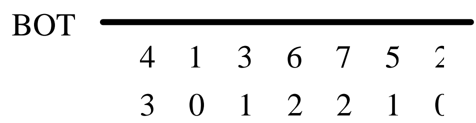


Figure 2-13 Final situation.

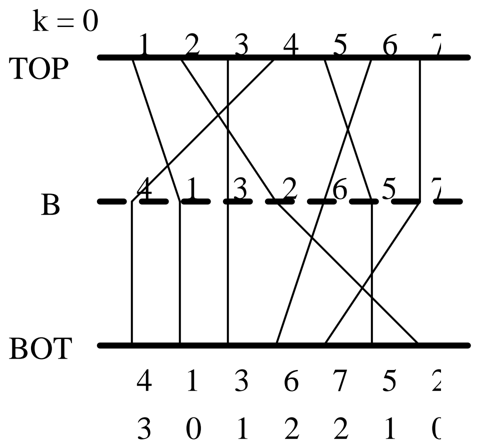


Figure 2-14 Totally 9 crossings. 5 in R_1, 4 in R_2.

Theorem 2.3: The two-sided CDP problem for n nets can be solved in O(nlogn) time.

Proof:

When $k = 0$ or when we have scanned the whole BOT list, the algorithm terminates. Since the algorithm does not create redundant crossings (two-sided nets by default have no redundant crossing), it satisfies the quota requirement [6], so it is correct. Let us now analyze the time complexity of the algorithm. It is obvious that other than the time needed to compute the magic/inversion number for each element in BOT, the total time needed in linear. As we will show below the time needed to compute the magic numbers and hence the inversion numbers is $O(n\log n)$. This completes the proof.

2.3 Finding magic number efficiently

To find the magic number of b_i, we should find out how many numbers are less than b_i which occur before b_i in the permutation. If we record all the scanned elements, we can easily compute the number of scanned elements that

are smaller than the current element. The inversion number induced by the current element is obtained by simply subtracting the magic number plus 1 from b_i. So if we could find the magic number in $O(\log n)$ time, the total time complexity would be $O(n\log n)$.

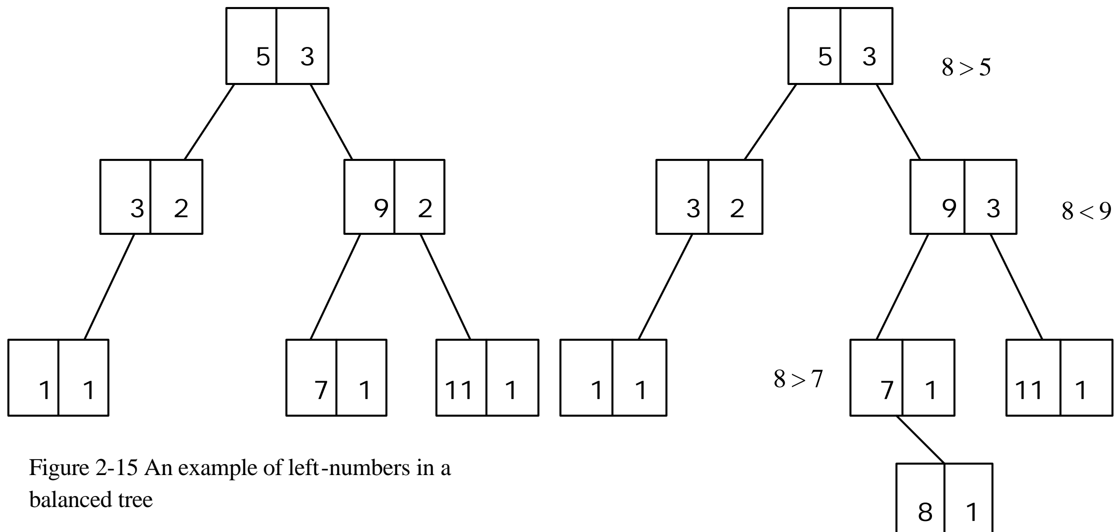


Figure 2-15 An example of left-numbers in a balanced tree

Figure 2-16 After inserting 8 to balanced tree

Since this problem is rather straightforward, we illustrate this by using an example. The underlying data structure used is a height-balanced binary search tree. Basically for each node we store a key plus an additional field, called *left-number*, containing the total number of elements stored in its left subtree plus one. That is, for each key, we store the total number of elements in the current binary search tree less than or equal to itself. When an element is scanned, it is inserted into the tree in an appropriate position. In the meantime, its magic number is computed as follows. Initially it is 0. Each time a right subtree of a node v is followed, the magic number is incremented by left-number(v). When a left subtree of a node v is followed, left-number(v) is incremented by 1 (indicating that the newly inserted number is less than key(v)). Since insertions may result in height imbalance, rotations to restore height

balance will be needed. All these operations are known to take $O(\log n)$ time per insertion for height balanced binary search trees.

See Figure 2-15, 5's left-number is 3, since it has 2 nodes in its left subtree. Similarly 9 has "7" in its left subtree so its left-number is 2.

See Figure 2-16. When we insert 8 into this tree, since $8 > 5$, we follow the right subtree, magic number is now 3. Going down the tree we meet 9. Since $8 < 9$, we follow the left subtree, so the left-number of node 9 is incremented by 1. Finally, we insert node 8 into correct place and add its left-number 1 to obtain its final magic number, which is 4.

Algorithm 2.4: Inset a node into and find the magic number from a left-numbered tree

Input: a node p; a left-numbered height-balanced tree T

Output: magic number m of p

1. m=0; u = T.root
2. *if* p.key > u.key

    then m = m + u.left-number

        u = u.right

    else u.left-number=u.left-number+1

        u = u.left
3. *if* u = null

    then insert p into u

        return m

    else goto 2

2.4 CDP in the presence of one-sided nets

We will only discuss P_3 here (one-sided net on BOT), The method for P_1 is similar. In [6] Song and Wang enumerate all crossings and use a topological sort to solve this problem. The crossing number is bounded by n^2, so their algorithm takes $O(n^2)$ time in the worst case. We show below that we need not enumerate all crossings to solve this problem.

In this section we also use the left-numbered height-balanced tree to solve this problem. We make some observations to help us understand this problem better. Figure 2.17 shows an example of one-sided net routing, and N_3, N_5 and N_7 are two-sided nets, others are one-sided nets. Let each one-sided net N be denoted by (Begin(N), End(N)), where Begin(N) and End(N) denote the beginning and ending terminal of the net.

Definition: Net N_i is said to contain Net N_j, if and only if Begin(N_i) < Begin(N_j) < End(N_j) < End(N_i) (In Figure 2-17 Begin(N_1) is 1, End(N_1) is 7, and N_1 contains N_2.).

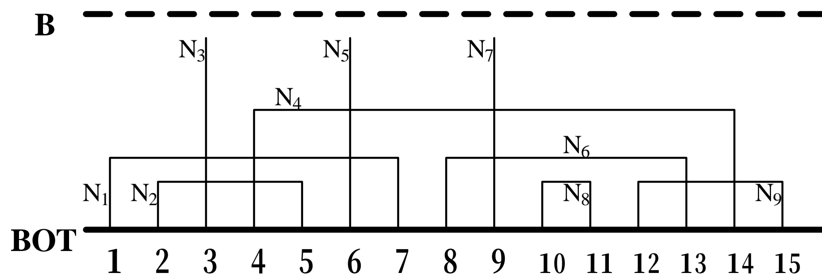


Figure 2.17 An example of one-sided net routing

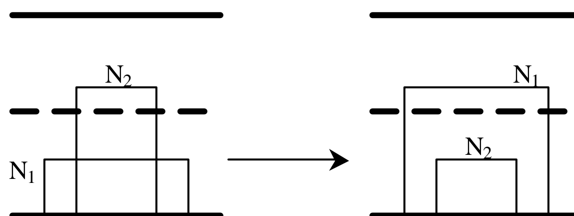


Figure 2-18 Prevent redundant crossings

In Figure 2-18 N_1 contains N_2. If we route N_2 in the upper region and N_1 in the lower region, two redundant crossings would occur. So if N_i contains N_j, we should always route N_i "above" N_j to prevent redundant crossings. This "above" routing constraint is implicitly maintained in our algorithm below when N_i is routed before N_j.

Lemma 2.5: If Begin(N_i) < Begin(N_j), N_j cannot contain N_i.

Proof: Immediate. ð

With lemma 2.5, we can use the order of beginning terminal to solve this problem. We will scan the BOT list once to find some information: (1) beginning position of all one-sided nets, (2) ending position of all one-sided nets, and (3) positions of all two-sided nets. This information can be found in linear time.

After finding this information, we initialize a left-numbered height-balanced tree by inserting all two-sided nets with positions as keys. This tree records the current order of net lines piercing boundary B. In other words, it records the current permutation of two-sided net on boundary B.

Algorithm 2.6: Crossing distribution in the presence of one-sided nets.

Input: BOT = $\{b_1,b_2,\ldots,b_n\}$; k an integer.

Output: B = $\{c_1,c_2,\ldots,c_m\}$ a permutation on B

1. find the information of all nets about their positions.
2. build a left-numbered height-balanced tree T by inserting two-sided nets in order
3. scan BOT from b_1 to b_n

3.1 let current node be b_i, and its correspond net be called N_j

3.2 *if* N_j has been processed (we encounter its right terminal) or it is a two-sided net *then* continue

3.3 find left numbers of Begin(N_j) and End(N_j) in T, called m_1 and m_2

3.4 *if* m_2-$m_1 \geq k$ *then*

insert Begin(N_j) to T; Break

else

insert Begin(N_j) and End(N_j) to T; $k = k - (m_2\text{-}m_1)$; continue.

4. output nodes in tree T by inorder traversal. After outputting Begin(N_j), we continue outputting k nodes, then outputting End(N_j). Finally, we output other nodes in tree T.

In Step 3.3, the number m_1 (respectively m_2) denotes the number of nets piercing the boundary B and lying to the left of Begin(N_j) (respectively End(N_j)). So m_2-m_1 denotes the number of nets piercing the boundary and lying between the two terminals of one-sided net N_j. For instance, Figure 2-19 shows an initial state of a one-sided net routing, and the initial left-numbered tree contains N_3, N_5 and N_7. At first iteration of Step 3 we find N_1. Using Begin(N_1) and End(N_1) to find m_1=0, m_2=2. Quota $k = 5 > 2 = m_2\text{-}m_1$, so we insert Begin($N_1$) and End($N_1$) to the left-numbered tree and decrease k by 2 (2-20).

The meaning is that we route net N_1 in the upper region R_1, we will make m_2-m_1 crossing to occur in R_1. Now N_{1b}(beginning) and N_{1e}(ending) behave as if they were two-sided nets after this iteration, so we insert them to the left-numbered tree.

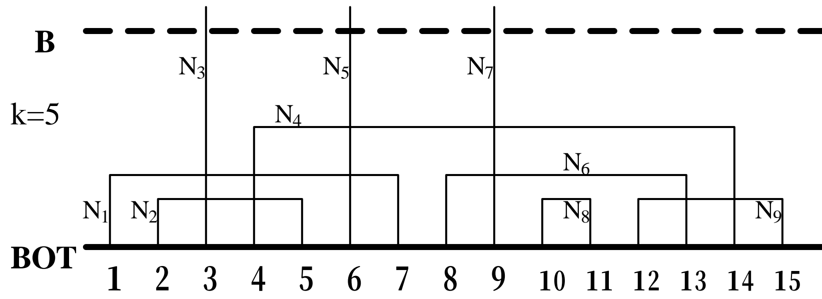


Figure 2-19 Initial state

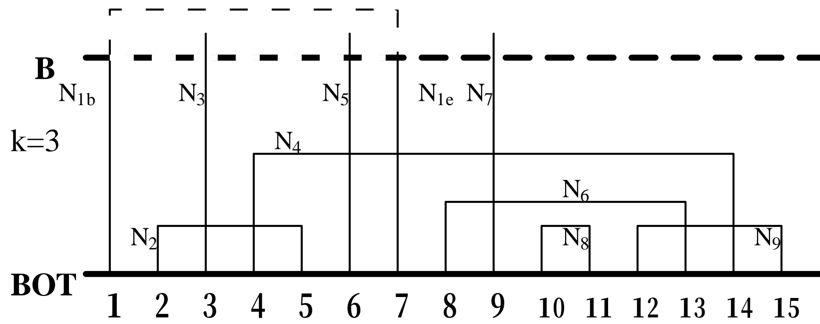


Figure 2-20 After processing N_1

On the next iteration, N_2 is scanned, and m_1=1, m_2=2. Since $k = 3 > 1 = m_2\text{-}m_1$, Insert Begin($N_2$) and End($N_2$) to the left-numbered tree and decrease k by 1 (Figure 2-21).

Next finding N_4, we have m_1=3, m_2=7. $k = 2 < m_2\text{-}m_1$, we insert Begin(N_4) into T to obtain the sequence (N_{1b}, N_{2b}, N_3, N_{4b}, N_{2e}, N_5, N_{1e}, N_7), and go to Step 4. We then output 4 nodes in inorder traversal of T until Begin(N_4). So we output k (=2) nodes in T, then output End(N_4). And than others in T are outputting. That is, the final permutation is (N_{1b}, N_{2b}, N_3, N_{4b}, N_{2e}, N_5, N_{4e}, N_{1e}, N_7) and the routing is shown in Figure 2-22. Initially (fig. 2-19), there are 10 crossings in R_2. After running this algorithm, we have now 5 crossings in R_1 and 5 crossings in R_2.

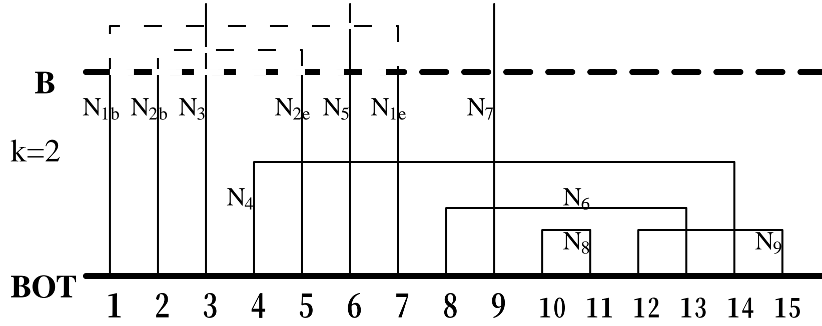


Figure 2-21After processing N_2

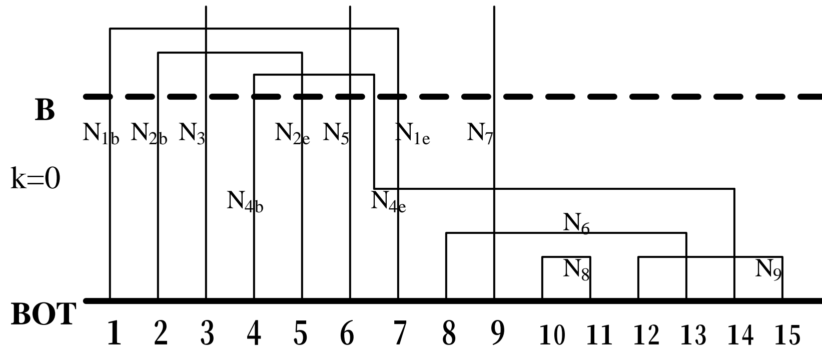


Figure 2-22 The final result

Theorem 2.7: The CDP in the presence of one-sided nets with n nets can be solved in O(nlogn) time.

Proof:

The algorithm terminates when $k \le m_2\text{-}m_1$ or the entire list BOT is scanned. The quota is not exceeded when $k > m_2\text{-}m_1$. When $k \le m_2\text{-}m_1$, we output Begin(N_i) at k positions to the left of End(N_i) to create k crossings in the upper region to meet the quota requirement. As for time complexity, Step 2 takes time $O(t\log t)$, where t is the number of two-sided nets. Steps 1 and 4 each need linear time, and Step 3 takes $O(n\log n)$ time for the operations needed for left-numbered tree searching and insertions. So it is totally $O(n\log n)$.

After solving these sub-problems, we now describe how we solve the original problem. Given a CDP with inputs: permutation of TOP, BOT and an

integer k, we first solve P_1 first. Let C(P_i) denote the crossing number in P_i, i=1,2,3. Initially, the permutation on boundary B is just like TOP and all the crossings (including P_1, P_2, P_3) are assumed to be in the lower region. Note that C(P_i) for i=1,2,3 can be obtained by scanning the permutations on TOP and BOT. If C(P_1)≥ k, we can find the solution after processing P_1. If C(P_1)< k, there remain k-C(P_1) crossings to be move into the upper region. We will process P_2 next. If C(P_2) < k-C(P_1), there remain k- C(P_1)- C(P_2) crossings. Noting that after P_2 has been processed (all crossings are moved to upper region), the permutation of B is just like BOT. So we can process P_3 in the final step.

Chapter 3

CDP for Three-terminal Nets in Two Regions

3.1 Problem definition

In [5] Marek-Sadowska and Sarrafzadeh introduced the *multi-terminal net problem.* They gave a deterministic algorithm for three-terminal nets in time $O(mn^2+m\boldsymbol{x}^{3/2})$, where m is the number of modules, n is the number of nets, $\boldsymbol{x}$ is the number of crossings, and a heuristic algorithm for multi-terminal nets in multi regions. In this chapter we study the CDP of three-terminal nets in two regions. It is not clear how to find the minimum number $\boldsymbol{x}$ of crossings. The crossing minimization problem (CMP) about the 3-terminal nets is not trivial. We will discuss this problem in chapter 4. If we have the homotopy of wiring that gives an optimal number of crossings, we can give an $O(n^2)$ algorithm to solve the problem.

We first distinguish all possible configurations of a three-terminal net containing 2 or 3 terminals. There are four types shown in Figure 3-1, referred to as I-type, U-type, Y-type and M-type respectively according to their appearance.

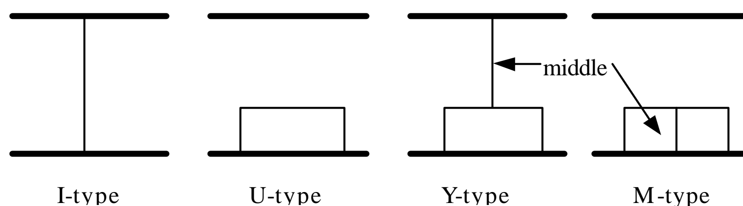


Figure 3-1 four types of nets.

We first need to guarantee that there are no redundant crossings. The only difference between a two-terminal net and a three-terminal net is that a three-terminal net may have an extra terminal, *middle terminal* denoted by Middle(N_i), in addition to beginning and ending terminals (Figure 3-1).

In the two-terminal version, there are three types of crossings (Figure 2-2). But in three-terminal version, there are many more crossing types. Thus, we list all the types in the Appendix A and pick some representative examples. The CDP of three-terminal nets can be divided into three parts just like two-terminal version (Figure 2-4). Let Q_1, Q_2 and Q_3 represent the three parts of problems corresponding to P_1, P_2 and P_3 of two-terminal version respectively.

3.2 CDP of three-terminal nets on Q_2

The same as before, we start with Q_2. Q_2 contains many types of crossings (Figure 3-2): (1) I-type with I-type, (2) I-type with Y-type, and (3) Y-type with Y-type. The Y-type net in theory has a special feature: the middle terminal can be connected to any position of the U-type base defining the beginning and ending terminals (Figure 3-3 (a)). In Figure 3-3 (b), by moving the position of middle net, we can prevent a redundant crossing. How to determine the positions of the middle terminal for all the Y-type nets to minimize the crossings becomes another problem, *crossing minimization problem (CMP)* [1]. This is a non-trivial problem and we will discuss it in chapter 4. From now on we shall assume that the position of the middle terminal for each Y-type net is fixed and given in advance. Once this information is given, Q_2 can be solved by the method given in chapter 2 within $O(n\log n)$ time.

Lemma 3.1: The problem Q_2 of CDP can be solved in $O(n\log n)$ time, when the middle terminal positions of Y-type nets are given. ÿ

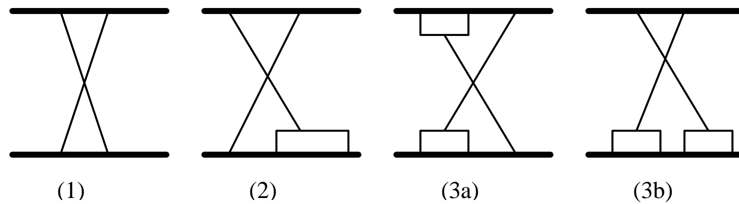


Figure 3-2 crossings in Q_2: (1) I-type and I-type, (2) I-type and Y-type, (3a) (3b) Y-type and Y-type.

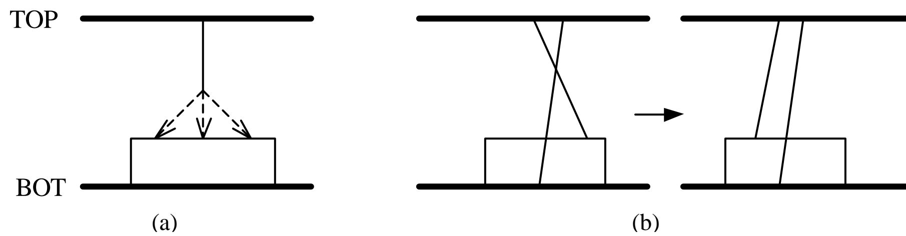


Figure 3-3 (a) The middle terminal can connect to an arbitrary position between beginning and ending terminals. (b) Move middle terminal to remove redundant crossings.

3.3 CDP of three-terminal nets on Q_1 and Q_3

We will consider Q_3 now (Q_1 is similar). We treat an M-type net like two U-type subnets in succession and the base (the two-terminal end) of Y-type net like a U-type net.

Figure 3-4 presents two examples: in the example A, the method of Section 2.4 works, namely, we treat the M-type net as if there were two U-type subnets; but in the example B, if we apply the method, it will create a redundant crossing, unless the right U-type subnet (of the M-type) is also routed in the upper region along with the left U-type subnet.

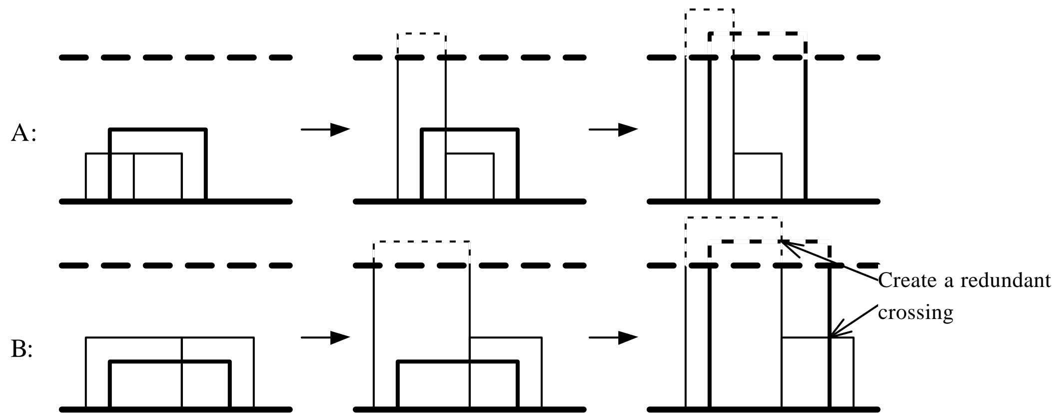


Figure 3-4 A: The method of Chapter 2 works; B: The method of Chapter 2 creates a redundant crossing.

When a U-type or an M-type net is contained in another M-type net, as shown in Figure 3-5, where the inner net crosses the middle terminal of the outer net, we call this an *ill combination*. Routing for ill combination pairs needs extra treatment, for otherwise redundant crossings may result. Indeed, these two cases are the same. The left-subnet of inner M-type net in Figure 3-5 (b) makes no effects to the combination. Figure 3-5 (b) has a symmetric case that the left-subnet route across the middle terminal of the outer net, but this is similar. So we will always take Figure 3-5 (a) as an example when presenting this problem.

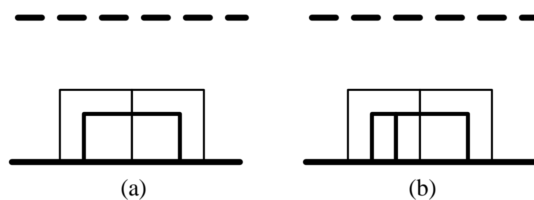


Figure 3-5 Two situations that break the method in Chapter 2.

Figure 3-6 lists all possible interactions between U-type nets and M-type nets, and between two M-type nets, excluding ill combinations. By treating

M-type nets as two consecutive U-type subnets and using the method in Section 2.4, we can easily verify the correctness of the algorithm for Q_3 in case no ill combinations exist. The following lemma is obvious.

Lemma 3.2: If there are only I-type, U-type, M-type nets and there exists no ill combination, we can solve the Q_3 of CDP in O(nlogn) time.

Proof:

Since the middle terminal positions of the Y-type nets are fixed, the problem reduces to that of the two-sided two-terminal nets routing problem. Therefore it can be solved in $O(n\log n)$ time using the same method given in Section 2.2. ÿ

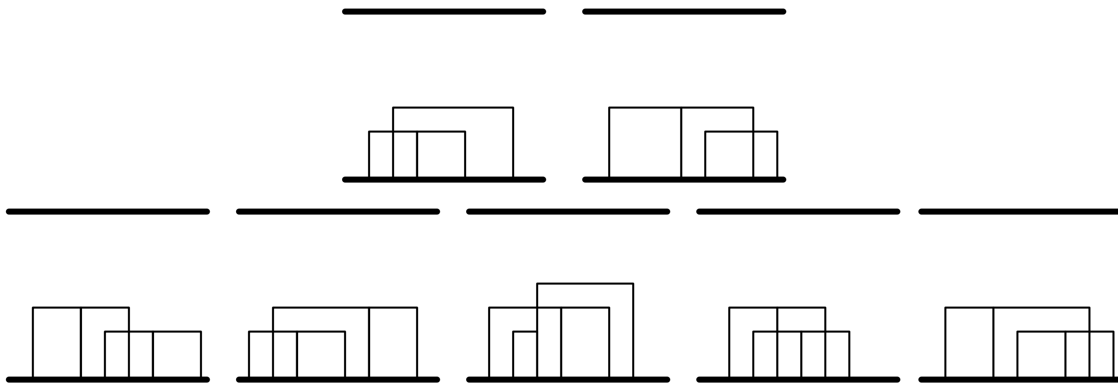


Figure 3-6 All possible crossings between a U-type and an M-type and between two M-types excepts ill combination.

We shall now address the problem of redundant crossings that may result due to ill combinations (Figure 3-5). Recall that when a U-type net N_i contains another U-type net N_j, net N_i should be routed above N_j to avoid creating redundant crossings. In our algorithm given in Section 2.4, N_i gets processed before N_j and N_i does implicitly get routed above N_j. A U-type net U_i and an

M-type net M_j form an ill combination when the M-type net M_j contains the U-type net U_i and the middle terminal *m* crosses the U-type net U_i causing one inherent crossing (Figure 3-5 (a)). When the left U-type subnet of M_j and the U-type net U_i have been processed and thus both get routed in the upper region, it is possible that the algorithm terminates because the quota requirement has been met. In this case there will be a redundant crossing caused by the pair of right U-type subnet of M_j and U-type net U_j as shown in Figure 3-4 B. We thus need to ensure that no redundant crossing occurs in each of our processing steps. In other words, when the algorithm terminates, we should guarantee that three exists no redundant crossing.

We adopt the strategy, called delay processing, to solve the problem. Consider Figure 3-7 for example. Since routing the whole inner U-type net into the upper region may cause redundant crossings, we can route just the initial part of the U-type net into upper region (Figure 3-7 picture 2) by creating a "jog" or a bend at the leg associating with the end-terminal of the U-type net. After the right U-type subnet of the outer M-type net has been routed into the upper region, the inner U-type net will be completely routed into upper region (Figure 3-7 picture 3), if needed. This inner U-type net will be referred to as a *floating* net and the leg associating with the ending terminal of the floating net as *floating end*. If an inner net has many outer M-type nets with which to form ill combinations, its floating end will float at each of the middle terminals of every outer M-type net. Figure 3-8 shows an example. Figure 3-8(a) shows the initial condition, noting that N and M_1 form an ill combination, and N and M_2 also form an ill combination. Consider Figure 3-8(b), in which both Begin(M_1) and Begin(M_2), i.e., the left U-type subnets, have been processed. To prevent redundant crossing we must process the floating net N in an appropriate manner.

In Figure 3-8 (c) Middle(M_1) lies to the left of Middle(M_2), so the floating end of N will float till Middle(M_1) at the beginning. In Figure 3-8(d) after Middle(M_1) has been processed at which point, M_1 and N will not cause a redundant crossing, the initial part of N can be extended further, floating till Middle(M_2). In other words, the "bend" of the ending terminal of the floating net is considered "stretchable" and will become "straight" only when the scanning process reaches the position of the ending terminal.

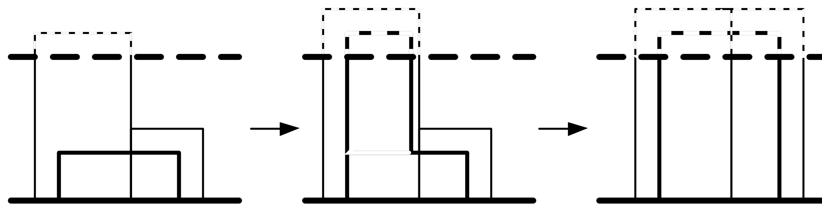


Figure 3-7 The strategy for preventing redundant crossings.

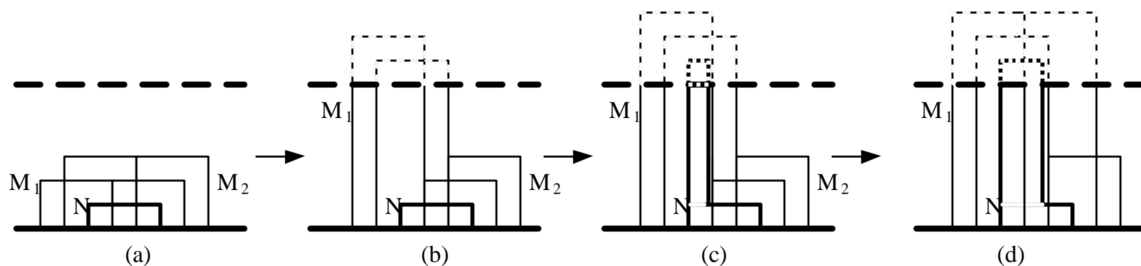


Figure 3-8 (a) Initial condition. (b) Begin(M_1) and Begin(M_2) have been processed. (c) N has been processed, but floats till middle(M_1). (d) Middle(M_2) has been processed, and N floats process till Middle(M_2)

We now describe our algorithm more formally. We shall first identify the pair of nets that form an ill combination, and give the routing constraint relation "above". Net A is "above" net B if and only if no redundant crossing will result when Net A is routed above Net B. Net B is a floating net if it is a U-type net totally contained in an M-type net M or a U-type subnet of an M-type net totally contained in another M-type net M and they form an ill

combination pair. We use Net A$\rightarrow$ Net B to denote the "above" routing constraint. In Figure 3-8, we have net $M_1 \rightarrow$ net N, and net $M_2 \rightarrow$ N. Note that this directed graph (for relation "above"), called *ill-combination graph*, has no cycles. For all nets M_i such that $M_i \rightarrow$ net N, we shall maintain for floating net N sorted list of the middle terminals of M_i at which the floating end of net N will float to in that order. The task to perform is to maintain the "above" routing constraint imposed by the ill-combination graph. To ensure that the floating ends of the floating nets float to appropriate middle terminal positions correctly we'll use the corresponding sorted lists and process the floating ends in an order-preserving manner. Consider Figure 3-8 for example. Suppose there is another floating U-type net N' which contains U-type net N and net $M_1 \rightarrow$ net N, net $M_2 \rightarrow$ N, $M_1 \rightarrow$ net N', and net $M_2 \rightarrow$ N'. Note that since net N' contains net N, net N' gets processed before net N and hence gets routed above net N. When we float net N till Middle(M_1) or till Middle(M_2), we must ensure that floating end of net N's must be processed before net N. That is, all those floating nets should be processed in an appropriate order so that the "above" routing relation can be preserved. Fortunately those floating nets have a natural ordering which is the same as the order of beginning terminals, or the order in which these nets are scanned. To identify which floating net needs to be processed as we scan the terminal positions, we need to maintain for each middle terminal of an M-type net the information of those U-type nets or subnets with which they form an ill combination pair. The set of ordered U-type nets or subnets is called the *out-neighbors set* of the corresponding M-type net.

Step 2 initializes the left-numbered height-balanced tree T by inserting all I-type nets. Also, a terminal number array recording the terminal numbers

between two adjacent middle terminals is initiated by the original elements in the initial left-numbered tree. This data structure is built for Step 3.5.3.

Step 3 is the main part. Step 3.3 and 3.4 do the delay process like Figure 3-7 and 3-8. The floating terminal is also inserted into the left-number tree. It ensures that the algorithm is correct whenever it terminates. We use the ill combination sorted list to handle floating terminals. Step 3.5 handles middle terminals. When a middle terminal of an M-type net is scanned, it means we will process the right subnet of the M-type net. After we process it, some ill combination pairs involving the M-type net may no longer exist, we will then proceed to finish processing of floating terminals, if any, of its out-neighbors. Now we describe how to use the terminal number array to assist the algorithm. In 3.5.4.1, checking the array of every net is in a monotone order. In other words, every net can only check the array from left to right. It makes the total access only needs $O(m)$ time, where m is the ill combination number. Every terminal is at most update the array one time when inserting into left-numbered tree within $O(n\log n)$ time. So totally operation about the array is $O(m+n\log n)$.

Algorithm 3.3: CDP of Q_3 containing I-type, U-type, M-type nets.

Input: BOT = $\{b_1,b_2,\ldots,b_n\}$; connecting information of Y-type nets; k an integer.

Output: B = $\{c_1,c_2,\ldots,c_m\}$ a permutation on B

1. Find all the ill combination pairs. For each floating U-type net (or subnet) construct a sorted list of the middle terminals of those M-type nets with which they form an ill combination. For each M-type net in an ill combination, we also maintain the information of those U-type nets (or subnets), i.e., the out-neighbors of the M-type net in the order of their

begin-terminal positions.

2. Build a left-numbered height-balanced tree T and a terminal number array A by inserting I-type nets in order; construct a empty queue Q for all floating nets that are yet to be processed. Initially Q is empty.

3. scan BOT from b_1 to b_n while k>0

   3.1 let current node be b_i, and its correspond net be called N_j

   3.2 *if* b_i is an end terminal or a terminal of I-type *then* continue (step 3)

   3.3 *if* N_j is a U-type net *then*

      3.3.1 check and delete the first element E of its ill combination sorted list. Use b_i and the middle terminal position e of E (if no elements in the queue, use End(N_j)) as keys to find left numbers m_1 and m_2 from T.

      3.3.2 insert Begin(N_j) with key b_i to T and update A;

      if m_2-$m_1 \geq k$ *then* insert End(N_j) into desired position; break;

      else

         if this net is floating *then* attach it to the middle terminal it floating for.

         else insert End(N_j) into T and update A.

      $k = k - (m_2$-$m_1)$; continue (step 3).

   3.4 *if* N_j is an M-type and b_i is a starting terminal *then*

      3.4.1 check and delete the first element E of its corresponding ill combination sorted list. Use b_i and the middle terminal position e of E (if no elements in the queue, use Middle(N_j)) as keys to find

left numbers m_1 and m_2 from T.

3.4.2 insert Begin(N_j) with key b_i to T and update A;

if m_2-m_1 $\geq$ k *then* insert Middle(N_j) into desired position; break;

else

if this net is floating *then* attach it to the middle terminal it floating for.

else insert End(N_j) into T and update A.

$k = k - (m_2\text{-}m_1)$; continue (step 3).

3.5 *else* N_j is an M-type and b_i is an middle terminal

3.5.1 check and delete the first element E of its corresponding ill combination sorted list. Use b_i and the middle terminal position e of E (if no elements in the queue, use End(N_j)) as keys to find left numbers m_1 and m_2 from T.

3.5.2 insert Middle(N_j) with key b_i to T and update A;

if m_2-m_1 $\geq k$ *then* insert End(N_j) into desired position; break;

else

if this net is floating *then* attach it to the middle terminal it floating for.

else insert End(N_j) into T and update A.;

$k = k - (m_2\text{-}m_1)$;

3.5.3 add all its outer neighbors in ill combination graph into Q by the order we mentioned before.

3.5.4 check all elements in Q

3.5.4.1 now we consider the net N_k. check and delete the first element E of its corresponding ill combination priority queue. Find the number of terminals m in A between [b_i, e) where e is the middle position of E (if no such E in the queue, check it like 3.3.1, 3.4.1, 3.5.1).

3.5.4.2 delete its floating relation it attaching before.

3.5.4.3 *if* $m \geq k$ *then* insert t into desired position; break;

else

if this net is floating *then* attach it to the middle terminal it floating for.

else insert the right terminal of this net or subnet into T and update A.;

$k = k - m$;

4. Output nodes in tree T by inorder traversal. When reaching a middle terminal, we output its attaching terminals in reverse order.

Now we want to describe the correctness of this strategy. We need to prove three things: (1) it will stop; (2) it prevents redundant crossings; (3) it distributes desired number of crossings into upper region.

The total size of the ill-combination sorted lists, and the total size of all sets of out-neighbors is *m*. So after at most $O(m+n)$ steps, it will stop. Within all our operations in the algorithm, we have prevented redundant crossings. And the crossing number is obviously satisfied.

The time complexity analysis is given in the following. Step 1 needs $O(m+n)$ time to construct the ill combination graph with the out-neighbor order and also $O(m+n)$ time to construct all ill combination sorted list. Step 2 needs $O(n\log n)$ time to construct the tree and the array. Step 3 has at most $O(m)$ iterations mentioned before, and every iteration needs $O(1)$ time to access the terminal number array. For every terminal, it accesses left-numbered tree at most once, inserts into left-numbered tree once, and update terminal number array once. All the operation before needs $O(\log n)$ time. So this step needs totally $O(m)$ time for accessing terminal number array and $O(n\log n)$ time for other operations. Step 4 needs linear time. So the whole algorithm has time complexity $O(m+n\log n)$.

Lemma 3.4: The CDP on Q_3 (Q_1) can be solved in $O(m+n\log n)$ time, when only I-type, U-type, and M-type nets can appear.

Note that the position of the middle terminal of a Y-type net must be placed in such a way that no redundant crossing may be created. For instance, if the position of the middle terminal of a Y-type net is fixed as shown in Figure 3-9, a straightforward routing may create a redundant crossing. We can adopt the strategy mentioned before (delay processing) to obtain a routing as shown in Figure 3-9 (c) in time $O(m+n\log n)$.

Lemma 3.5: the CDP on Q_3 (Q_1) can be solved in $O(m+n\log n)$ time, when the middle terminal positions of Y-type nets are determined.

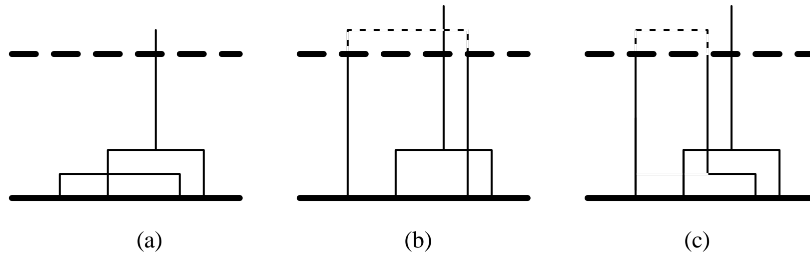


Figure 3-9 (a) a situation will cause the redundant crossing. (b) an example of causing redundant crossing. (c) the strategy to prevent the problem.

Theorem 3.6: The three-terminal CDP problem for n nets on two regions can be solved in $O(m+n\log n)$ *time, when the middle terminal positions of Y-type nets are determined.*

Proof:

By Lemmas 3.2, 3.3, 3.4, and 3.5, the time complexity is $O(n^2)$. ÿ

Chapter 4

Crossing Minimization Problem for Two-terminal Nets with Sliding Ends

4.1 CMP for two-terminal nets with sliding ends

We have introduced an interesting problem about the crossing minimization problem (CMP) in chapter 3. This problem has been studied by [1] [3] and [5]. In [3] Groenveld introduced CMP and presented an $O(m^2n)$ algorithm where m is the number of modules and n is the number of nets. In [3], the homotopy of nets in the global routing is fixed and cannot be changed during the process of crossing minimization. That is called constrained CMP (CCMP). But in [5] Marek-Sadowska and Sarrafzadah considered a variation of this problem, where the homotopy of the wiring can be changed in the process of crossing minimization. That is called un-constrained CMP (UCMP). Their algorithm has time complexity $O(mn^2+\boldsymbol{x}^2)$, where $\boldsymbol{x}$ is the total number of crossings. In [1] Chen and Lee presented an $O(mn)$ time algorithm for CCMP and an $O(n(m+\boldsymbol{x}))$ time algorithm for UCMP. In this chapter, we will introduce a totally different model of CMP.

In chapter 3, we simply introduced the model of this problem. We will explain in more detail in the following section. We consider a variation of the two-sided two-terminal net problem. Unlike the two-terminal net problem in the previous section, one of the two end terminals is allowed to slide within a given interval. We distinguish two cases: (i) top-fixed case, in which the top

terminal of the two-terminal net is fixed and (ii) bottom-fixed case, in which the bottom terminal of the two-terminal net is fixed. The other end terminal may slide within a given interval or a range. In what follows we will consider top- or bottom-fixed two terminal nets with sliding ends. When an end terminal is constrained in a range, this net is called a one-end-sliding net. If both terminals can slide, it is called a two-end-sliding net.

Figure 4-1 Top-fixed one-ended sliding net.

Figure 4-1(a) shows a top-fixed one-end-sliding net, and in Figure 4-1(b), all of the 5 nets are valid.

Since a terminal of a two-terminal net can slide within a range, the number of crossings will vary depending on its final position. Thus finding a position for the sliding end terminal so as to minimize the total number of crossings becomes an interesting problem. For example, in Figure 4-2(a), the dashed net is a top-fixed sliding net, and it is obvious that different placements of the bottom terminal within the range will result in one or no crossing.

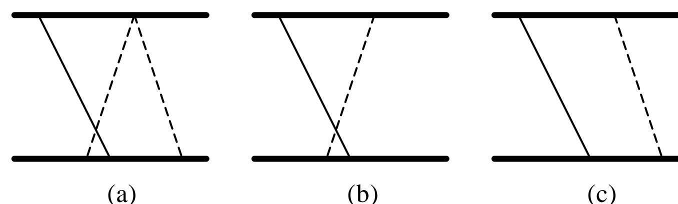


Figure 4-2 The connecting position of the sliding nets may affect the crossing number.

4.2 CMP for top-fixed two-terminal two-position nets

Before we study the sliding model of this problem, we introduce another model: two-position model (2P model). This model consists of a set of two-sided nets, and every net has a fixed end terminal and the other end terminal has two possible positions. If the fixed end terminal of a net is on TOP (respectively BOT), it is called a top-fixed (respectively bottom-fixed) 2P net. Figure 4.1 shows a top-fixed 2P net.

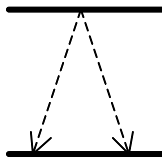


Figure 4-3 A top-fixed 2P net.

In this section, we focus on top-fixed 2P nets only, that is, only terminals on the bottom each have two positions to place the end terminal. Thus, we can obtain an ordering of nets using their positions on TOP from left to right. Figure 4-4 shows the ordering of the three nets, and the two positions for each net is shown at BOT.

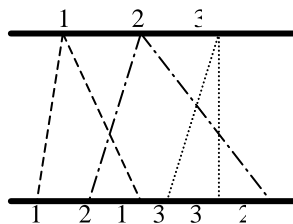


Figure 4-4 The ordering of the top-fixed 2P nets.

The working model is the following. There is a set of n top-fixed 2P

two-terminal nets {N_1, N_2,…, N_n} between TOP and BOT. The labels of these N nets are ordered from left to right on TOP.

Observation 4.1: Let us denote the two positions of the bottom terminal for each top-fixed 2P two-terminal net N_i by (l_i, r_i). For two nets N_i and N_j, $i<j$, if we move N_i to l_i and N_j to rj, they are more likely to avoid the crossing between them.

Proof:

For two nets N_i and N_j, $i<j$, we know that the top terminal of N_i lies at the left side of that of N_j. If we move N_i to l_i and N_j to r_j, it is more likely that they will not contribute crossings. Because if $l_i < r_j$, they will contribute no crossing. If $l_i > r_j$, they have an inherent crossing.

The strategy of our algorithm is iteratively inserting a net according to the Observation 4.1 mentioned above. Since the inserted net has the largest index currently, it is natural that we insert it at its right position. Other nets that were already routed in the region could move to left positions to avoid crossing. In the following we give our algorithm below.

Algorithm 4.2: Crossing minimization for top-fixed 2P two-terminal nets

Input: a set of n top-fixed 2P nets {N_1, N_2,…, N_n}

Output: an optimal routing of these nets that produce a minimum number of crossings

1. Initially, the region between TOP and BOT is empty.
2. Consider N_i, i from 1 to n
    2.1. Insert N_i at r_i, let S_i={N_j| N_j has a crossing with N_i}

2.2. For $N_j \in S_i$ (net of the smallest index j gets processed first)

2.2.1. Check if N_j causes a block movement (see definition below) that can decrease the number of crossings.

2.2.2. if so, perform a block movement.

2.3. Move a net to its right position if this movement does not increase the number of crossings.

3. Output the routing and the minimum number of crossings c.

A set of nets $\{N_{i0}, N_{i1}, \ldots, N_{ik}\}$ is said to form a move-block, if change of their end-terminal positions as a block will decrease the total number of crossings. In other words, all the nets in a move-block will have their end terminals moved. We say in this case that the net with the largest index induces the move-block. And we also want a move-block to decrease as many crossings in one block move as possible. Figure 4-5 shows an example. {2,3} and {2,3,4} are two move-blocks.

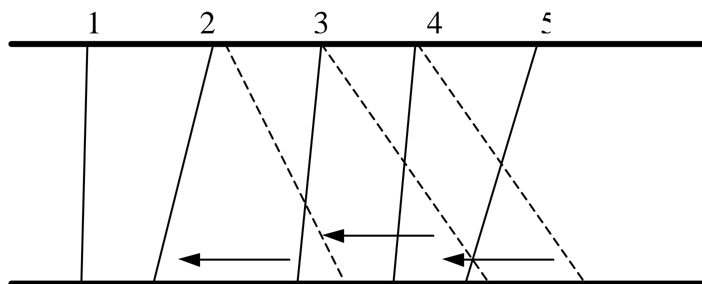


Figure 4-5 {2,3} and {4} are two move-blocks.

Lemma 4.3: After each iteration, a 2-position net assumes its left position if and only if the net being placed at its right position will cause extra crossings.

Proof:

(=>) Step 2.3 has ensured this situation.

(<=) If it were not the case, it would have broken the optimality

condition.

In other words, if a net's end terminal is moved to its left position, it must be forced to do so. The following is the main theorem of this section.

Theorem 4.4: Algorithm 4.2 solves the CMP of top-fixed 2P two-terminal nets.

Proof:

First of all, the algorithm terminates after all nets are considered.

Now we will prove the correctness of the algorithm that the resulting net placement is optimal. We claim that after inserting of N_i this algorithm finds the minimum number of crossings for $\{N_1,N_2,\ldots,N_i\}$. We prove it by induction. When N_1 is inserted, the claim holds trivially. Let us assume that the claim holds until when N_{k-1} is inserted. When N_k is inserted, we need to check the optimality. Note that only a move-block, which contains some nets of Si, can decrease the number of crossings. Or we could have performed the block move before insertion of N_k.

Let us assume the contrary that after N_k was inserted, the number of crossings is not minimum. That is, we can change the position of an end terminal of some net to left or right position to decrease the number of crossings. Thus, there must exist some crossings that get eliminated. We consider one of the eliminated crossings contributed by N_p and N_q, with largest q. We could either move N_p to left or move N_q to right to eliminate this crossing. By lemma 4.3, moving N_q to its right position will help nothing. So only moving N_p to its left position is considered. Since moving N_p to its left position decreases the number of crossings, N_p must belong to some move-block, say $B=\{N_p,\ldots\}$.

Case 1: If the move-block B contains a net of S_i, then this move-block would have been checked in Step 2.2, and N_p should have been moved to its left position at that time.

Case 2: Suppose now that the move-block does not contain any net of S_i. In this case, at the time when N_k is inserted, none of the nets that are moved cross N_p, for otherwise one of the crossing net would belong to some move-block B', which contains some net of S_i. And $B' \cup B$ would form a move-block and got moved at that time. Since no nets that are moved would cross N_p, routing before N_p is all the same as the situation before N_k is inserted. N_p must have been moved to its left position before insertion of N_k. We have a contradiction.

The bottleneck of this algorithm is Step 2.2.1. A naïve implementation of this Step is searching all possible move-blocks containing nets of S_i. Since the move-blocks are formed like an augmenting path, one net crossing the subsequent one, searching all possible move-blocks needs at most $O(n^2)$ time. So totally the algorithm takes $O(n^4)$ time.

Chapter 5

Conclusion and Future work

We have considered the crossing distribution problem (CDP) of n two-terminal nets of two regions and presented an $O(n\log n)$ time algorithm to solve this problem, improving upon a previously known result [6] which takes $O(n^2)$ time. We have also solved the CDP for three-terminal nets of two regions in $O(m+n\log n)$ time. The CDP for multi-terminal nets with more than three terminals remains to be a difficult problem.

By introducing more cutting lines $B_1, B_2 \ldots, B_t$ we can solve the CDP by distributing all C crossings in regions $R_1, R_2 \ldots, R_{t+1}$ each containing $k_1, k_2 \ldots, k_{t+1}$ =C crossings respectively in time $O(t*n\log n)$ by repeating the two-region CDP problem t times, if the *region adjacent graph* [5] is a path. When the routing region is not a simply connected region, i.e., it has holes whose boundary also contains terminals, the CDP problem becomes a lot harder. Whether one can find a more efficient algorithm than the previous result, which makes use of Max-flow model [5] remains to be seen.

Another problem that arises from the study of 3-terminal nets in chapter 3, is the crossing minimization problem (CMP) of sliding nets. We have solved the top-fixed model on CMP for two-position (2P) two-terminal nets. The 2P model can be easily extend to a k-P model. It means that every net has a fixed terminal and the other end terminal has at most k positions to choose. This is only a partial result to the general CMP. We are working on CMP for top-fixed sliding two-terminal nets and on that which contains both top-fixed nets and

bottom-fixed nets. CMP for the most general case where both end terminals are allowed to slide within a respective interval is an interesting problem worthy of further study.

REFERENCES

1. Hsiao-Feng Steven Chen, D.T. Lee, "*On Crossing Minimization Problem*", IEEETCAD: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.
2. Cramer, G. 1750. *Introduction l' analyse des lignes courbes algèbrique*s. Geneva, Geneva, Switzerland.
3. Groenveld, P, 1989, On global wire ordering for macro-cell routing. In *Proceedings of the 26th ACM/IEEE Conference on Design Automation* (DAC ' 89, Las Vegas, NV, June 25–29,1989), D. E. Thomas, Ed. ACM Press, New York, NY, 155–160.
4. Knuth, D. E. 1973. *The Art of Computer Programmin*g. Addison-Wesley Longman Publ. Co., Inc., Reading, MA.
5. Marek-Sadowska, M.AND Sarrafzadeh, M. 1995. The crossing distribution problem,. *IEEE Trans. Comput.-Aided Des. 1*4, 4 (Jan.).
6. X. Song and Y. Wang, 1999, "On the crossing distribution problem," in ACM Press , New York, NY, 39 – 51.
7. Wang, D.C. AND Shung, C. B. 1992. Crossing distribution. In *Proceedings of the European Conference on Design Automatio*n. 354–361.
8. T.K. Yu AND D.T. Lee, 2002. An O(nlogn) algorithm on the crossing distribution problem. *Conf. ICS 2002, C3-3, Taiwan, R.O.C.*

Appendix A

We have four types of nets, I, U, Y and M. Let the type I-Y means the crossing contributed by an I-type net and an Y-type nets. All the crossing types of three-terminal nets are showed below (the symmetric cases are omitted):