DOI:10.7631/issn.1000 - 2243.2013.06.1028 **文章编号**:1000 - 2243(2013)06 - 1028 - 07

利用树状数组的两区域电路线交叉分布

邓新国[1], 高董英[1], 郭朝珍[1], 肖如良[2]

(1. 福州大学数学与计算机科学学院, 福建 福州 350116; 2. 福建师范大学软件学院, 福建 福州 350117)

摘要: 在两区域电路线交叉分布中计算交叉点的数目, 目前采用线性表或者动态规划的方法, 其算法时间复杂性均为 $O(n^2)$. 为有效降低现有算法的时间复杂性, 给出一种时间复杂性为$O(n\log n)$、利用树状数组的计数算法, 并且可以找到每条布线的所有交叉线. 理论分析和相应实验结果证实了该算法的有效性.

关键词: 电路布线; 交叉分布; 交叉线; 树状数组

中图分类号: TP312 **文献标识码**: A

Crossing distribution of circuit wires between two regions using arborescence array

DENG Xin - guo[1], GAO Dong - ying[1], GUO Chao - zhen[1], XIAO Ru - liang[2]

(1. College of Mathematics and Computer Science, Fuzhou University, Fuzhou, Fujian 350116, China;
2. College of Software, Fujian Normal University, Fuzhou, Fujian 350117, China)

Abstract: Counting the crossing wires in the crossing distribution of circuit wires between two regions, the current algorithms using either linear list or dynamic programming to do so have the time complexity $O(n^2)$. In order to reduce the time complexity of the existing algorithms efficiently, a counting algorithm with the time complexity of $O(n\log n)$ using arborescence array is introduced in this paper. Furthermore, all crossing wires of every circuit wire are found with this algorithm. The effectiveness of the algorithm is illustrated through the theoretical analysis and by the corresponding experiment results.

Keywords: circuit wiring; crossing distribution; crossing wire; arborescence array

0 引言

一个电路包含一组模块和网. 在模块的边界上, 每个网指定点的一个子集, 称为终端. 布局问题是按照不同的工艺设计规则来互相连接网详细说明的模块. 由于问题的复杂性, VLSI 布局设计通常分为三个阶段: 布置、全局路由和详细路由. 在布置阶段, 电路模块根据几何学放置在阵列表面(芯片). 全局路由选择经过该区域的连线必须是需要的连接. 在全局路由阶段, 路由区域被分割成简单的子区域, 每个子区域称为基本区域, 确定每个网的全局布线路径. 详细路由是完成在每个布线区域中连线部分的设计. 在详细路由阶段, 确定每个路由区域的详细布线[1]. 交叉分布问题在详细路由之前出现. 研究表明, 互相交叉的网比那些没有交叉的网更难路由. 互相交叉的网的布局必须在两层以上实现, 因此需要大量的通孔[2].

在交叉分布问题中, 从一个布线通道开始, 通道的顶部和底部各有 n 个针脚. 图 1 给出了 $n = 10$ 的情况. 布线区域是图中的长方形区域, 针脚的序号从 1 到 n, 在通道的顶部和底部从左至右分布. 另外, 有[1, 2, 3, …, n]的一个排列 wire[ ]. 必须用一根电线将顶部的针脚 i 与底部的针脚 wire[i]连接起来. 在图 1 的例子中, wire[ ] = [8, 7, 4, 2, 5, 1, 9, 3, 10, 6]. 需要连接的 n 根线被编号为 1 到 n, 第 i 根线连接顶部的 i 和底部的 wire[i]. 当且仅当 $i < j$ 时, 连线 i 在连线 j 的左边.

收稿日期: 2012 - 07 - 01

通讯作者: 邓新国(1975 -), 副教授, E - mail: xgdeng@ fzu. edu. cn

基金项目: 福建省自然科学基金资助项目(2009J05142); 福州大学人才基金资助项目(0220826788); 福州大学科技发展基金资助项目(2011 - xq - 24)

图 1 显示了在布线区域中无论线路 9 和 10 如何设置，它们一定会在某一点相交. 人们不希望交叉的出现，以免出现短路，为此可以在交叉点放置绝缘物或将两条线路布设在不同层. 因此，计算交叉点数目成为两区域电路线交叉分布的核心问题.

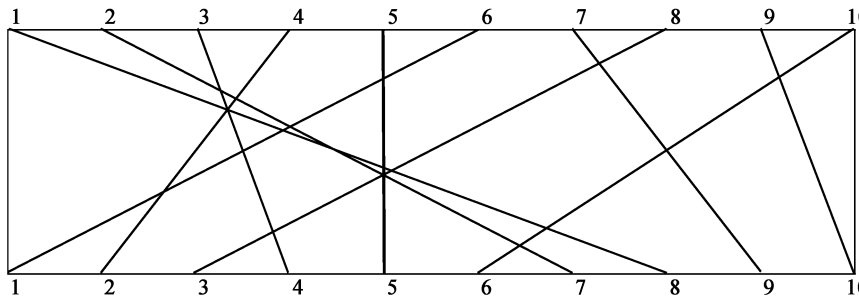

图 1　布线实例

Fig. 1　A wiring instance

每个交叉点用(i, j)表示，i 和 j 是通过该交叉点的两条线路. 识别所有交叉点的一种方法是检查线路的每一个(i, j)并且检验其中两条直线是否相交. 为避免对同一交叉点检验两次，可以要求 $i < j$(即交叉点(10, 9)与(9, 10)是一样的). 设 count[i] 是交叉线(i, j)的数量，$i < j$. 在图 1 中，count[9] = 1，count[10] = 0[3].

文献[3 - 4]分别采用线性链表和动态规划[5-7]计算两区域电路线交叉分布的数量. 然而，这两种算法的时间复杂度均为$O(n^2)$. 这里给出使用树状数组、时间复杂度为$O(n\log n)$的计数算法. 此外，还寻找每条布线的所有交叉线. 算法的有效性将通过理论分析和相应 C++ 程序的实验结果来阐明.

1　树状数组

定义 1　设 i 整除 2^k，i 不能整除 2^{k+1}，k 为非负整数，则称 2^k 是 i 的最小比特，记为 lowbit(i).

定义 2　设 $c[1]$，$c[2]$，…，$c[n]$ 是一个数组，$c[i]$ 称为点或数. 定义 $c[i]$ 的父节点是 $c[i + \text{lowbit}(i)]$. 称 c 为树状数组.

性质 1　若 i 是奇数，则 $c[i]$ 没有子节点. 以 $c[i]$ 为根的子树只有 1 个点.

性质 2　若 i 的最小比特是 2^k，则以 $c[i]$ 为根的子树有 2^k 个点，有 k 个子节点，它的 k 个子节点是(若 i 的二进制数是 ＊＊＊100000，以 k = 5 为例)：

$c[***10000]$，$c[***11000]$，$c[***11100]$，$c[***11110]$，$c[***11111]$

定义 3　设 $c[1]$，$c[2]$，…，$c[n]$ 是树状数组，$b[1]$，$b[2]$，…，$b[n]$ 是与之对应的一个 n 元数组. j = lowbit(i)，定义 $c[i]$ 的值为：

$$c[i] = b[i - j + 1] + b[i - j + 2] + \cdots + b[i] \tag{1}$$

即 $c[i]$是到 i 为止的 lowbit(i)个数的和.

例 1　设 n = 10，则：

$c[1]=b[1]$，$c[2]= b[1] + b[2]= c[1] + b[2]$

$c[3]=b[3]$，$c[4]= b[1] + b[2] + b[3] + b[4] = c[2] + c[3] + b[4]$

$c[5]=b[5]$，$c[6]= b[5] + b[6]= c[5] + b[6]$，$c[7]= b[7]$

$c[8]=b[1] + b[2] + b[3] + b[4] + b[5] + b[6] + b[7] + b[8]= c[4] + c[6] + c[7] + b[8]$

$c[9]=b[9]$，$c[10]= b[9] + b[10]= c[9] + b[10]$

例 1 可以看出 $c[i]$ 是 $b[i]$ 与 $c[i]$ 的子节点的值之和，如图 2 所示[8-10].

2　利用树状数组的两区域电路线交叉分布

图 2 是依定义 2、3 得到父、子结构的一颗确定的树，初始时是一颗空树，尚未放入需要计算的项目. 据此结构特征，再通过定义 c 的内涵以及具体问题所引入的数组 b 的内涵等，即可用来解决具体计算问

题. 利用树状数组处理两区域电路线交叉分布问题. 设数组 b 为树状数组中节点上所放元素的个数, 初始时 $b[i] = 0$, $i = 1, 2, \cdots, n$. 设数组元素 $c[i]$ 为树状数组中以节点 i 为根的树上所含元素的个数, 满足式(1). 定义 sum(i) 为放置在 i 以及 i 之前的节点所含元素的个数和, 可以利用最小比特高效计算. 首先建立一棵有 n 个节点的空树(树状数组). 接着, 顶部针脚从后往前扫描, 当前布线(i, wire[i]) 的交叉线数量 count[i] 为树状数组中节点 wire[i] 及其之前的节点所含元素的个数和. 然后, wire[i] 及其祖先节点的值都增加 1. 如此循环, 可以计算每条布线的交叉线数目. count[i] 是交叉线(i, j) 的数量, $i < j$. 因此 count[n] = 0. 也由于这个原因, 算法中没有显示用到数组 b[ ]. 然后, 利用交叉线定义和辅助矢量保存每条布线的交叉线.

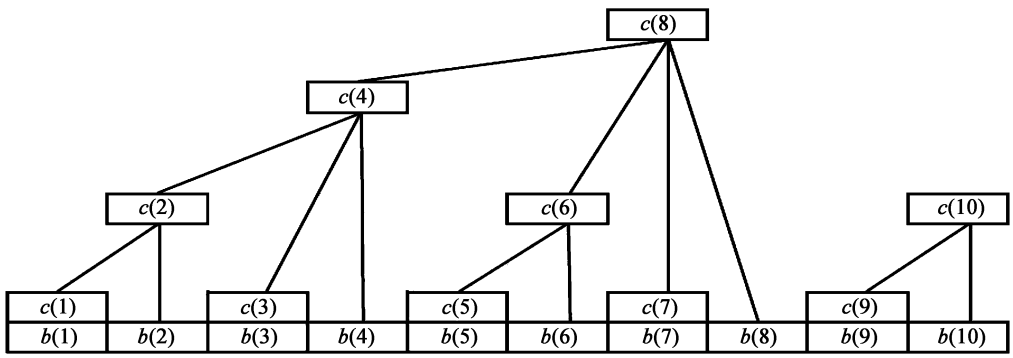

图 2 树状数组 c 和数组 b

Fig. 2 Arborescence array c and b

算法 计算每条布线(i, wire[i]) 的交叉线数量 count[i]、寻找每条线的所有交叉线 cross[i]. (i、wire[i] 分别表示每条布线顶部、底部针脚编号, $i = 1, 2, \cdots, n$).

```
K=0;        // 所有交叉线总和初始化
for(i=1; i<=n; i++)c[i]=0;        //初始化
for(i=n; i>=1; i--)  // 顶部针脚从后往前扫描
{   count[i] = sum(wire[i]);        //当前布线的交叉线个数, 即式(1)中数组 b[ ]前 p 项和
plus(wire[i], 1);        // wire[i]及其祖先节点的值都增加 1
K += count[i];        // 所有交叉线总和
aided_set.insert((i, wire[i]));        //在集合 aided_set 中插入第 i 条布线
for (cit = aided_set.begin(); cit != aided_set.end(); ++cit)
        {   if (cit->second < wire[i])        // j>i, wire[j]< wire[i], 即布线 j 和 i 交叉
                {cross[i].push_back(cit->first);        //在矢量 cross[i]最后添加一条交叉线}
            else
                break;    }   }
```

2.1 算法时间复杂性

树状数组是一个查询和修改复杂度都为 $O(\log(n))$ 的数据结构, 并支持随时修改某个元素的值, 可以高效地进行区间统计. 因为树的高度不会超过 $\log n$, 所以当修改 $b[i]$ 的值时, 可以从节点 $c[i]$ 向根节点一路上溯, 调整这条路上的所有 c[ ]即可, 这个操作的复杂度在最坏情况下就是树的高度即 $O(\log n)$. 另外, 对于求数列的前 n 项和, 只需找到 n 以前的所有最大子树, 把其根节点的 c 加起来即可. 这些子树的数目是 n 在二进制时 1 的个数, 或者说是把 n 展开成 2 的幂方和时的项数, 因此, 求和操作的复杂度是 $O(\log n)$. 这里, 计算一条线、所有线的交叉线数量的时间复杂度分别为 $O(\log n)$、$O(n\log n)$.

用矢量保存、输出每条线的交叉线, 时间复杂度为 $O(n^2)(n + (n-1) + \cdots + 2 + 1 = n(n+1)/2)$. 因此, 如果只计算交叉线的总数, 那么算法的复杂度为 $O(n\log n)$; 如果还保存、输出每条线的所有交叉线, 时间复杂度为 $O(n^2)$.

2.2　计数实例演示

计算每条布线的相交数量 count[i]（$1 \leqslant i \leqslant n$）时，检查顶部针脚的次序为 n，$n-1$，…，1，即从后往前扫描（表 1）.

表 1　利用树状数组计算图 1 中布线的交叉线数量

Tab. 1　Counts cross wires in Figure 1 using arborescence array

顶部针脚 i	底部针脚 wire[i]	树状数组求和 sum(wire[i])	布线(i, wire[i])的交叉线数量 coum[i]	wire[i]及其祖先节点的值加 1 plus(wire[i], 1)	$c[i]$ i=1 2 3 4 5 6 7 8 9 10
10	6	sum(6) =c[6]+c[4] =0+0=0	0	plus(6, 1) 节点 6、8 的值加 1 c[6]=1, c[8]=1	0 0 0 0 0 1 0 1 0 0
9	10	sum(10) =c[10]+c[8] =0+1=1	1	plus(10, 1) 节点 10 的值加 1 c[10]=1	0 0 0 0 0 1 0 1 0 1
8	3	sum(3) =c[3]+c[2] =0+0=0	0	plus(3, 1) 节点 3、4、8 的值加 1 c[3]=1, c[4]=1, c[8]=2	0 0 1 1 0 1 0 2 0 1
7	9	sum(9) =c[9]+c[8] =0+2=2	2	plus(9, 1) 节点 9、10 的值加 1 c[9]=1, c[10]=2	0 0 1 1 0 1 0 2 1 2
6	1	sum(1) =c[1]=0	0	plus(1, 1) 节点 1、2、4、8 的值加 1 c[1]=1, c[2]=1, c[4]=2, c[8]=3	1 1 1 2 0 1 0 3 1 2
5	5	sum(5) =c[5]+c[4] =0+2=2	2	plus(5, 1) 节点 5、6、8 的值加 1 c[5]=1, c[6]=2, c[8]=4	1 1 1 2 1 2 0 4 1 2
4	2	sum(2) =c[2]=1	1	plus(2, 1) 节点 2、4、8 的值加 1 c[2]=2, c[4]=3, c[8]=5	1 2 1 3 1 2 0 5 1 2
3	4	sum(4) =c[4]=3	3	plus(4, 1) 节点 4、8 的值加 1 c[4]=4, c[s]=6	1 2 1 4 1 2 0 6 1 2
2	7	sum(7) =c[7]+c[6]+c[4] =0+2+4=6	6	plus(7, 1) 节点 7、8 的值加 1 c[7]=1, c[s]=7	1 2 1 4 1 2 1 7 1 2
1	8	sum(8) =c[8]=7	7	plus(8, 1) 节点 8 的值加 1 c[s]=8	1 2 1 4 1 2 1 8 1 2

对于图 1 中的例子，在表 1 中首先检查顶部针脚 10，底部针脚 wire[10]=6，count[10]=sum(6)=c[6]+c[4]=0（右边没有布线），plus(6, 1)使树状数组节点 6 及其父节点 8 的值均增加 1，即 c[6]=c[8]=1. 然后检查顶部针脚 9，底部针脚 wire[9]=10，count[9]=sum(10)=c[10]+c[8]=1（右边有 1 条布线与其相交），plus(10, 1)使树状数组节点 10 的值增加 1，即 c[10]=1. 类似地，第 10 次检查顶部针脚 1，底部针脚 wire[1]=8，count[1]=sum(8)=c[8]=7（右边有 7 条布线与其相交），plus(8, 1)使树状数组节点 8 的值增加 1，即 c[8]=8. 由于利用了底部针脚在树状数组中的位置，树状数组能够统计每条布线的交叉线数量. 顶部针脚从后往前扫描相当于 $j > i$，树状数组求和相当于统计 wire[j] < wire[i]. 这正好与当前布线(i, wire[i])后面与其相交的布线(j, wire[j])的定义一致.

3　C^{++}实现

采用自顶向下的模块化方法[11]把 C^{++}程序划分为三个部分：输入、求解和输出. 每个部分用一个模

块实现. 第四个模块显示欢迎信息、软件名称及作者信息,这个模块与其他三个模块之间没有任何直接关系,主要用于增强程序界面的友好性. 求解模块包含计算与每条线交叉的线的数量以及这些交叉线. 输出模块输出每条线以及求解模块的处理结果. 求解模块进一步细分为两个模块:初始化和树状数组模块. 也可以添加欢迎模块显示程序的功能. 尽管不直接与问题相关,但是欢迎模块的使用增强了程序的用户友好性. 用一个根模块来调用这五个模块,调用的次序是:欢迎模块、输入模块、初始化模块、树状数组模块和输出模块. 程序的模块结构如图 3 所示. 每个模块均独立编写. 根模块将被编写成一个 main 函数,欢迎模块、输入模块、初始化模块、树状数组模块和输出模块将分别对应于一个函数. 至此,程序如图 3 所示. 欢迎函数"welcome"解释了 C^{++}程序的整体功能. 输入函数"input"通知用户首先输入针脚的数量,接着输入通道顶部针脚对应的底部针脚排列. 这里从"input. txt"文本文件导入数据,求解后的结果输出到文本文件"output. txt". 图 4 详细描述了动态树状数组类.

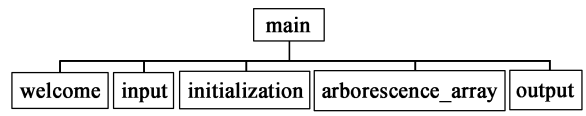

图 3 模块结构

Fig. 3 Modular structure

```
// 动态树状数组类
template < class T >
class ArborescenceArray {
private:
  int n; // 树状数组大小
  vector < T > c; // 矢量
  int lowbit( int i) { // 计算 i 的最小比特 2^k
      return i & (i ^ (i - 1)); // 按位与 & 和按位异或^是 c++ 语言的位运算符; }
public :
  void initialize( int _n) { // 初始化
      n = _n;
      c. assign( _n + 1, 0); // void assign( size_type num, const TYPE &val ); 赋 num 个值为 val 的元素到 vector 中; }
  void plus( int p, T num) { // 如果某个数值发生改变,例如 b[p]的值增加了 num,那么 c[p]和 c[p]的祖先节点的值都要增加 num
      while (p < = n) {
          c[p] + = num;
          p + = lowbit(p);     } }
T sum( int p) {// 计数组 b[ ]前 p 项和. c[i] = b[i-j+1] + b[i-j+2] + … + b[i], j = lowbit(i).
    T sum = 0;
    while (p > 0) {
        sum + = c[p];
        p - = lowbit(p);  }
    return sum;
  }     };
```

图 4 树状数组类

Fig. 4 Arborescence array class

变量 n 是树状数组大小. 树状数组用 c 表示. 数组写法为 $c[N]$,当 n 小时,会浪费空间,当 n 大时,数组会越界. 矢量和数组相比的优点是元素个数可以根据输入进行调整. 初始化函数 initialize(int _n)把树状数组的元素赋初值 0. 若某个数值发生改变,如 $b[p]$的值增加了 num,则 $c[p]$和 $c[p]$的祖先节点的值都要增加 num(参考定义 3). 这是通过 plus(int p, T num)函数来实现的. 函数 sum(int p) 计算数组 $b[\ ]$的前 p 项和.

图 5 是树状数组函数 arborescence_array(),计算每条线的交叉线数量以及这些交叉线. 计算每条线的交叉线数量的方法是在树状数组中从后往前扫描,动态求和. 先记录与当前连线相交的连线个数,然后把树状数组元素 $c[\text{wire}[i]]$和 $c[\text{wire}[i]]$的祖先结点的值都增加 1. 计算每条线的交叉线的方法是采用辅助

集合 set < pair < int, int > 、SetCompare > aided_set，插入布线的上、下针脚对. 集合是随着不断地插入而动态变化的. cross 是二维整数矢量 vector < vector < int > > . cross[i]记录第 i 条布线的所有交叉线.

由于篇幅的原因，这里省略了函数“welcome”、“input”、“output”和“main”. 下一节将会阐明这些函数的效果.

```
void arborescence_array( ) {// 利用树状数组计数
  int i;
  for (i = n; i > = 1; --i)
  {  // 从后往前扫描，动态求和
    count[i] = arborescence_array_object. sum( wire[i]);// 记录当前布线的交叉线个数
    arborescence_array_object. plus( wire[i], 1); //c[wire[i]]及其祖先节点的值都增加 1
    // 以下记录第 i 条布线的所有交叉线
    aided_set. insert( make_pair(i,wire[i])); // 在集合中插入当前布线
    cross[i]. clear( );
    /* begin( ) 返回指向第一个元素的迭代器; end( ) 返回指向最后一个元素的迭代器 */
    // 保存 wire[i]的所有交叉线
    for (set < pair < int, int > , SetCompare > ::const_iterator cit = aided_set. begin( ); cit ! = aided_set. end( ); ++cit) ;   {
          if ( cit - > second < wire[i]) ;   {
            cross[i]. push_back( cit - > first); //在 vector 最后添加一个元素
          } else
            break;
    }     }   }
```

图 5　树状数组函数

Fig. 5　Arborescence_array function

4　实验结果

图 6 是 C++ 程序运行图 1 布线实例后的输出结果. 首先，欢迎函数“welcome”说明程序的功能. 其次，输入函数 “input”提示用户输入布线数据. 接着，后台利用树状数组、矢量运行后输出每条线的交叉线数量以及具体的交叉线. 最后，统计所有线的交叉线总数.

```
////////////////////////////////////////////////////////////////////
This program uses the data structure of Arborescence Array
to handle the crossing distribution problem of circuit wires.
////////////////////////////////////////////////////////////////////
Input the size of wires
Input the array of wiring permutation
i: the number of wire
count [i]: the amount of wires crossing with wire i
Crossings: Wires crossing with wire i
i    count[i]    Crossings                  i    count[i]    Crossings
1      7         2  3  4  5  6  8  10       6      0
2      6         3  4  5  6  8  10          7      2         8  10
3      3         4  6  8                    8      0
4      1         6                          9      1         10
5      2         6  8                       10     0
The total number of crossings K is 22
```

图 6　交叉布线输出

Fig. 6　Output of crossings

5 结语

在两区域电路线交叉分布问题中计算每条线的交叉线数量、统计所有线的交叉线总和，采用线性表或者动态规划算法的时间复杂度均为 $O(n^2)$，利用树状数组可以把时间复杂度降为 $O(n\log n)$. 此外，这里还利用矢量数据结构输出了每条线的具体交叉线，输出的时间复杂度为 $O(n^2)$. 理论分析和相应 C++ 程序的实验结果表明了算法的有效性.

多区域之间的电路线交叉分布问题仍然难以解决. 接下来的工作进一步研究多区域之间电路线交叉分布的高效算法，最大限度地减少布线拥塞.

参考文献：

[1] Wolf W. Modern VLSI design: system - on - chip design[M]. 3rd ed. New Jersey: Pearson Education, 2003: 510 - 522.

[2] Song Xiao - yu, Wang Yu - ke. On the crossing distribution problem[J]. ACM Transaction on Design Automation of Electronic Systems, 1999, 4(1): 39 - 51.

[3] Sahni S. Data structures, algorithms, and applications in C++[M]. 2nd ed. New Jersey: Silicon Press, 2004: 546 - 553.

[4] Wang Xiao - dong. Algorithm design and analysis[M]. 3rd ed. Beijing: Publishing House of Electronics Industry, 2007: 74 - 76.

[5] Skiena S S. The algorithm design manul[M]. 2nd ed. Berlin: Springer, 2008: 273 - 315.

[6] Han Zhu, Liu K J R. Resource allocation for wireless networks: basics, techniques, and applications[M]. Cambridge: Cambridge University Press, 2008: 167 - 170.

[7] Pandey H M. Designanalysis and algorithm[M]. New Delhi: Firewall Media - Laxmi Publications, 2008: 258 - 286.

[8] 周培德. 计算几何：算法设计与分析[M]. 4 版. 北京：清华大学出版社, 2011.

[9] 刘洋. 基于纹理合成的图像修复与基于分形的图像分割方法的研究与应用[D]. 长春：吉林大学, 2010: 71 - 73.

[10] 周娟, 曹义亲, 谢昕. 基于树状数组的逆序数计算方法[J]. 华东交通大学学报, 2011, 28(2): 45 - 49.

[11] Pressman R S. Software engineering: a practitioner's approach[M]. 7th ed. New York: McGraw - Hill, 2009: 243 - 538.

(责任编辑：沈芸)